

## **LENGUAJE DE COMANDOS – SCRIPTS .CMD .BAT**

¿Qué es un script en lenguaje de comandos? No es nada más que un fichero de texto, que puede generarse con el simple cuaderno de notas, y cuya extensión es .bat o .cmd. Su contenido son los comandos que ejecutaríamos en una consola de comandos (cmd) y cuyo fin es evitar las tareas repetitivas que podríamos realizar en una consola de comandos.

Aunque esta es la definición clásica, no debemos perder de vista que desde una consola de comandos

podemos realizar, mediante comandos, prácticamente todo lo que la imaginación nos permita. Todo lo que se configura, o las tareas de mantenimiento que realizamos en Windows se pueden hacer desde una consola de comandos. Igualmente existen muchos comandos que son sólo de consola.

Revisemos un poco la tipología de los comandos: un comando es “algo” que o bien entiende directamente el shell (el intérprete de comandos, en este caso el cmd.exe) o bien es un programa con

extensión .com o .exe -o incluso un visual basic- que no use la interfaz gráfica y que por tanto esté orientado a la consola. Un ejemplo clásico es el programa ipconfig. Este programa (de consola) nos da la configuración TCP/IP de la máquina. O bien el programa ping.

### **ENTORNO DE UN PROGRAMA**

Todos los sistemas operativos, y Windows no es una excepción, tienen un área de datos llamada “entorno”. No es nada más que un área donde se guardan ciertas variables con su contenido.

Es importante entender que cada programa de Windows tiene su entorno (igual o diferente a otro programa). Los entornos se heredan. Cada programa (y el propio intérprete de comandos, cmd.exe, es un programa más) cuando se lanza, “hereda” dicho entorno. Por heredar, no quiere decir que “use”

el mismo que el programa padre, sino que al lanzarse, el “loader” -cargador- del sistema operativo realiza una copia del entorno padre en una nueva área de datos y al lanzar el programa le da como dirección del área de entorno esa nueva copia del entorno del “padre”.

En otras palabras, cualquier modificación en las variables de entorno dentro de un programa no afecta al sistema ni al resto de programas, ya que lo que haría es modificar su propio entorno: la copia del entorno original del programa padre.

El sistema operativo al cargarse predefine ya una serie de variables de entorno. Podemos verlas, bien

con botón derecho en Mi PC / propiedades / pestaña de opciones avanzadas / botón de variables de entorno, o bien de una manera más simple, lanzando el intérprete de comandos (cmd.exe) y tecleando el comando “set” (sin comillas).

NOTA: Realmente, aunque lo veamos en conjunto, existen dos entornos: uno del sistema y uno de usuario, pero la visión de ambos es el conjunto de los dos.

Acabamos de ver nuestro primer comando: “set”. Este comando nos permite no sólo ver todas las variables, sino también definir, cambiar, borrar su contenido y algunas opciones más.

Si lo ejecutamos por consola, nos dará algo similar a:

```
ALLUSERSPROFILE=C:\Documents and Settings\All Users
APPDATA=C:\Documents and Settings\mi usuario\Datos de programa
CommonProgramFiles=C:\Archivos de programa\Archivos comunes
COMPUTERNAME=MIMÁQUINA
ComSpec=C:\WINDOWS\system32\cmd.exe
FP_NO_HOST_CHECK=NO
HOMEDRIVE=C:
HOMEPATH=\Documents and Settings\mi usuario
NUMBER_OF_PROCESSORS=1
OS=Windows_NT
Path=C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;.....etc
```

```
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH
PROCESSOR_ARCHITECTURE=x86
PROCESSOR_IDENTIFIER=x86 Family 6 Model 7 Stepping 3, GenuineIntel
PROCESSOR_LEVEL=6
PROCESSOR_REVISION=0703
ProgramFiles=C:\Archivos de programa
...etc...
```

Fijémonos que la estructura es: nombre de variable=contenido de la variable

Las variables, dentro de una consola de comandos o bien dentro de un script se referencian para poder ver su contenido encerradas entre símbolos de %. Por ejemplo, en el caso anterior, para ver el contenido de la variable COMPUTERNAME, simplemente podemos ejecutar:

```
echo %COMPUTERNAME%
```

Esto nos dará como resultado:

```
MIMÁQUINA
```

Igualmente podríamos cambiarlo con el comando set citado anteriormente:

```
set COMPUTERNAME=nuevoNOMBRE
```

Pero...

\* ¿Esto realmente tiene el efecto de cambiar el nombre del ordenador? por supuesto que no. Esto sólo cambia el contenido de dicha variable. ¿Dónde lo cambia? pues tal y como hemos comentado anteriormente, lo cambia en el entorno del programa, es decir, en la copia del entorno original heredado por el programa. En nuestro caso, al ejecutarlo desde un cmd.exe, implica que cambia el entorno del cmd.exe (y sólo de él, es decir, si tuviésemos lanzadas dos consolas de comandos, cada una de ellas con cmd.exe, sólo se cambiaría en el cmd.exe que hayamos ejecutado ese comando “set”). Y ¿para que puede servir? simple, si recordamos que un programa hereda una copia del entorno del programa que lo lance, esto implicará que todo lo que lancemos desde esa consola de comandos, tendrá el contenido de esa variable modificado.

\* ¿Cómo podemos crear una nueva variable de entorno? tan simple como darle un nombre y asignarle su contenido. Por ejemplo:

```
set nuevo=prueba
```

Esto creará si no existe, o modificará el contenido si existiese, de una variable de entorno llamada “nuevo”, y le asignará el contenido de “prueba”.

\* ¿Podemos asignar a una variable el contenido de otra?: sí, por supuesto. Simplemente recordando que el contenido de una variable es precisamente el nombre de la variable encerrada entre símbolos %.

```
set otra=%nuevo%
```

Esto creará la variable “otra” con el contenido de la variable “nuevo”, el cual era el texto “prueba”.

Si

después de ejecutar el comando anterior realizamos:

```
echo %otra%
```

su resultado será:

```
prueba
```

\* ¿Cómo podemos borrar -eliminar- una variable? simplemente no asignando nada.

```
set otra=
```

En este caso borrará la variable. Si ahora damos el comando “set” que nos muestra todas las variables, la variable “otra” ya no aparecerá.

\* ¿Puedo concatenar textos con variables en una asignación?: Sí, por supuesto. Por ejemplo:

```
set otra_de_nuevo=Esto es una %nuevo% de concatenación
```

El resultado de lo anterior será que la variable “otra\_de\_nuevo” contendrá el valor “Esto es una prueba de concatenación”.

\* ¿Podemos tener contenidos numéricos y no sólo alfabéticos, y por tanto realizar operaciones matemáticas con el comando set? sí, se puede. La manera es con el modificador /a. Pongamos un ejemplo:

```
set n=234 (asignar 234 a la variable "n")
```

```
set /a i=%n%/4
```

Veremos que esto nos devuelve por pantalla el valor 58 (es decir la división de 234/4) y además se lo

asigna a la variable "i".

Es necesario el modificador /a, ya que si no lo hubiésemos puesto, la expresión:

```
set i=%n%/4
```

no hubiese realizado la operación matemática y se hubiese quedado con el contenido (como literal) de:

```
234/4
```

es decir, una cadena de caracteres con ese contenido y no con el resultado matemático de la división.

Fijémonos, por curiosidad en este último caso y se puede ir realizando una prueba en una consola de comandos.

Realicemos:

```
set n=234
```

```
set i=%n%/4
```

```
echo %i%
```

veremos entonces:

```
234/4
```

Si ahora realizamos:

```
set /a j=%i%
```

¿Que contendrá? ¿Contendrá el literal de texto "234/4"? ¿O sólo "234" ya que el carácter "/" no es numérico? ¿O contendrá "58"?

Realizadlo como prueba

La asignación a una variable numérica, puede ser decimal, hexadecimal u octal.

Podemos hacer:

```
set /a i=14
```

```
set /a i=0x0E
```

en ambos casos contendrá el decimal 14 (recordemos que el hexadecimal 0E equivale al decimal 14).

O bien, si queremos introducir un numero en octal, simplemente lo empezamos por cero.

```
set /a i=021
```

hay que tener cuidado con esto, ya que el octal 021 es el decimal 17. Podéis comprobarlo en pantalla

simplemente tecleando la línea anterior y viendo el resultado. Siempre los resultados nos los expresará en decimal.

Esto puede ser útil para convertir directamente de hexadecimal o bien de octal a decimal de una manera rápida.

\* Ya que hemos empezado con el comando "set" vamos a finalizar todo lo que dicho comando puede

hacer. Además de lo anterior:

```
set p
```

nos mostrará en pantalla "todas" las variables de entorno que empiecen por p.

Y para finalizar el modificador /p. Es decir, una sintaxis del estilo:

```
set /p variable=[literal]
```

lo que hace es muestra el "literal" en pantalla y el cursor se queda a continuación esperando que metamos un dato. Cuando lo metemos y tecleamos "intro", lo que hayamos tecleado se asignará a la variable de entorno definida en el set. Por ejemplo:

```
set /p dato=Introduce datos:
```

Nos mostrará por pantalla:

Introduce datos: \_

y cuando introduzcamos los datos los asignará a la variable "dato".

Esto es válido realizarlo en un script. En ese momento se detendrá la ejecución del script y nos pedirá

datos por pantalla. Al pulsar la tecla "intro", se los asignará a dicha variable y continuará la ejecución del script.

### CARACTERES ESPECIALES

Hay ciertos caracteres "especiales" que hay que usar con extrema precaución: por ejemplo, evitar usar, a no ser que lo necesitemos explícitamente, caracteres reservados como &, >, <, |, %, =, ^.

Pongamos un ejemplo. El carácter & ("and" en inglés, es decir "y" en castellano) se interpreta como un "y", por tanto, ejecuta lo que hay antes del & "y" además a continuación, ejecuta lo que hay después del &. Es decir, si ejecutásemos:

```
set var=a&a
```

nos dará un error que veremos por pantalla:

"a" no se reconoce como un comando interno o externo, programa o archivo por lotes ejecutable

\* ¿Qué ha hecho el sistema? Ha ejecutado el comando que hay antes del &, es decir:

```
set var=a
```

Por tanto le ha asignado a la variable de entorno "var" el contenido "a". A continuación, el intérprete

de comandos intenta ejecutar lo que hay después del &, es decir intenta interpretar el comando "a". Como "a" no es un comando reconocido, ni un .exe, ni un .com, ni el nombre de otro script nos dará el error mostrado anteriormente. Curiosamente:

```
set var=a&cmd
```

¿qué hace? Ejecutadlo y ved el resultado.

Pero... ¿y si realmente queremos asignar el contenido del literal "a&a" a la variable "var", cómo lo hacemos? Simplemente anteponiendo el carácter ^ a cualquiera de los caracteres especiales. En nuestro caso deberíamos haber hecho:

```
set var=a^&a
```

Si posteriormente ejecutamos un:

```
echo %var%
```

nos seguirá mostrando un error, ya que aunque realmente tiene el contenido "a&a", nos mostrará "a"

e intentará ejecutar lo que va a continuación: intentará la ejecución de un comando llamado "a".

Para ver que realmente tiene el contenido solicitado, podemos hacer dos cosas:

```
set v
```

el nos mostrará el contenido de todas las variables que empiezan por "v" y podremos comprobar que

la variable "var" contiene lo que realmente deseábamos. O bien, redirigir la salida a un archivo (los símbolos de redirección y su significado los veremos más adelante).

NOTA: Un error bastante corriente cuando se usan caracteres de asignación y concatenación (&) en la misma línea puede ser:

```
set var=prueba & set var1=otra_prueba
```

¿Cuál es, o puede ser, el error cometido? pues... recordemos que todo lo anterior al & se ejecuta como

un comando. Es decir: ¿qué hemos asignado a la variable "var"? podemos suponer que "prueba".

Pero no es así, hemos asignado todo lo anterior al &, es decir "prueba ", con un espacio en blanco al final. Esto nos puede causar problemas posteriores si lo que queríamos hacer era asignar sólo "prueba".

Podemos verificarlo tecleando entre algún delimitador, por ejemplo un \*:

```
echo *%var%*
```

Lo cual nos mostrará un \*, el contenido de var y otro \*. Obtendríamos:

\*prueba \*

con el fatídico (en este caso) carácter en blanco al final.

—

## OPERACIONES CON VARIABLES DE ENTORNO

Hemos visto hasta el momento que las variables de entorno no son nada más que cadenas de caracteres, las cuales mediante algún modificador especial en el comando set (/a) pueden tratarse como numéricas para realizar algunas pequeñas operaciones matemáticas.

Vamos a ver primero las funciones que podemos aplicar a las variables de caracteres y en segundo lugar las operaciones matemáticas que podemos hacer así como su precisión cuando el contenido es numérico. Igualmente veremos las llamadas variables dinámicas de entorno: variables que aunque no

veamos mediante el comando “set” pueden usarse en instrucciones de comandos.

Tengamos siempre presente que los caracteres “especiales” deben llevar el símbolo ^ precediéndolos.

Por ejemplo, un error que viene en la propia ayuda de Windows del comando set, es que la propia ayuda nos dice: “El comando SET no permitirá que un signo de igual sea parte de una variable”.

Esto

no es “del todo” verdad. Si hacemos:

```
set var=a^=b
```

la variable “var” quedará con el contenido “a=b”. Se puede comprobar si a continuación de lo anterior ejecutamos:

```
set v
```

### 1) FUNCIONES SOBRE VARIABLES DE CARACTERES

\* Extraer una subcadena:

```
%var:~n,m%
```

Esto nos extrae de la variable “var”, la subcadena desde la posición “n” con longitud “m”. “n” es el desplazamiento empezando a contar desde 0, por tanto, la primera posición es la 0, la segunda la 1, etc.

Pongamos un ejemplo:

```
set var=123456789
```

```
echo %var:~1,4%
```

nos mostrará la subcadena de “var” que va desde la posición 2 (recordemos: con desplazamiento +1 ya que empieza por 0) y con longitud 4. Es decir, veremos: “2345”.

Tanto n como m, son opcionales. Es decir, por ejemplo:

```
%var:~1%
```

nos mostrará desde la posición 2 (offset de 1), hasta el \*final\* de la variable (ya que no se ha especificado longitud). Por tanto, mostrará en el ejemplo anterior: “23456789”.

Igualmente:

```
%var:~,5%
```

al no especificarse posición se asume desde el principio de la cadena y por tanto nos mostrará 5 caracteres iniciales. En nuestro caso “12345”.

NOTA: si “n” es un número negativo, se refiere a la longitud de toda la cadena menos ese número.

Por ejemplo:

```
set var=123456789
```

```
echo %var:~-2%
```

nos mostrará los dos últimos caracteres de la cadena, es decir “89”.

Igualmente, si “m” es negativo, se refiere a \*toda\* la cadena menos “m” caracteres.

```
set var=123456789
```

```
echo %var:~,-2%
```

nos mostrará todo menos los dos últimos caracteres de la cadena, es decir “1234567”.

\* Sustituir dentro de una cadena, un literal por otro:

```
%var:str1=str2%
```

Este comando buscará todas las ocurrencias de la subcadena “str1” dentro de “var” cambiándolas por “str2”.

Por ejemplo:

```
set var=hola cómo estás
```

Ejecutad para verlo:

```
echo %var:cómo=que tal%
```

Igualmente podríamos asignárselo de nuevo a “var” o bien a otra nueva variable de entorno. Por ejemplo:

```
set var=%var:cómo=que tal%
```

con lo cual el contenido habría quedado modificado.

NOTA1: puede omitirse “str2”. En ese caso la cadena str1 dentro de la variable var quedará eliminado. Por ejemplo:

```
set var=12345xx6
```

```
set var=%var:xx=%
```

```
echo %var%
```

NOTA2: se cambiarán todas las ocurrencias de “str1”. Si “str1” figura varias veces se cambiará todas las veces en la cadena de caracteres.

## 2) FUNCIONES MATEMÁTICAS SOBRE VARIABLES DE ENTORNO

Con el modificador /a en el comando set podemos realizar operaciones matemáticas \*simples\* sobre variables.

Debemos primero hablar de la precisión: sólo se opera con números enteros. Por tanto la división de 5/2 nos dará como contenido un 2. Igualmente la máxima precisión es 2<sup>32</sup>.

En general la sintaxis es:

```
set /a var=expresión matemática.
```

Por “expresión matemática” puede entenderse cualquiera de las siguientes y en orden de precedencia

de mayor a menor (extraído de la propia ayuda del comando set):

() – agrupar

! ~ - - operadores unarios

\* / % – operadores aritméticos

+ - - operadores aritméticos

<< >> – desplazamiento lógico (bit a bit)

& – bit a bit y

^ – bit a bit exclusivo o

| – bit a bit

= \*= /= %= += -= – asignación

&= ^= |= <<= >>=

, – separador de expresión

Si usamos una variable de entorno dentro de una expresión matemática y dicha variable no existe, será asumido que contiene un cero.

Veamos los matices de lo anterior, ya que si no tenemos en cuenta algunas matizaciones ya comentadas en este documento habrá expresiones que nos darán error.

**IMPORTANTE:** Volvamos a recordar que hay símbolos especiales que no pueden escribirse directamente, por ejemplo: &, <, >, |, o incluso ^. Cada uno de ellos debe ir precedido por un ^. Es decir, el bit a bit exclusivo ^, debe escribirse como ^^ ya que si no el intérprete de comandos lo interpretará erróneamente.

Veamos ejemplos de “asignaciones”.

```
set /a x=x+1
```

podemos sustituirlo (abreviarlo) por una notación similar al C o C++, es decir:

```
set /a x+=1
```

(a la variable “x” sumarle 1 y asignárselo a “x”)

Lo mismo:

set /a x=x-1 es equivalente a: set /a x=-1

set /a x=x\*23 es equivalente a: set /a x\*=23

etc... para el resto de operaciones matemáticas de asignación citadas anteriormente.

Veamos el siguiente problema: ¿qué solución tiene?

set /a x=8, y=2, z=16

set /a z=(x-y)/2+5^<<2

Quizá no sea tan trivial el ver que el resultado de lo anterior es -16. Intentad razonarlo.

TIP: Aunque no lo he encontrado documentado, pueden hacerse operaciones del tipo:

set /a x=(y=3+4)\*(z=2+1)

x vale 21, y vale 7, z vale 3

### 3) VARIABLES DINÁMICAS DE ENTORNO

Son variables que aunque no veamos mediante el comando “set” tienen contenido que va variando o puede variar dinámicamente.

Un ejemplo de esto es la variable %TIME%. Cada vez que ejecutemos:

echo %TIME%

nos dará dinámicamente la hora del sistema.

Estas variables son (autoexplicativas desde la ayuda del comando set):

%CD% – se expande a la cadena del directorio actual.

%DATE% – se expande a la fecha actual usando el mismo formato que el comando DATE.

%TIME% – se expande a la hora actual usando el mismo formato que el comando TIME.

%RANDOM% – se expande a un número decimal aleatorio entre 0 y 32767.

%ERRORLEVEL% – se expande al valor de NIVEL DE ERROR actual

%CMDEXTVERSION% – se expande al número actual de versión de las extensiones del comando del procesador (lo veremos posteriormente)

%CMDCMDLINE% – se expande a la línea de comando original que invocó el Procesador de comandos.

### PRECAUCIONES EN LA EXPANSIÓN DE VARIABLES DE ENTORNO

Aunque lo veremos en detalle posteriormente, vamos a introducir el por qué algunas veces aparecen las variables de entorno en un script encerradas entre símbolos “!” en vez de los clásicos “%” y cuándo debemos usar el símbolo “!”.

ATENCIÓN: El intérprete de comandos lee una línea de ejecución completa y lo primero que hace es sustituir las variables de entorno antes de procesarla.

Esta máxima debemos tenerla presente siempre. Veamos un ejemplo que, aunque aparentemente está

bien escrito, nos producirá siempre resultados anómalos o no esperados:

set var=antes

if %var%==antes (set var=despues&if %var%==despues echo “Cambiado a despues”)

Hemos introducido el comando “if”. Realmente un “if” no es nada más que un “si” condicional en castellano. Es decir: si se cumple una expresión, entonces se ejecuta lo que va a continuación.

Si además ponemos la palabra reservada “else”, esto se ejecutará en caso contrario. Recordemos igualmente que los paréntesis tienen precedencia.

Veamos entonces el conjunto de las instrucciones anteriores: asignamos “antes” a la variable “var”.

En la segunda línea preguntamos: Si la variable “var” contiene “antes” -que sí que es verdad-, ejecutaremos: asignar a la variable “var” la palabra “despues” y además a continuación comprobar que si tiene el contenido de “despues” -lo cual parece que es evidente puesto que acabamos de hacerlo- sacaremos por pantalla el texto “Cambiado a despues”.

Nos parece por tanto obvio que lo que vamos a ver después de ejecutar los comandos anteriores es “Cambiado a despues”.

Pues bien, si lo hacemos...sorprendentemente veremos que no sale nada. ¿No lo cambia?

Realmente

lo cambia, porque si a continuación de lo anterior ejecutamos:

```
echo %var%
```

veremos que contiene “despues”. Entonces ¿por qué no ha ejecutado correctamente lo que tenemos después del & en la línea anterior?

Volvamos a leer ahora despacio el párrafo que he comenzado anteriormente por “ATENCIÓN”.

Repasemos lo que hace por defecto el intérprete de comandos: lee la línea \*entera\* y en ese momento el contenido de las variables de entorno queda sustituido por el contenido real. A continuación, ejecuta la línea entera. Es decir lo primero que hace el intérprete de comandos es la “expansión” de la línea, y por tanto, antes de ejecutarla, quedaría:

```
if antes==antes (set var=despues&if antes==despues echo “Cambiado a despues”)
```

es evidente... que el segundo IF no lo ejecutará nunca.

Este es el funcionamiento por defecto del intérprete de comandos. Si queremos que no funcione así, y que haga dinámicamente la expansión de las variables (y no “antes” de ejecutar la línea de comando) hay que ejecutar dos cosas:

1) Arrancar el intérprete de comandos con el parámetro /V:ON para indicarle que debe expandir las variables. (Si estamos en un script, colocar como primera línea del script: setlocal ENABLEDELAYEDEXPANSION. Lo veremos más en detalle posteriormente)

2) Referenciar las variables que deben ser expandidas en ese momento, y no en el momento de leer la

línea por primera vez mediante los símbolos de “!” en vez de “%”.

En nuestro caso, y para verlo más claro:

1) En inicio-ejecutar arrancamos el intérprete de comandos como:

```
cmd /V:ON
```

2) Ejecutamos:

```
set var=antes
```

```
if %var%==antes (set var=despues&if !var!==despues echo “Cambiado a despues”)
```

Nótese el cambio de “%” por “!” en el segundo IF.

En este caso podemos comprobar que la ejecución es lo que deseábamos.

Volveremos con estos temas más adelante.

## ENTENDIENDO LA ENTRADA/SALIDA BÁSICA DE COMANDOS

Cuando ejecutamos un comando en una consola puede ser de dos tipos: o bien un comando interno de la shell de comandos (cmd.exe) o bien un programa externo (.com, .exe, .bat, etc.).

Lo primero que verifica el intérprete de comandos es si es un comando interno (por ejemplo “copy”).

Si lo es, lo ejecuta directamente. Si no pertenece a su tabla interna de comandos, empieza su búsqueda:

\* ¿Dónde y cómo lo busca?

1) Si se ha tecleado extensión del comando, por ejemplo: “ping.exe” se buscará sólo con dicha extensión. Si no se ha tecleado extensión, es decir, simplemente es “ping”, lo primero que hace el sistema es localizar la variable de entorno %PATHEXT% para ver las posibles extensiones y su orden de búsqueda.

En un entorno NT (y XP lo es), si ejecutamos: echo %PATHEXT% veremos su contenido:

```
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH
```

Es decir, primero intentará localizar un ping.com, si lo encuentra lo ejecuta, si no lo encuentra buscará un ping.exe, si lo encuentra lo ejecuta, etc...

2) Hemos hablado de que intentará localizar los anteriores. Pero ¿dónde?

2.1) En primer lugar en el directorio en curso.

2.2) Si no lo localiza, buscará en la variable de entorno %PATH% e irá recorriendo carpeta por carpeta y dentro de cada una de ellas, si no se ha tecleado extensión, en el orden predeterminado de extensiones de la variable %PATHEXT%

2.3) Si al final de esta búsqueda lo encuentra, lo ejecutará. En otro caso nos enviará un

mensaje de error como por ejemplo:

```
C:\>micomando
```

“micomando” no se reconoce como un comando interno o externo, programa o archivo por lotes ejecutable.

\*\*\* Debemos fijarnos en que básicamente un comando es “algo” que puede solicitar datos de entrada, posteriormente hace “algo” y por último nos da datos de salida.

Por ejemplo:

```
copy c:\carpeta\archivo.dat d:\backup
```

El comando (interno) copy, recibe datos, en este caso de la propia línea de comandos (y en la consola). Esos datos son dos: “c:\carpeta\archivo.dat” y “d:\backup”. A continuación ejecuta la copia.

Y por último nos informa de cómo ha ido esa copia. Nos puede dar dos tipos de mensajes:

a) O bien nos informa de copia correcta: ” 1 archivos copiados.”

b) o puede informarnos de una situación de error, por ejemplo:

“El sistema no puede hallar el archivo especificado.”, o bien:

“Acceso denegado.

0 archivos copiados.”

En general los programas o comandos tienen una entrada de datos estándar (STDIN), y una salida de

datos cuando ocurre un funcionamiento correcto (STDOUT) y cuando ocurre algún tipo de error (STDERR).

Fijémonos que las tres, en los casos de comando de consola, en general, están redirigidas a la propia consola. Tanto la entrada de datos como los mensajes, sean o no de error, “entran” y “salen” por pantalla. Pero... el comando o programa, internamente los redirige a salidas o entradas que pueden físicamente estar redirigidas, o ser redirigidas desde dispositivos que no sean la consola.

Técnicamente, esto se hace a nivel del “handle” (manejador) del fichero. Algunas veces en la literatura hispana esto se ha mal traducido por controlador. Básicamente un “handle” es un simple número que referencia internamente en un programa a un fichero. Fijémonos que realmente la entrada de datos es “como” si estamos leyendo de un fichero y la salida es como si se “escribe” en un

fichero, aunque en el ejemplo que acabamos de ver, estos ficheros son redirigidos a la propia consola.

Existen tres números, o handles, que históricamente en informática han representado siempre lo mismo: el 0 (STDIN), el 1 (STDOUT) y el 2 (STDERR). El resto son libres, a voluntad del programador.

Por tanto, y de cara interna a un programa, los datos que recibe de entrada se leen por STDIN, los mensajes de salida se escriben en STDOUT, y los mensajes de error en STDERR y además se activa un código de error (aunque algunas veces es costoso de distinguir, ya que sólo está en la cabeza del programador que lo haya realizado, qué es lo que él considera mensaje informativo y lo que considera mensaje de error... pero esto es otra cuestión).

Debido a la estandarización del STDIN, STDOUT y STDERR (“handles” 0, 1 y 2) lo que normalmente hace un sistema operativo es redirigir estos a la consola. Este es el comportamiento estándar del sistema, pero nosotros podemos igualmente hacer que sean redirigidos a donde queramos, tal y como veremos posteriormente.

Por ejemplo, en el caso del ejemplo anterior del comando “copy”, podemos hacer:

```
copy c:\carpeta\archivo.dat d:\backup > c:\log.txt
```

El símbolo “>” está indicando que todo lo que vaya a salir por el STDOUT se escribirá, en vez de en

la consola, en un archivo de texto c:\log.txt. Repito, porque es importante el matiz: STDOUT a fichero de texto.

Es decir, si el comando copy no puede copiar, recordemos que el mensaje de error lo saca en STDERR, por tanto en el fichero de texto no se escribiría nada, ya que en el ejemplo anterior, con el

símbolo >, sólo se ha redirigido el STDOUT. El STDERR queda con los valores predeterminados por el sistema operativo: es decir, se escribe el mensaje de error en consola.

\*\*\* Hemos comentado que a voluntad del programador de un comando o programa, este puede terminar correctamente o bien establecer un código de error al finalizar (escriba o no además un mensaje explicativo). El sistema operativo es el encargado de capturar esos códigos de error y los deja a nuestra disposición en una variable de entorno llamada %ERRORLEVEL%

Podemos, por ejemplo, verificar cual es el comportamiento del comando “copy”. Para ello vamos a realizar el siguiente ejemplo creándonos una carpeta de trabajo llamada c:\tmp

```
c:
```

```
cd \
```

```
md tmp
```

```
echo “texto de un fichero” > c:\tmp\fichero.txt
```

Con esto nos hemos posicionado en c: (por si no lo estuviésemos), hemos ido a su carpeta raíz (cd \),

y hemos creado una carpeta llamada tmp en donde estuviésemos posicionados en ese momento (md tmp) -md es abreviatura de “makedir”: crear directorio-. La última línea (echo) nos mostraría por pantalla “texto de un fichero”, es decir nos lo mostraría en el STDOUT. Pero con el símbolo “>” lo que hacemos es redirigir el STDOUT (la consola en este caso) a un fichero que en nuestro caso es: c:\tmp\fichero.txt. Si lo abrimos con el cuaderno de notas, veremos realmente el texto anterior.

Vamos a copiarlo en otro. Ejecutamos por ejemplo:

```
copy c:\tmp\fichero.txt c:\tmp\copia.txt
```

Si a continuación de este comando ejecutamos:

```
echo %ERRORLEVEL%
```

veremos que el comando anterior nos informa que ha terminado con 0 (se considera normalmente 0 una finalización correcta).

Vamos a provocar que el copy termine con error para ver cual es su comportamiento en este caso.

Un

método muy sencillo sería proteger contra escritura el fichero de salida y luego intentar escribir en él.

Es decir:

```
attrib c:\tmp\copia.txt +r (+r es “read only” -sólo lectura-)
```

```
copy c:\tmp\fichero.txt c:\tmp\copia.txt
```

Vemos que ahora en pantalla nos informa que “Acceso denegado”. Si a continuación ejecutamos:

```
echo %ERRORLEVEL%
```

veremos que el código de retorno es 1.

NOTA: en general los mensajes de error deben buscarse en la documentación del programa o comando ya que son a voluntad del programador. El sistema operativo lo único que hace es situarnos

el código de retorno en %ERRORLEVEL% y nosotros podremos tomar acciones en un script de comandos analizando dicha variable de entorno cuando nos interese controlar los códigos de terminación.

Bien, lo anterior no ha sido más que un ejemplo para intentar introducir los “redirectores” del sistema operativo.

## OPERADORES DE REDIRECCIÓN

> Escribe la salida del comando (normalmente STDOUT) en un fichero o un dispositivo (puede por tanto redirigirse a otro dispositivo, no sólo a archivo), en lugar de en la ventana del Símbolo del sistema.

< Lee la entrada (STDIN) del comando desde un archivo, en lugar de leerla desde la consola.

>> Añade la salida del comando al final de un archivo sin eliminar la información que ya está en él.

>& Escribe la salida de un controlador en la entrada de otro controlador (lo vemos posteriormente en detalle)

<& Lee la entrada desde un controlador y la escribe en la salida de otro controlador.

| Lee la salida de un comando y la escribe en la entrada de otro comando. Mal traducido por canalización o filtro ("pipe")

Hemos visto que con >& se escribe la salida de un controlador ("handle") en otro. Vayamos a un ejemplo práctico. En los ejemplos anteriores hemos visto que la salida de un comando como el copy podemos escribirlo en un fichero con el símbolo >. Pero también hemos comentado que sólo dirige

en STDOUT, por tanto, si el copy enviase un mensaje de error (STDERR) esto no se vería en el fichero. Esto es un problema si lo que queremos es crear un log de ejecución de un script para poder ver posteriormente la ejecución: los mensajes de error precisamente los habríamos perdido.

Pero... recordemos que > escribe el STDOUT. Si fuésemos capaces de "redirigir" el STDERR al STDOUT, entonces el > nos escribiría todos los mensajes, de funcionamiento correcto o erróneo en el fichero.

Es decir, si hacemos:

```
copy c:\tmp\fichero.txt c:\tmp\copia.txt >c:\log.txt
```

sólo los mensajes de funcionamiento correcto quedarían en c:\log.txt. Pero en cambio si redirigimos la salida de errores: STDERR ("handle" numero 2) al STDOUT ("handle" numero 1) y luego escribimos, en ese caso todo quedaría reflejado en el archivo log. Por tanto:

```
copy c:\tmp\fichero.txt c:\tmp\copia.txt 2>&1 >c:\log.txt
```

\*\*\* Operador de canalización ("|") o "pipe". Este es de los más interesantes ya que permite pasar "toda" una salida de datos a un programa que espera "toda" una entrada de datos en flujo. No todos los programas están preparados para admitir este tipo de redirección y debe leerse la documentación de cada uno de ellos al respecto.

Tenemos tres programas del sistema operativo "more" (más), "find" (buscar) y "sort" (ordenar) que admiten pipes y que nos pueden servir de ejemplo.

Veamos. Un dir /s nos muestra todos los archivos desde donde estemos posicionados incluyendo todos los archivos en subcarpetas. Si ejecutamos:

```
dir c:\s
```

veremos que empiezan a pasar pantallas de datos que no somos capaces de leer. Podemos redirigir la

salida de este comando al "more". Realmente el "more" no es nada más que un programa que admite

como entrada toda la salida de otro comando y que lleva un contador de líneas para irnos mostrando en pantalla 24 líneas y luego esperar a que pulsemos una tecla.

Ejecutemos por ejemplo:

```
dir c:\s | more
```

(cualquiera de los comandos anteriores puede abortarse con CTRL-C).

Un ejemplo de canalización y redirección: imaginemos que queremos en nuestro disco todos los archivos con extensión .log y dejar la información en un fichero de salida.

```
dir c:\b\s | find /I ".log" > c:\ficheros_log.txt
```

Los parámetros que he puesto, tanto en el comando dir, como en el comando find, siempre podemos verlos ejecutando dir /? y find /? para buscar los que mejor se adapten a lo que queremos.

Otro ejemplo: ver en orden "inverso" los ficheros de un directorio:

```
dir /b | sort /REVERSE
```

Los filtros ("pipes") no sólo admiten canalización. También pueden usarse, como cualquier programa, con comandos de redirección para leer entradas y escribir salidas en archivos. Por ejemplo, imaginemos que tenemos un fichero con nombres y queremos ordenarlo alfabéticamente.

Simplemente:

```
sort < lista.txt > listaalf.txt
```

Esto indicaría que la entrada al comando sort viene redirigida desde el fichero lista.txt y que la salida,

en vez de dejarla en pantalla, nos la escriba en listaalf.txt.

\*\*\* Los símbolos de redirección y canalización son potentísimos y a veces, incluso a un programador avezado le pueden causar más de un problema o costarle su correcta interpretación. En ejemplos que iremos viendo a lo largo de estos capítulos quedará más claro el concepto.

#### OPERADORES CONDICIONALES

Son los operadores `&&` y el operador `||`

Lo que exista después del operador `&&` se ejecutará “sólo” si la instrucción previa a él -en la misma línea- ha terminado correctamente (código 0).

Por contra, si usamos `||` sólo se ejecutará si la terminación de la instrucción previa termina incorrectamente (código distinto de cero)

Por ejemplo:

```
md kkk && echo “finalizado correcta la creación”
```

Sólo veremos el mensaje de aviso si la creación de la carpeta kkk ha sido posible.

#### CONFIGURACIÓN CÓMODA DEL INTÉRPRETE DE COMANDOS

Aunque esto no tiene nada que ver con los scripts, podemos fijarnos en que al abrir un intérprete de comandos (`cmd.exe`) no se pueden realizar funciones de copiar y pegar, lo cual puede sernos incómodo en Windows.

La manera de configurar el intérprete de comandos es pinchando con el ratón en la esquina superior izquierda del intérprete de comandos, propiedades, marcamos el casillero de “modalidad de edición rápida”.

De esta manera, con el botón izquierdo del ratón ya podremos marcar textos. Una vez marcado, pinchamos botón derecho del ratón y esto nos lo envía al portapapeles (es decir, esta es la acción “copiar” a la que estamos acostumbrados).

La acción “pegar”, es simplemente botón derecho (sin haber marcado nada previamente). Lo que tengamos en el portapapeles (lo hayamos llevado mediante la acción anterior, o bien lo hayamos copiado desde cualquier otra ventana Windows) lo pegará en donde esté el cursor.

#### CONCEPTOS EN UN SCRIPT – CONTROL DE FLUJOS

Acabamos de ver unos conceptos sobre entorno, comandos, variables, etc., que son básicos para poder realizar un script, pero hasta el momento no hemos visto ninguna instrucción excepto las puestas en los ejemplos previos, ni tampoco hemos visto cómo el shell ejecuta el script y cómo se puede tener control sobre la ejecución.

Básicamente un script de comandos, tal y como hemos comentado, no es nada más que escribir en un

archivo de texto con extensión `.bat` o `.cmd` todas las instrucciones que iríamos haciendo manualmente. Además, dentro del propio script, en función de resultados que vayamos obteniendo se pueden tomar acciones.

Recordemos algunas de las instrucciones vistas hasta el momento:

Instrucción: `ECHO`, su objetivo es mostrar por el `STDOUT` (normalmente la consola) el literal que exista a continuación. Recordemos que lo primero que hace el intérprete de comandos es “expandir” las instrucciones, es decir, si existe una variable de entorno (delimitada por símbolos `%`), lo primero que hace es sustituir dicha variable por su contenido. Es decir, si escribimos en un script:

```
echo esto es %var% prueba
```

y la variable de entorno “var” contiene el literal “una”, el intérprete de comandos:

1) Realiza la sustitución de las variables de entorno, por tanto la línea quedaría como:

```
ECHO esto es una prueba
```

2) A continuación, ejecuta la instrucción:

Como `ECHO` es un comando interno que escribe en el `STDOUT` lo que hay a continuación, nos mostrará el literal “esto es una prueba”.

Es importante la matización anterior ya que la expansión de variables es el primer paso y lo es para todas las instrucciones. Si hubiésemos cometido un error y hubiésemos puesto:

```
ECHA esto es %var% prueba
```

1) Expandiría la instrucción como:

ECHA esta es una prueba

2) Intentaría ejecutar la instrucción. “ECHA” no es el nombre de un comando interno, por tanto buscaría en la carpeta en curso y posteriormente en el PATH un programa (otro script, o un .com, o un .exe, etc... de acuerdo a las reglas vistas anteriormente). Si lo encuentra ejecutará el programa o script “ECHA” y si no lo encuentra nos dirá que sucede un error.

NOTA: La sintaxis completa de ECHO es:

ECHO literal a escribir

ECHO. (un “.” pegado a la instrucción sin separación: escribe una línea en blanco)

ECHO ON

ECHO OFF

Estas dos últimas son instrucciones a ejecutar sólo dentro de un script. Veamos su uso. Nos creamos un script llamado por ejemplo: pr.cmd en mi usuario y en una nueva carpeta de pruebas llamada “tmp” con el contenido:

```
set var=Hola
```

```
echo %var% mundo.
```

```
set var=
```

Lo guardamos en disco y ahora lo ejecutamos. ¿Qué vemos?:

```
C:\Documents and Settings\miusuario\tmp>set var=Hola
```

```
C:\Documents and Settings\miusuario\tmp>echo Hola mundo
```

```
Hola mundo
```

```
C:\Documents and Settings\miusuario\tmp>set var=
```

Es decir, nos ha ido mostrando todas las líneas que va ejecutando y mezclado con las salidas que a su

vez va dando el script.

Si queremos suprimir la salida de las líneas y dejar solamente la salida de los comandos, es decir “Hola mundo”, lo primero que debemos decirle al script es que no queremos “ECO” de lo que va traduciendo. Este “ECO” es útil cuando estamos depurando y probando un script, pero, cuando lo tenemos finalizado y correcto, lo lógico es ver sólo la salida y no línea por línea. En este caso podemos poner como primera línea del script:

```
@ECHO OFF
```

El símbolo @ como primer carácter en una línea del script indica que no queremos que esa línea se “vea” en la ejecución. Si además le damos la orden ECHO OFF, le estamos diciendo que hasta el final del script, o bien hasta que se encuentre otra orden ECHO ON, no muestre nada de la ejecución

(sí de la salida de instrucciones) por pantalla.

Es decir, en nuestro caso, si no queremos ver nada más que el literal “Hola mundo”, nos quedan dos posibilidades:

1) Poner una @ al comienzo de cada instrucción en el script. O bien:

2) Poner como primera línea del script @ECHO OFF

Vamos a hacer un paréntesis en este capítulo e introducir el concepto de nombre lógico de dispositivo.

## NOMBRES LÓGICOS DE DISPOSITIVOS

Hay ciertos dispositivos que podemos usar en cualquier comando o dentro de un script con nombres reservados que se refieren a dispositivos, normalmente lógicos, de nuestra máquina. Por ejemplo:

CON se refiere a la consola. Lo que se envíe o reciba desde la consola se puede referenciar con CON.

LPT1 se refiere a la impresora. Lo que se envíe a él se escribirá en la impresora (si está asignada a ese puerto).

NUL dispositivo nulo. Lo que se envíe a él, se pierde.

Aunque existen más dispositivos lógicos, llegado este punto sólo vamos a ver estos tres. Veamos unos ejemplos:

1) Imaginemos que hemos recibido una clave de registro de un producto por mail, y lo queremos

guardar en un fichero llamado licencia.txt en una carpeta determinada.

La manera clásica de hacerlo mediante Windows es abrir el cuaderno de notas, copiar la clave desde el mail recibido y guardar el fichero en la carpeta que queremos con el nombre licencia.txt. Bien, podríamos hacer en una consola de comandos:

```
copy con licencia.txt
```

En este punto se quedará el cursor esperando una entrada. Tecleamos la clave recibida (o la copiamos

desde el mail), y para finalizar la entrada de datos le damos a CTRL-Z (CTRL-Z es la marca estándar

de fin de fichero). Es decir, hemos copiado desde la consola a un fichero de la misma forma que si hubiésemos copiado un fichero a otro fichero.

2) Queremos enviar un fichero a impresora:

```
copy fichero.txt lpt1
```

3) No queremos ver, por ejemplo después de un comando copy la línea de “xx archivos copiados” que siempre nos informa el copy. Simplemente redirigimos la salida al dispositivo nulo.

```
copy fichero1.txt fichero2.txt >nul
```

### RETOMANDO EL CONTROL DE FLUJOS DE EJECUCIÓN

Una vez hecho el paréntesis anterior, vamos a ver las instrucciones de control flujo. Básicamente son

3.

1) Instrucción GOTO (ir a)

Permite en un punto determinado de nuestro script “saltar” a otro punto del script. Normalmente se usa después de una pregunta lógica por algún resultado (por ejemplo, dentro de una instrucción IF).

```
GOTO nombre_etiqueta
```

Y en algún lugar de nuestro script debemos tener

```
:nombre_etiqueta (nótense los “:”)
```

En este caso, al ejecutar la instrucción GOTO, buscará en el script la línea :nombre\_etiqueta y bifurcará a dicha línea.

NOTA: Existe una etiqueta implícita en cualquier script que no es necesario definir, :EOF (End Of File – Fin de fichero). Por tanto, la ejecución de la instrucción GOTO :EOF implica que se saltará hasta el final del fichero y por tanto finalizará el script.

2) Instrucción IF (si). Recordemos que lo encerrado entre corchetes es opcional

```
if [not] errorlevel número comando [else expresión]
```

Estamos verificando si la variable %ERRORLEVEL%, que indica el código de retorno de un comando anterior, tiene un valor determinado. Si lo tiene ejecutará el “comando” y si no lo tuviese y

hemos codificado en la línea (es opcional) un ELSE (en otro caso), ejecutará precisamente ese ELSE.

```
if [not] cadena1==cadena2 comando [else expresión]
```

cadena1 y cadena2 pueden ser cualquier cosa incluidas variables de entorno.

Recordemos que lo primero que se hace es la expansión de variables. Hay que tener muchísimo cuidado con esto y no me gustan las instrucciones del estilo anterior a menos que tengamos delimitadores específicos. Voy a intentar explicarlo. Imaginemos que tenemos una variable “var1”

con contenido “1” y otra “var2” con contenido también “1”. La instrucción:

```
if %var1%==%var2% echo “valen lo mismo”
```

Se ejecutará sin problemas y nos sacará el mensaje “valen lo mismo”.

Si “var2” contiene un “2”, la instrucción anterior, al no ser iguales, no mostrará nada.

Pero... ¿y si “var1” contiene “Hola mundo”, que sucedería? Pues un desastre. Al expandir la expresión quedaría:

```
if Hola mundo==2 echo “valen lo mismo”
```

Detrás del hola no hay símbolo de comparación, por lo cual la instrucción es sintácticamente incorrecta. Lo mismo pasaría si “var1” o “var2” no tienen contenido.

En estos casos, el truco consiste en encerrarlas entre algún delimitador. Por ejemplo, comillas -ya que estas definen un literal. Es decir:

```
if "%var1%"=="%var2%" echo "valen lo mismo"
```

Esto al expandirse quedaría:

```
if "hola mundo"=="2" echo "valen lo mismo"
```

La comparación se realiza entre literales entrecomillados lo cual no provoca error ni aun en el caso de que alguno sea nulo al no tener contenido.

```
if [not] exist nombreDeArchivo comando [else expresión]
```

```
if [/i] cadena1 operadorDeComparación cadena2 comando [else expresión]
```

El /i realiza la comparación de tal manera que no se distingue entre mayúsculas y minúsculas. Es decir "HOLA" es lo mismo que "hOLA".

Fijémonos que hemos puesto no un == (igualdad) en este caos, sino que por primera vez se ha puesto "operadorDeComparación". Este operador que veremos a continuación, y el símbolo == no es

nada más que uno de ellos.

```
if defined variable comando [else expresión]
```

## OPERADORES DE COMPARACIÓN

Especifica un operador de comparación de tres letras. En la siguiente tabla puede ver los valores de operadorDeComparación.

EQU igual a (es lo mismo que ==)

NEQ no es igual a

LSS menor que

LEQ menor que o igual a

GTR mayor que

GEQ mayor que o igual a

## PARÁMETROS

Cualquier script es capaz de recibir parámetros desde la línea invocante. Se considera parámetro cualquier literal enviado en la línea que invoca al script.

El sistema considera que los parámetros están separados por espacios en blanco o bien por el delimitador " ; ".

Cada parámetro se referencia dentro del script por %1, %2, %3, etc...

Vamos a crear un script que reciba dos parámetros, el nombre de un fichero y el nombre de una carpeta. El objetivo del script es copiar el archivo desde la carpeta en donde estemos posicionados en

la carpeta destino, siempre y cuando no exista ya en el destino.

Una manera grosera de realizar el script, llamémosle copia.cmd, sin verificar que los parámetros son

correctos, sería:

```
@echo off
```

```
if not exist %2\%1 copy %2 %1
```

Si invocásemos al script de la siguiente manera:

```
copia.cmd mifichero.txt c:\backup
```

al desarrollar la instrucción el script, realizaría:

```
if not exist c:\backup\mifichero.txt copy mifichero.txt c:\backup
```

lo cual es precisamente lo que queríamos hacer. He comentado que este script, tal y como está, está realizado de una manera grosera. Si el nombre del archivo o el nombre de la carpeta destino tuviese espacios en blanco el sistema consideraría que dicho espacio en blanco es un delimitador de parámetros y por tanto nos llegarían al script más parámetros de los deseados. Realmente para realizar correctamente el script deberíamos hacer:

```
@echo off
```

```
if {%1}=={} goto error
```

```
if {%2}=={} goto error
```

```

if {%3} NEQ {} goto error
if not exist %1 goto error1
if not exist %2\nul goto error2
if exist %2\%1 goto aviso
copy %1 %2\%1 2>&1 >nul
if not errorlevel 1 goto correcto
echo Copia errónea. Verifique permisos de la carpeta destino.
goto:EOF
:error
echo parámetros erróneos. Deben recibirse dos parámetros.
goto :EOF
:error1
echo Nombre del fichero origen no existe.
goto :EOF
:error2
echo Carpeta destino no existe.
goto :EOF
:aviso:
echo El archivo ya existe en el destino. No se ha realizado copia.
goto :EOF
:correcto
echo Archivo copiado correctamente.
goto :EOF

```

Y la manera de invocar al comando sería:

```
copia.cmd "nombre fichero.xxx" "c:\carpeta destino"
```

Nótese que hemos entrecomillado cada uno de los parámetros. De esta manera sólo se recibirán dos parámetros ya que si no, se recibirían 4 al usar como delimitador el sistema los espacios en blanco entre ellos.

Comentemos un poco el script.

- \* Verificamos que hemos recibido 2 parámetros y sólo 2.

- \* Verificamos que exista el fichero origen

- \* Verificamos que exista la carpeta destino. Para verificarlo usamos un pequeño truco: preguntar por

el dispositivo "nul" en la carpeta destino. Si no existe, es que la carpeta no existe.

- \* Verificamos que de acuerdo a las especificaciones del script no exista ya el fichero en la carpeta destino.

- \* Realizamos la copia. Redirigimos todo a nul para que no de mensajes de archivos copiados.

- \* Comprobamos el código de retorno y si no es cero enviamos el mensaje correspondiente.

- \* Damos un mensaje de copia correcta cuando se ha finalizado correctamente el script.

Bien, el ejemplo anterior nos puede servir para la introducción de lo que son los parámetros y cómo se deben usar o referenciar en el script. Vamos a ver toda la casuística más en detalle.

Realmente podemos referenciar los parámetros desde %0 a %9. El parámetro recibido en %0 no se teclea y siempre el sistema nos pasa como contenido el nombre del script invocado. Hagamos una prueba:

```
@echo off
```

```
echo Parámetro 0: %0
```

%1 a %9 son los posibles parámetros enviados en la línea de comandos. Realmente pueden pasarse más de 9 parámetros pero sólo 9 de ellos serán accesibles directamente. Si pasásemos más, hay que usar el comando shift, el cual tiene por objeto desplazar los parámetros. Es decir, al ejecutar shift lo que se hace es que el %2 pasa a ser %1 (el %1 se pierde por lo que debemos guardarlo), el %3 pasa a

ser el %2, etc... De tal manera que al final el %9 pasa a contener el posible décimo parámetro

pasado.

Realicemos como prueba:

```
@echo off
```

```
echo Antes de desplazar
```

```
echo Parametro 1: %1
```

```
echo Parametro 2: %2
```

```
echo Parametro 3: %3
```

```
echo Parametro 4: %4
```

```
echo Parametro 5: %5
```

```
echo Parametro 6: %6
```

```
echo Parametro 7: %7
```

```
echo Parametro 8: %8
```

```
echo Parametro 9: %9
```

```
shift
```

```
echo Despues de desplazar
```

```
echo Parametro 1: %1
```

```
echo Parametro 2: %2
```

```
echo Parametro 3: %3
```

```
echo Parametro 4: %4
```

```
echo Parametro 5: %5
```

```
echo Parametro 6: %6
```

```
echo Parametro 7: %7
```

```
echo Parametro 8: %8
```

```
echo Parametro 9: %9
```

```
shift
```

```
echo Despues de desplazar de nuevo
```

```
echo Parametro 1: %1
```

```
echo Parametro 2: %2
```

```
echo Parametro 3: %3
```

```
echo Parametro 4: %4
```

```
echo Parametro 5: %5
```

```
echo Parametro 6: %6
```

```
echo Parametro 7: %7
```

```
echo Parametro 8: %8
```

```
echo Parametro 9: %9
```

e invoquemos al script anterior (parm.cmd, por ejemplo) de la siguiente manera:

```
parm.cmd 1 2 3 4 5 6 7 8 9 10 11
```

NOTA: Podemos referenciar “toda” la línea de parámetros mediante %\*

Si reescribimos el bat anterior:

```
@echo off
```

```
echo Toda la línea de parámetros es: %*
```

y ejecutamos

```
parm.cmd 1 2 3 4 5 6 7 8 9 10
```

podemos comprobar la salida. Esto puede ser útil si luego queremos tratar todo como una sola línea y

usamos comandos específicos, que veremos más adelante, para tratar dicha línea.

A los parámetros y sólo a los parámetros, pueden anteponérseles modificadores los cuales realizan una conversión específica. Vamos a ver un ejemplo y a continuación daremos todas las posibilidades.

Hagamos un script (parm.cmd):

```
@echo off
```

```
echo Parámetro 1 expandido: %~f1
```

echo Parámetro 2 expandido: %~f2

Supongamos que nuestro script esté en la carpeta:

C:\Documents and Settings\miusuario\tmp>

Al ejecutar:

parm.cmd aaa.www bbb

nos devolverá:

Parámetro 1 expandido: C:\Documents and Settings\miusuario\tmp\aaa.www

Parámetro 2 expandido: C:\Documents and Settings\miusuario\tmp\bbb

Es decir, el delimitador %~f por delante del número de parámetro nos expande el parámetro considerándolo como si fuese nombre de archivo (sin verificarlo) al nombre completo de unidad, ruta

y archivo en donde se esté ejecutando el script.

Realmente todos los modificadores posibles de parámetros son los que se describen a continuación:

%~1 Expande %1 y quita todas las comillas ("").

%~f1 Expande %1 y lo convierte en un nombre de ruta de acceso completo.

%~d1 Expande %1 a una letra de unidad.

%~p1 Expande %1 a una ruta de acceso.

%~n1 Expande %1 a un nombre de archivo.

%~x1 Expande %1 a una extensión de archivo.

%~s1 Ruta de acceso expandida que únicamente contiene nombres cortos.

%~a1 Expande %1 a atributos de archivo.

%~t1 Expande %1 a una fecha/hora de archivo.

%~z1 Expande %1 a un tamaño de archivo.

%~\$PATH:1 Busca los directorios enumerados en la variable de entorno PATH y expande %1 al nombre completo del primer directorio encontrado. Si el nombre de la variable de entorno no está definido o la búsqueda no encuentra el archivo, este modificador se expande a la cadena vacía.

La tabla siguiente enumera las combinaciones posibles de modificadores y calificadores que puede usar para obtener resultados compuestos.

%~dp1 Expande %1 a una letra de unidad y una ruta de acceso.

%~nx1 Expande %1 a un nombre y extensión de archivo.

%~dp\$PATH:1 Busca los directorios enumerados en la variable de entorno PATH para %1 y expande a la letra de unidad y ruta de acceso del primer directorio encontrado.

%~ftza1 Expande %1 a una línea de salida similar a dir.

NOTA: En los ejemplos anteriores, se puede reemplazar %1 y PATH con otros valores de parámetros de proceso por lotes.

No se pueden manipular parámetros de proceso en los scripts de la misma manera en que se manipulan variables de entorno (funciones de manejo de cadenas vistas anteriormente). No se pueden buscar y reemplazar valores ni examinar subcadenas. Sin embargo, se puede asignar el parámetro a una variable de entorno para, después, manipular la variable de entorno.

Aunque no es necesario memorizarlos sí que es importante saber que existen, ya que algunas veces nos puede ser necesario usarlos para expansión sobre todo en nombres de archivos.

#### INVOCACIÓN A PROCEDIMIENTOS. INSTRUCCIÓN CALL

Muchas veces es necesario desde un script llamar a otro procedimiento, o llamar a una subrutina del mismo procedimiento. Vemos ambos casos.

\* Llamada a procedimiento externo. Vamos a crear dos script. El primero de ellos lo llamaremos principal.cmd y con el contenido:

```
@echo off
```

```
echo estoy en el principal
```

```
call invocado.cmd
```

```
echo de vuelta en el principal
```

Y un segundo procedimiento llamado invocado.cmd:

```
@echo off
```

echo estoy en el invocado

Si lo ejecutamos veremos los resultados. Recordemos que tanto el procedimiento principal como el invocado pueden recibir parámetros y podemos jugar con ellos como queramos. Por ejemplo:

principal.cmd:

```
@echo off
```

echo estoy en el principal

echo mi primer parámetro: %1

echo mi segundo parámetro: %2

call invocado.cmd %2

echo de vuelta en el principal

invocado.cmd:

```
@echo off
```

echo estoy en el invocado

echo el parámetro del invocado es: %1

y ahora lo ejecutamos como

principal.cmd primero segundo

¿Qué es lo que vemos?...

Nótese, que el ejemplo anterior, si en vez de llamar con CALL, hubiésemos omitido dicho CALL, es

decir la línea:

```
CALL invocado.cmd
```

sólo tuviese:

```
invocado.cmd
```

también se ejecutaría el invocado. Pero cuando este terminase, terminaría todo y no volvería al script

principal.

\* Llamada a procedimiento interno (subrutina). La sintaxis es “CALL :nombre”, en donde :nombre es un nombre de etiqueta interno al cual saltará la ejecución del script, pero a diferencia del goto, cuando finalice la propia rutina continuará la ejecución después del CALL invocante. Se considera la

finalización de la subrutina cuando se alcanza el fin del archivo por lotes.

Pongamos un ejemplo (script.cmd)

```
@echo off
```

echo Parámetro 1: %1

echo Parámetro 2: %2

call :rutina1 %2

echo estoy de nuevo en el principal

call :rutina1 %1

echo estoy de nuevo en el principal

call :rutina2 parametrosjuntos%1%2

goto final

```
:rutina1
```

echo en la rutina recibo parámetro: %1

goto:EOF

```
:final
```

echo estoy ya al final

Fijémonos que la subrutina la terminamos con goto :EOF. Esto indica que cuando se la invoca con CALL volverá a la línea de continuación del invocante (realmente salta a fin de fichero, con lo que provoca la condición de vuelta que hemos citado anteriormente).

## CONSIDERACIONES SOBRE ÁMBITO DE VARIABLES

Hemos comentado anteriormente que cada programa tiene su área de entorno la cual es una copia del

área de entorno del programa invocante, y que por tanto, si se modifican o añaden variables de entorno, esto \*sólo\* afecta a dicho programa (y por supuesto a los que él invoque, ya que heredarán su entorno).

Pero ¿cuál es el entorno de un script? Un script no es en sí un programa: es una secuencia de comandos dentro del shell. Dentro del cmd.exe. Pero además dentro “sólo” del cmd.exe que lo ha invocado. Un script no tiene entorno propio: se usa, y por tanto se modifica o se añaden variables al entorno del cmd.exe invocante (y sólo a él, es decir si tenemos arrancado o arrancamos posteriormente otro cmd.exe, este último no se verá afectado).

Si un script llama a otro script (mediante CALL) tendrá acceso a todas las variables modificadas en el script principal.

No es conveniente modificar el entorno del cmd.exe (aunque sea el invocante del script), y lo ideal sería que el script, al finalizar, dejase el entorno tal y como estaba antes de arrancarse.

Pongamos un ejemplo: un simple script que haga:

```
@echo off
set a=%1
set computername=%a%_mio
....
```

Esto es válido y hemos usado una variable llamada “computername” simplemente porque se nos ha ocurrido en el script. Pero, curiosamente, ese nombre de variable ya existía en el sistema (en el entorno del cmd.exe). Por tanto, lo que sucede es que al finalizar el script, ese cmd.exe -y sólo esese quedará en su entorno con una variable que ya existía modificada por el script, y además con otra variable “a” que antes de empezar el script no tenía valor.

Lo ideal sería no usar nombres que ya existan en el entorno en el propio script, y además que el script

limpiase sus variables internas antes de finalizar. Esto es engorroso y normalmente difícil de controlar.

El sistema nos da la posibilidad de hacerlo él automáticamente mediante las instrucciones “setlocal” y “endlocal”. Por ejemplo, si en el script anterior hubiésemos hecho:

```
@echo off
setlocal
set a=%1
set computername=%a%_mio
.....
endlocal
```

Es decir, le estamos diciendo desde la instrucción “setlocal” que todo lo que hagamos será local al script. Y cuando se ejecuta “endlocal” (al final del script) le estamos diciendo que deshaga todas las variables usadas y permanezca tal y como estaba desde que se ejecutó la anterior instrucción “setlocal”.

Como norma es conveniente en los scripts (o al menos en el principal si usásemos llamadas CALL - esto dependerá de la lógica-) el usar un “setlocal” al inicio del script y un “endlocal” como última instrucción del script.

La instrucción “setlocal” admite además opcionalmente la siguiente sintaxis:

```
setlocal {enableextension | disableextensions} {enabledelayedexpansion | disabledelayedexpansion}
```

Algunas de ellas -las dos últimas- ya las hemos comentado: por defecto el intérprete de comandos tiene desactivada la expansión diferida (a menos que se haya arrancado con /V:ON o bien se hayan modificado ciertas claves de registro para cambiar el comportamiento general del intérprete de comandos), y por ello hay que prestar especial atención a la expansión de las variables de entorno en

la línea de comando antes de que esta pase a ejecución (recordemos la sintaxis con “!” como delimitador de la variable de entorno en vez de usar el delimitador “%” normal. Es conveniente revisar ese capítulo ya que el concepto es extremadamente importante).

Las dos primeras: {enableextension | disableextensions} no es conveniente tocarlas ya que se

desactivarían las extensiones de comandos y las instrucciones:

DEL o ERASE

COLOR

CD o CHDIR

MD o MKDIR

PROMPT

PUSHD

PUSHD

POPD

SET

SETLOCAL

ENDLOCAL

IF

FOR

CALL

SHIFT

GOTO

START (también incluye cambios en la invocación de comandos externos)

ASSOC

FTYPE

Tendría un comportamiento diferente. Podemos ver estas modificaciones pidiendo la ayuda de cualquiera de las instrucciones anteriores, por ejemplo `CALL /?`

Por defecto, tal y como viene Windows, el valor es:

```
setlocal enableextension disabledelayedexpansion
```

Debido a que es posible que el sistema en donde se vayan a ejecutar nuestros scripts pueda haberse modificado el comportamiento global, siempre es conveniente forzar en nuestro script principal lo que deseamos.

Mi consejo además es permitir la expansión diferida del entorno, por tanto, es conveniente siempre poner al comienzo del script principal la instrucción:

```
setlocal enableextension enabledelayedexpansion
```

con lo cual ya podremos usar el delimitador “!” cuando nos interese.

#### COMANDOS INTERNOS DEL INTÉRPRETE DE COMANDOS

Llegado a este punto ya podemos ver todos los comandos internos que tiene el intérprete de comandos. Recordemos igualmente que hay una lista alfabética y detallada en el fichero de ayuda `ntcmds.chm` que reside en `\windows\help`. Es conveniente para los que se vayan a dedicar a la escritura de scripts el tener un acceso directo en Windows a dicho archivo. En esa lista alfabética figuran tanto los comandos internos de Windows como los externos que nos da la propia versión del sistema operativo. Vamos a dar un repaso de todas formas y un pequeño comentario a cada uno de ellos, excepto para el comando más potente de Windows: el comando FOR, el cual merecerá todo un

capítulo aparte. Es conveniente llegado a este punto que se vaya consultando cada uno de los comandos que citaré a continuación en `ntcmds.chm`.

Los comandos internos son:

ASSOC Ver / cambiar asociación a la extensión de un archivo.

CALL Llamada a procedimiento.

CD o CHDIR Cambiar carpeta en curso.

CLS Borrar pantalla.

COLOR Atributos de color en el texto enviado a consola.

COPY Copiar archivos.

DEL o ERASE Borrar archivos.

DIR Listar contenido de la carpeta en curso.

ECHO Mostrar por pantalla (en STDOUT).

ENDLOCAL Fin de tratamiento de variables locales de entorno.  
EXIT Final de la secuencia de comandos por lotes o sale de un cmd.exe si se invoca directamente.  
FOR (lo veremos en todo un capítulo aparte).  
FTYPE Muestra o modifica los tipos de archivos empleados en asociaciones de extensiones de archivo.  
GOTO Ir a. (Salto incondicional a una etiqueta del script en curso).  
IF Instrucción de comparación.  
MD o MKDIR Crear una carpeta.  
PATH Como instrucción devuelve el contenido de la variable %PATH%.  
PAUSE Detiene la ejecución de un script y espera a que se pulse una tecla para continuar.  
POPD Cambia el directorio actual al que se haya almacenado con PUSHHD.  
PROMPT Cambia el símbolo de sistema de Cmd.exe.  
PUSHHD Guarda el nombre del directorio actual para que lo use el comando POPD.  
REM Es un comentario en un script. Lo que va a continuación no se ejecuta.  
REN o RENAME Renombra un archivo o una carpeta.  
RM o RMDIR Borrar una carpeta.  
SET Asignación de contenido a variables de entorno.  
SETLOCAL Tratamiento o comportamiento de las variables en un script y de las extensiones de comandos.  
SHIFT Desplazamiento de los parámetros con que se invoca a un script.  
START Inicia una ventana independiente de símbolo del sistema para ejecutar un programa o un comando especificado.  
TYPE Vuelca el contenido de un archivo al STDOUT.  
VER Muestra el número de versión de Windows.  
COMANDO 'FOR'

Es el comando más importante y más potente que podemos usar en un script. Por tanto, vamos a verlo en detalle y haciendo hincapié en ejemplos de su uso.

NOTA: Es el único comando que tiene sintaxis diferente ejecutado desde la línea de comandos y ejecutado desde un script. Los subíndices del comando -tal y como se verán a continuación- se expresan con un símbolo % desde la línea de comandos, pero en un script deben expresarse con un doble símbolo, es decir con %%.  
Definición: Ejecuta un comando especificado para cada archivo de un conjunto de archivos.

Sintaxis. Admite varias formas: (recordando que en script %variable debe ser %%variable)

1) for %variable in (grupo) do comando  
%variable

Requerido. Representa un parámetro reemplazable. Utilice %variable para ejecutar for en el símbolo

del sistema. Utilice %%variable para ejecutar el comando for dentro de un archivo por lotes. Las variables distinguen entre mayúsculas y minúsculas y se deben representar con un valor alfabético, como %A, %B o %C.

(grupo)

Requerido. Especifica uno o varios archivos, directorios, intervalo de valores o cadenas de texto que se desea procesar con el comando especificado. Los paréntesis son obligatorios.

El parámetro grupo puede representar un único grupo de archivos o varios. Puede utilizar caracteres comodín (es decir, \* y ?) para especificar un grupo de archivos. Los siguientes son grupos de archivos válidos:

(\* .doc)

(\* .doc \* .txt \* .me)

(ene\* .doc ene\* .rpt feb\* .doc feb\* .rpt)

(ar??1991.\* ap??1991.\*)

por ejemplo, desde la línea de comandos:

```
for %f in (c:\windows\*.*) do @echo %f
```

o bien desde un script:

```
for %%f in (c:\windows\*.*) do @echo %%f
```

nos mostrará los archivos que están dentro de la carpeta c:\windows (similar a un “dir” pero sólo nos

mostrará el nombre.

Ejercicio [for-1]: usando el comando for en la forma citada anteriormente obtener únicamente el nombre del archivo. Si lo ejecutamos vemos que la salida es c:\windows\fichero.ccc, c:\windows\fichero2.xxx, etc., y lo que queremos es únicamente el nombre y extensión del archivo. Evidentemente una manera simple de hacerlo sería:

c:

```
cd \windows
```

```
for %f int (*.*) do @echo %f
```

Pero esto no es lo que se solicita, sino que, con una sola línea for y sin estar posicionado en la carpeta

de Windows, se obtenga el mismo resultado que con las tres instrucciones anteriores.

(la respuesta a estos ejercicios al final del capítulo).

comando

Requerido. Especifica el comando que desea ejecutar en cada archivo, directorio, intervalo de valores

o cadena de texto incluido en el (grupo) especificado.

NOTA: Hay que prestar especial atención al posible uso del %variable dentro de la acción ‘do’ así como al posible uso de variables dentro de él. Recordemos la parte vista anteriormente de expansión diferida o no del comando a la hora de traducir la línea.

NOTA: Cuando están habilitadas las extensiones de comandos (es el valor por defecto del cmd.exe en Windows) -ver capítulos anteriores- es posible también las sintaxis que veremos a continuación

2) for /D %variable in (grupo) do comando

/D -> Sólo directorios

Si grupo contiene caracteres comodín (\* y ?), el comando especificado se ejecuta para cada directorio (en lugar de un grupo de archivos de un directorio especificado) que coincida con grupo.

La sintaxis es:

```
for /D %variable in (grupo) do comando [opcionesDeLíneaDeComandos]
```

Por ejemplo, para recorrer todas las subcarpetas de la carpeta Windows:

```
for /d %s in (c:\windows\*.*) do @echo %s
```

3) for /R [unidad :]rutaDeAcceso] %variable in (grupo) do comando

[opcionesDeLíneaDeComandos]

/R -> Recursividad. Recorre el árbol de directorios con raíz en [unidad:]rutaDeAcceso y ejecuta la instrucción for en cada directorio del árbol. Si no se especifica un directorio después de /R, se considera el directorio actual. Si grupo tiene únicamente un punto (.) sólo se enumerará el árbol de directorios.

Ejercicio [for-2]: Se desea obtener en un fichero llamado c:\datos.txt todos los archivos, un delimitador como puede ser “;” y el nombre de la carpeta en la cual residan. Es decir un archivo de texto con el formato:

```
notepad.exe;c:\Windows
```

... etc...

Ejercicio [for-3]: Obtener los mismos datos que en el ejercicio anterior pero recibiendo la carpeta origen por parámetro.

4) for /L %variable in (númeroInicial,númeroPaso,númeroFinal) do comando

/L -> Iteración de un intervalo de valores. Se utiliza una variable iterativa para establecer el valor inicial (númeroInicial) y, después, recorrer un intervalo de valores especificado hasta que el valor sobrepase el valor final (númeroFinal) especificado. /L ejecutará la iteración mediante la comparación de númeroInicial con númeroFinal. Si númeroInicial es menor que númeroFinal, el

comando se ejecutará. Si la variable iterativa sobrepasa el valor de númeroFinal, el shell de comandos sale del bucle. También se puede utilizar un valor númeroPaso negativo para recorrer un intervalo de valores decrecientes. Por ejemplo, (1,1,5) genera la secuencia 1 2 3 4 5 y (5,-1,1) genera

la secuencia (5 4 3 2 1)

```
for /l %i in (5,-1,1) do @echo %i
```

Ejercicio [for-4]: Dar un barrido de ping a las primeras 30 ip's desde 172.16.0.1

5) Interacción y análisis de archivos. Es la parte más potente del FOR ya que permite ejecución interna de comandos dentro del propio "do" -realiza o puede realizar "spawn" del posible comando dentro del do-

Lo veremos después de entender la solución a los ejercicios.

RESPUESTAS A LOS EJERCICIOS (1)

```
[for-1]
```

```
for %f in (c:\windows\*.*) do @echo %~nxf
```

Revisar el capítulo de "parámetros" para ver los prefijos ~n o bien ~x que pueden ponerse a parámetros (o bien a variables explícitas o implícitas en un for -son equivalentes a parámetros-) para

entender el comando anterior.

```
[for-2]
```

Vamos a realizarlo en un script.

```
@
```

Esta entrada fue escrita por [backupjavcasta](#), publicada en Agosto 15, 2009 a las 12:21 pm, archivada bajo [Scripting](#), [Sistemas](#), [Utilidades](#). Agrega el favorito al [enlace permanente](#). Sigue los comentarios aquí con el [feed para esta entrada](#). [Envía un comentario](#) o deja una ruta: [Trackback URL](#).

[« Script VBS: Windows, Cambiar nombre conexión de red](#)

[Barrapunto | Escalada de privilegios local en todos los kernels linux »](#)

## 2 comentarios

1.  Avedefuego

Publicado el Agosto 18, 2009 a las 5:35 pm | [Permalink](#) | [Responder](#)

Me sirvió mucho la información que tienes en tu sitio ya que me estoy iniciando en este lenguaje y me ayudó a comprender cosas que no había comprendido en otro lado además es como una guía rápida para cuando necesite buscar algún comando o si quiero saber para que sirve algo que no recuerdo; gracias

Por otro lado a ver si tu me puedes ayudar quiero o intento, tal vez ni se pueda hacer un script que me busque en archivos TXT una palabra o frase específica.

Lo que pasa es esto que ya hice uno que me almacena en txt los resultados, por decir algo mando un ping y me lo guarda en txt ahora lo que quiero es un script que me busque cuando en el resultado obtenga un "time out" por ejemplo

Cualquier ayuda o sugerencia es bienvenida y se agradecerá

2.  [javcasta](#)

Publicado el Agosto 18, 2009 a las 5:51 pm | [Permalink](#) | [Responder](#)

Tengo algo parecido en:

<http://javcasta.wordpress.com/2009/07/31/script-cmd-o-bat-centinela-o-control-de-red/>

...

```
set iphost=192.168.1.1
set ERRORLEVEL=
ping %iphost% -n 3 > testigo.txt
find /C "recibidos = 0" testigo.txt
Rem también valdria "perdidos =3"

goto alertant%ERRORLEVEL%

Rem Hay alerta errorlevel=0
:alertant0
echo %iphost% no responde a ICMP request
pause
```

Este extracto de script del link te muestra que lo que pides se consigue volcando el resultado del ping a un fichero y luego buscando con find en ese fichero.

Intenta implementarlo, sino lo consigues me pones el código que te falle y te oriento (no doy pescado, enseñó a pescar  ).