7

Seeing the World as the Shell Sees It
Ver el mundo como lo ve el termina

In this chapter, we are going to look at some of the “magic” that occurs on the command line when we press the enter key.
En este capítulo, veremos algo de la “magia” que ocurre en la línea de comando cuando presionamos la tecla enter.
While we will examine several interesting and complex features of the shell, we will do it with just one new command:
Si bien examinaremos varias características interesantes y complejas del shell, lo haremos con un solo comando nuevo:

- echo --Display a line of text
  echo -- muestra una línea de texto

Expansion
Expansión

Each time we type a command and press the enter key, bash performs several substitutions upon the text before it carries out our command.
Cada vez que escribimos un comando y presionamos la tecla enter, bash realiza varias sustituciones en el texto antes de ejecutar nuestro comando.

We have seen a couple of cases of how a simple character sequence, for example * , can have a lot of meaning to the shell. The process that makes this happen is called expansion. With expansion, we enter something, and it is expanded into something else before the shell acts upon it. To demonstrate what we mean by this, let’s take a look at the echo command.
Hemos visto un par de casos de cómo una secuencia de caracteres simple, por ejemplo *, puede tener mucho significado para el shell. El proceso que hace que esto suceda se llama expansión. Con la expansión, entramos en algo y se expande en otra cosa antes de que la cáscara actúe sobre él. Para demostrar lo que queremos decir con esto, echemos un vistazo al comando echo.

echo is a shell builtin that performs a very simple task. It prints its text arguments on standard output.
echo es un shell integrado que realiza una tarea muy sencilla. Imprime sus argumentos de texto en la salida estándar.

```
[me@linuxbox ~]$ echo this is a test
this is a test
```

That's pretty straightforward. Any argument passed to echo gets displayed. Let's try another example.
Eso es bastante sencillo. Se muestra cualquier argumento pasado a echo. Probemos con otro ejemplo.

```
[me@linuxbox ~]$ echo *
Desktop Documents ls-output.txt Music Pictures Public Templates Videos
```

So what just happened? Why didn't echo print * ? As we recall from our work with wildcards, the * character means match any characters in a filename, but what we didn't see in our original discussion was how the shell does that. The simple answer is that the shell expands the * into something else (in this instance, the names of the files in the current working directory) before the echo command is executed. When the enter key is pressed, the shell automatically expands any qualifying characters on the command line before the command is carried out, so the echo command never saw the * , only its expanded result. Knowing this, we can see that echo behaved as expected.
Entonces, ¿qué acaba de pasar? ¿Por qué no se hizo eco de la impresión *? Como recordamos de nuestro trabajo con comodines, el carácter * significa coincidir con cualquier carácter en un nombre de archivo, pero lo que no vimos en nuestra discusión original fue cómo lo hace el shell. La respuesta simple es que el shell expande * en algo más (en este caso, los nombres de los archivos en el directorio de trabajo actual) antes de que se ejecute el comando echo. Cuando se presiona la tecla enter, el shell expande automáticamente cualquier carácter calificado en la línea de comando antes de ejecutar el comando, por lo que el comando echo nunca vio el *, solo su resultado expandido. Sabiendo esto, podemos ver que el eco se comportó como se esperaba.

Pathname Expansion
Expansión de nombre de ruta

The mechanism by which wildcards work is called pathname expansion. If we try some of the techniques that we employed in earlier chapters, we will see that they are really expansions. Given a home directory that looks like this:
El mecanismo por el cual funcionan los comodines se llama expansión de nombre de ruta. Si probamos algunas de las técnicas que empleamos en capítulos anteriores, veremos que en realidad son expansiones. Dado un directorio de inicio que se ve así:

```
[me@linuxbox ~]$ ls
Desktop ls-output.txt  Pictures Templates
Documents Music Public Videos
```

we could carry out the following expansions:
podríamos realizar las siguientes ampliaciones:

```
[me@linuxbox ~]$ echo D*
Desktop Documents
```

and this:
y esto:

```
[me@linuxbox ~]$ echo *s
Documents Pictures Templates Videos
```

or even this:
o incluso esto

```
[me@linuxbox ~]$ echo [[:upper:]]*
Desktop Documents Music Pictures Public Templates Videos
```

and looking beyond our home directory, we could do this:
y mirando más allá de nuestro directorio de inicio, podríamos hacer esto:

```
[me@linuxbox ~]$ echo /usr/*/share
/usr/kerberos/share /usr/local/share
```

Pathname Expansion of Hidden Files
Expansión de nombre de ruta de archivos ocultos

As we know, filenames that begin with a period character are hidden. Pathname expansion also respects this behavior. An expansion such as the following does not reveal hidden files:
Como sabemos, los nombres de archivo que comienzan con un punto están ocultos. La expansión del nombre de ruta también respeta este comportamiento. Una expansión como la siguiente no revela archivos ocultos:

```
echo *
```

It might appear at first glance that we could include hidden files in an expansion by starting the pattern with a leading period, like this:
A primera vista, podría parecer que podríamos incluir archivos ocultos en una expansión iniciando el patrón con un punto inicial, como este:

```
echo .*
```

It almost works. However, if we examine the results closely, we will see that the names . and .. will also appear in the results. Because these names refer to the current working directory and its parent

> directory, using this pattern will likely produce an incorrect result. We can see this if we try the following command:
> Casi funciona. Sin embargo, si examinamos los resultados de cerca, veremos que los nombres. y .. también aparecerán en los resultados. Debido a que estos nombres se refieren al directorio de trabajo actual y su directorio principal, el uso de este patrón probablemente producirá un resultado incorrecto. Podemos ver esto si probamos el siguiente comando:

```
ls -d .* | less
```

> To better perform pathname expansion in this situation, we have to employ a more specific pattern.
> Para realizar mejor la expansión del nombre de ruta en esta situación, tenemos que emplear un patrón más específico.

```
echo .[!.]*
```

> This pattern expands into every filename that begins with only one period followed by any other characters. This will work correctly with most hidden files (though it still won't include filenames with multiple leading periods). The ls command with the -A option ("almost all") will provide a correct listing of hidden files.
> Este patrón se expande en todos los nombres de archivo que comienzan con un solo punto seguido de cualquier otro carácter. Esto funcionará correctamente con la mayoría de los archivos ocultos (aunque aún no incluirá nombres de archivo con varios puntos iniciales). El comando ls con la opción -A ("casi todos") proporcionará una lista correcta de archivos ocultos.

```
ls -A
```

Tilde Expansion
Expansión Tilde

As we may recall from our introduction to the cd command, the tilde character (~) has a special meaning. When used at the beginning of a word, it expands into the name of the home directory of the named user or, if no user is named, the home directory of the current user.
Como podemos recordar de nuestra introducción al comando cd, el carácter tilde (~) tiene un significado especial. Cuando se utiliza al principio de una palabra, se expande al nombre del directorio de inicio del usuario designado o, si no se nombra ningún usuario, al directorio de inicio del usuario actual.

```
[me@linuxbox ~]$ echo ~
/home/me
```

If user "foo" has an account, then it expands into this:
Si el usuario "foo" tiene una cuenta, entonces se expande a esto:

```
[me@linuxbox ~]$ echo ~foo
/home/foo
```

Arithmetic Expansion
Expansión aritmética

The shell allows arithmetic to be performed by expansion. This allows us to use the shell prompt as a calculator.
El shell permite que la aritmética se realice mediante expansión. Esto nos permite usar el indicador de shell como calculadora.

```
[me@linuxbox ~]$ echo $((2 + 2))
4
```

Arithmetic expansion uses the following form:
La expansión aritmética utiliza la siguiente forma:

```
$((expression))
```

where expression is an arithmetic expression consisting of values and arithmetic operators.Arithmetic expansion supports only integers (whole numbers, no decimals) but can perform quite a number of different operations. Table 7-1 describes a few of the supported operators.
donde expresión es una expresión aritmética que consta de valores y operadores aritméticos. La expansión aritmética solo admite números enteros (números enteros, no decimales) pero puede realizar una gran cantidad de operaciones diferentes. La Tabla 7-1 describe algunos de los operadores admitidos.

Table 7-1: Arithmetic
Tabla 7-1: Aritmética

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division (but remember, since expansion supports only integer arithmetic, results are integers)
%	Modulo, which simply means “remainder”
**	Exponentiation

Spaces are not significant in arithmetic expressions, and expressions may be nested. For example, to multiply 5 squared by 3, we can use this:
Los espacios no son significativos en las expresiones aritméticas y las expresiones pueden estar anidadas. Por ejemplo, para multiplicar 5 al cuadrado por 3, podemos usar esto:

```
[me@linuxbox ~]$ echo $(($((5**2)) * 3))
75
```

Single parentheses may be used to group multiple subexpressions. With this technique, we can rewrite the previous example and get the same result using a single expansion instead of two.
Se pueden usar paréntesis simples para agrupar múltiples subexpresiones. Con esta técnica, podemos reescribir el ejemplo anterior y obtener el mismo resultado usando una sola expansión en lugar de dos.

```
[me@linuxbox ~]$ echo $(((5**2) * 3))
75
```

Here is an example using the division and remainder operators. Notice the effect of integer division.
A continuación, se muestra un ejemplo que utiliza los operadores de división y resto. Observe el efecto de la división de enteros.

```
[me@linuxbox ~]$ echo Five divided by two equals $((5/2))
Five divided by two equals 2
[me@linuxbox ~]$ echo with $((5%2)) left over.
with 1 left over.
```

Arithmetic expansion is covered in greater detail in Chapter 34.
La expansión aritmética se trata con más detalle en el capítulo 34.

Brace Expansion
Expansión de soporte

Perhaps the strangest expansion is called brace expansion. With it, you can create multiple text strings from a pattern containing braces. Here's an example:
Quizás la expansión más extraña se llama expansión de corsé. Con él, puede crear varias cadenas de texto a partir de un patrón que contenga llaves. Aquí tienes un ejemplo:

```
[me@linuxbox ~]$ echo Front-{A,B,C}-Back
Front-A-Back Front-B-Back Front-C-Back
```

Patterns to be brace expanded may contain a leading portion called a preamble and a trailing portion called a postscript. The brace expression itself may contain either a comma-separated list of strings or a range of integers or single characters. The pattern may not contain unquoted whitespace.
Los patrones que se van a expandir con llaves pueden contener una parte inicial llamada preámbulo y una parte final llamada posdata. La expresión de llaves en sí puede contener una lista de cadenas separadas por comas o un rango de números enteros o caracteres individuales. El patrón no puede contener espacios en blanco sin comillas.

Here is an example using a range of integers:
Aquí hay un ejemplo que usa un rango de números enteros:

```
[me@linuxbox ~]$ echo Number_{1..5}
Number_1 Number_2 Number_3 Number_4 Number_5
```

In bash version 4.0 and newer, integers may also be zero-padded like so:
En la versión 4.0 de bash y posteriores, los enteros también se pueden rellenar con ceros así:

```
[me@linuxbox ~]$ echo {01..15}
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
[me@linuxbox ~]$ echo {001..15}
001 002 003 004 005 006 007 008 009 010 011 012 013 014 015
```

Here is a range of letters in reverse order:
Aquí hay un rango de letras en orden inverso:

```
[me@linuxbox ~]$ echo {Z..A}
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
```

Brace expansions may be nested.
Las expansiones de tirantes se pueden anidar.

```
[me@linuxbox ~]$ echo a{A{1,2},B{3,4}}b
aA1b aA2b aB3b aB4b
```

So, what is this good for? The most common application is to make lists of files or directories to be created. For example, if we were photographers and had a large collection of images that we wanted to organize into years and months, the first thing we might do is create a series of directories named in numeric "Year-Month" format. This way, the directory names would sort in chronological order. We could type out a complete list of directories, but that's a lot of work, and it's error-prone. Instead, we could do this:
Entonces, ¿para qué sirve esto? La aplicación más común es hacer listas de archivos o directorios a crear. Por ejemplo, si fuéramos fotógrafos y tuviéramos una gran colección de imágenes que quisiéramos organizar en años y meses, lo primero que podríamos hacer es crear una serie de directorios nombrados en formato numérico "Año-Mes". De esta manera, los nombres de los directorios se ordenarían en orden cronológico. Podríamos escribir una lista completa de directorios, pero eso es mucho trabajo y es propenso a errores. En cambio, podríamos hacer esto:

```
[me@linuxbox ~]$ mkdir Photos
[me@linuxbox ~]$ cd Photos
[me@linuxbox Photos]$ mkdir {2007..2009}-{01..12}
[me@linuxbox Photos]$ ls
```

```
2007-01 2007-07 2008-01 2008-07 2009-01 2009-07
2007-02 2007-08 2008-02 2008-08 2009-02 2009-08
2007-03 2007-09 2008-03 2008-09 2009-03 2009-09
2007-04 2007-10 2008-04 2008-10 2009-04 2009-10
2007-05 2007-11 2008-05 2008-11 2009-05 2009-11
2007-06 2007-12 2008-06 2008-12 2009-06 2009-12
```

Pretty slick!
Bastante resbaladizo

Parameter Expansion
Expansión de parámetros

We're going to touch only briefly on parameter expansion in this chapter, but we'll be covering it extensively later. It's a feature that is more useful in shell scripts than directly on the command line. Many of its capabilities have to do with the system's ability to store small chunks of data and to give each chunk a name. Many such chunks, more properly called variables, are available for your examination. For example, the variable named USER contains your username. To invoke parameter expansion and reveal the contents of USER , you would do this:
En este capítulo, solo tocaremos brevemente la expansión de parámetros, pero lo cubriremos ampliamente más adelante. Es una función que es más útil en los scripts de shell que directamente en la línea de comandos. Muchas de sus capacidades tienen que ver con la capacidad del sistema para almacenar pequeños fragmentos de datos y dar un nombre a cada fragmento. Muchos de esos fragmentos, más propiamente llamados variables, están disponibles para su examen. Por ejemplo, la variable denominada USUARIO contiene su nombre de usuario. Para invocar la expansión de parámetros y revelar el contenido de USER, debe hacer esto:

```
[me@linuxbox ~]$ echo $USER
me
```

To see a list of available variables, try this:
Para ver una lista de variables disponibles, intente esto:

```
[me@linuxbox ~]$ printenv | less
```

You may have noticed that with other types of expansion, if you mistype a pattern, the expansion will not take place, and the echo command will simply display the mistyped pattern. With parameter expansion, if you misspell the name of a variable, the expansion will still take place but will result in an empty string.
Es posible que haya notado que con otros tipos de expansión, si escribe mal un patrón, la expansión no se llevará a cabo y el comando echo simplemente mostrará el patrón mal escrito. Con la expansión de parámetros, si escribe mal el nombre de una variable, la expansión seguirá teniendo lugar, pero dará como resultado una cadena vacía.

```
[me@linuxbox ~]$ echo $SUER
[me@linuxbox ~]$
```

Command Substitution
Sustitución de comando

Command substitution allows us to use the output of a command as an expansion.
La sustitución de comandos nos permite usar la salida de un comando como expansión.

```
[me@linuxbox ~]$ echo $(ls)
Desktop Documents ls-output.txt Music Pictures Public Templates Videos
```

One of my favorites goes something like this:
Uno de mis favoritos es algo como esto:

```
[me@linuxbox ~]$ ls -l $(which cp)
-rwxr-xr-x 1 root root 71516 2007-12-05 08:58 /bin/cp
```

Here we passed the results of which cp as an argument to the ls command, thereby getting the listing of the cp program without having to know its full pathname. We are not limited to just simple commands. Entire pipe lines can be used (only partial output is shown here).
Aquí pasamos los resultados de los cuales cp como argumento al comando ls, obteniendo así la lista del programa cp sin tener que saber su ruta completa. No estamos limitados a simples comandos. Se pueden utilizar tuberías completas (aquí solo se muestra la salida parcial).

```
[me@linuxbox ~]$ file $(ls -d /usr/bin/* | grep zip)
/usr/bin/bunzip2:   symbolic link to `bzip2'
/usr/bin/bzip2:     ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.9,
stripped
/usr/bin/bzip2recover: ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.9,
stripped
/usr/bin/funzip:    ELF 32-bit LSB executable, Intel 80386, version 1
(SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.9,
stripped
/usr/bin/gpg-zip:   Bourne shell script text executable
/usr/bin/gunzip:    symbolic link to `../../bin/gunzip'
/usr/bin/gzip:      symbolic link to `../../bin/gzip'
/usr/bin/mzip:      symbolic link to `mtools'
```

In this example, the results of the pipeline became the argument list of the file command.
En este ejemplo, los resultados de la canalización se convirtieron en la lista de argumentos del comando de archivo.

There is an alternate syntax for command substitution in older shell programs that is also supported in bash . It uses backquotes instead of the dollar sign and parentheses.
Existe una sintaxis alternativa para la sustitución de comandos en programas de shell más antiguos que también es compatible con bash. Utiliza comillas inversas en lugar del signo de dólar y paréntesis.

```
[me@linuxbox ~]$ ls -l `which cp`
-rwxr-xr-x 1 root root 71516 2007-12-05 08:58 /bin/cp
```

Quoting
Comillado

Now that we've seen how many ways the shell can perform expansions, it's time to learn how we can control it. Take, for example, the following:
Ahora que hemos visto de cuántas formas el shell puede realizar expansiones, es hora de aprender cómo podemos controlarlo. Tomemos, por ejemplo, lo siguiente:

```
[me@linuxbox ~]$ echo this is a test
this is a test
```

or this one:
o este:

```
[me@linuxbox ~]$ echo The total is $100.00
The total is 00.00
```

In the first example, word splitting by the shell removed extra whitespace from the echo command's list of arguments. In the second example, parameter expansion substituted an empty string for the value of $1 because it was an undefined variable. The shell provides a mechanism called quoting to selectively suppress unwanted expansions.
En el primer ejemplo, la división de palabras por el shell eliminó los espacios en blanco adicionales de la lista de argumentos del comando echo. En el segundo ejemplo, la expansión de parámetros sustituyó una cadena vacía por el valor de $ 1 porque era una variable indefinida. El caparazón proporciona un mecanismo llamado cotización para suprimir selectivamente expansiones no deseadas.

Double Quotes
Doble comillas

The first type of quoting we will look at is double quotes. If we place text inside double quotes, all the special characters used by the shell lose their special meaning and are treated as ordinary characters. The exceptions are $ (dollar sign), \ (backslash), and `(backtick). This means that word splitting, pathname expansion, tilde expansion, and brace expansion are suppressed; however, parameter expansion, arithmetic expansion, and command substitution are still carried out. Using double quotes, we can cope with filenames containing embedded spaces. Say we were the unfortunate victim of a file called two words.txt. If we tried to use this on the command line, word splitting would cause this to be treated as two separate arguments rather than the desired single argument.
El primer tipo de cita que veremos son las comillas dobles. Si colocamos texto entre comillas dobles, todos los caracteres especiales utilizados por el shell pierden su significado especial y se tratan como caracteres ordinarios. Las excepciones son $ (signo de dólar), \ (barra invertida) y `(tilde invertido). Esto significa que se suprimen la división de palabras, la expansión de nombre de ruta, la expansión de tilde y la expansión de llaves; sin embargo, todavía se llevan a cabo la expansión de parámetros, la expansión aritmética y la sustitución de comandos. Usando comillas dobles, podemos hacer frente a nombres de archivos que contienen espacios incrustados. Digamos que fuimos la desafortunada víctima de un archivo llamado two words.txt. Si intentáramos usar esto en la línea de comando, la división de palabras haría que esto se tratara como dos argumentos separados en lugar del único argumento deseado.

```
[me@linuxbox ~]$ ls -l two words.txt
ls: cannot access two: No such file or directory
ls: cannot access words.txt: No such file or directory
```

By using double quotes, we stop the word splitting and get the desired result; further, we can even repair the damage.
Al usar comillas dobles, detenemos la división de palabras y obtenemos el resultado deseado; Además, incluso podemos reparar el daño.

```
[me@linuxbox ~]$ ls -l "two words.txt"
-rw-rw-r-- 1 me me 18 2018-02-20 13:03 two words.txt
[me@linuxbox ~]$ mv "two words.txt" two_words.txt
```

There! Now we don’t have to keep typing those pesky double quotes.
¡Allí! Ahora no tenemos que seguir escribiendo esas molestas comillas dobles.

Remember, parameter expansion, arithmetic expansion, and command substitution still take place within double quotes.
Recuerde, la expansión de parámetros, la expansión aritmética y la sustitución de comandos todavía tienen lugar entre comillas dobles.

```
[me@linuxbox ~]$ echo "$USER $((2+2)) $(cal)"
me 4 February 2020
Su Mo Tu We Th Fr Sa
                   1
      2  3  4  5  6  7  8
 9 10 11 12 13 14 15
```

```
16 17 18 19 20 21 22
23 24 25 26 27 28 29
```

We should take a moment to look at the effect of double quotes on command substitution. First let's look a little deeper at how word splitting words.
Deberíamos tomarnos un momento para observar el efecto de las comillas dobles en la sustitución de comandos. Primero, veamos un poco más a fondo cómo funciona la división de palabras.

In our earlier example, we saw how word splitting appears to remove extra spaces in our text.
En nuestro ejemplo anterior, vimos cómo la división de palabras parece eliminar espacios adicionales en nuestro texto.

```
[me@linuxbox ~]$ echo this is a test
this is a test
```

By default, word splitting looks for the presence of spaces, tabs, and newlines (line feed characters) and treats them as delimiters between words.
De forma predeterminada, la división de palabras busca la presencia de espacios, tabulaciones y nuevas líneas (caracteres de avance de línea) y los trata como delimitadores entre palabras.

This means unquoted spaces, tabs, and newlines are not considered to be part of the text. They serve only as separators. Because they separate the words into different arguments, our example command line contains a command followed by four distinct arguments. If we add double quotes:
Esto significa que los espacios sin comillas, las tabulaciones y las nuevas líneas no se consideran parte del texto. Sirven solo como separadores. Debido a que separan las palabras en diferentes argumentos, nuestra línea de comando de ejemplo contiene un comando seguido de cuatro argumentos distintos. Si agregamos comillas dobles:

```
[me@linuxbox ~]$ echo "this is a test"
this is a test
```

then word splitting is suppressed and the embedded spaces are not treated as delimiters; rather, they become part of the argument. Once the double quotes are added, our command line contains a command followed by a single argument.
luego, se suprime la división de palabras y los espacios incrustados no se tratan como delimitadores; más bien, se vuelven parte del argumento. Una vez que se agregan las comillas dobles, nuestra línea de comando contiene un comando seguido de un solo argumento.

The fact that newlines are considered delimiters by the word-splitting mechanism causes an interesting, albeit subtle, effect on command substitution. Consider the following:
El hecho de que las nuevas líneas se consideren delimitadores por el mecanismo de división de palabras provoca un efecto interesante, aunque sutil, en la sustitución de comandos. Considera lo siguiente:

```

```

```
[me@linuxbox ~]$ echo $(cal)
February 2020 Su Mo Tu We Th Fr Sa 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17
18 19 20 21 22 23 24 25 26 27 28 29
[me@linuxbox ~]$ echo "$(cal)"
    February 2020
Su Mo Tu We Th Fr Sa
                 1
       2 3 4 5 6 7 8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
```

In the first instance, the unquoted command substitution resulted in a command line containing 38 arguments. In the second, it resulted in a command line with one argument that includes the embedded spaces and newlines.
En la primera instancia, la sustitución de comandos sin comillas dio como resultado una línea de comandos que contenía 38 argumentos. En el segundo, resultó en una línea de comando con un argumento que incluye los espacios incrustados y las nuevas líneas.

Single Quotes
Comillas simples

If we need to suppress all expansions, we use single quotes. Here is a comparison of unquoted, double quotes, and single quotes:
Si necesitamos suprimir todas las expansiones, usamos comillas simples. Aquí hay una comparación de comillas simples, dobles y sin comillas:

```
[me@linuxbox ~]$ echo text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
text /home/me/ls-output.txt a b foo 4 me
[me@linuxbox~]$ echo "text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER"
text ~/*.txt {a,b} foo 4 me
[me@linuxbox~]$ echo 'text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER'
text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
```

As we can see, with each succeeding level of quoting, more and more of the expansions are suppressed.
Como podemos ver, con cada nivel sucesivo de cotización, se suprimen más y más expansiones.

Escaping Characters
Personajes que escapan

Sometimes we want to quote only a single character. To do this, we can precede a character with a backslash, which in this context is called the escape character. Often this is done inside double quotes to selectively prevent an expansion.

A veces queremos citar solo un carácter. Para hacer esto, podemos preceder un carácter con una barra invertida, que en este contexto se denomina carácter de escape. A menudo, esto se hace entre comillas dobles para evitar selectivamente una expansión.

```
[me@linuxbox ~]$ echo "The balance for user $USER is: \$5.00"
The balance for user me is: $5.00
```

It is also common to use escaping to eliminate the special meaning of a character in a filename. For example, it is possible to use characters in filenames that normally have special meaning to the shell. These would include $, ! , & , spaces, and others. To include a special character in a filename, we can do this:
También es común utilizar el escape para eliminar el significado especial de un carácter en un nombre de archivo. Por ejemplo, es posible utilizar caracteres en nombres de archivos que normalmente tienen un significado especial para el shell. Estos incluirían $,! , &, espacios y otros. Para incluir un carácter especial en un nombre de archivo, podemos hacer esto:

```
[me@linuxbox ~]$ mv bad\&filename good_filename
```

To allow a backslash character to appear, escape it by typing \ . Note that within single quotes, the backslash loses its special meaning and is treated as an ordinary character.
Para permitir que aparezca un carácter de barra invertida, escriba \ para escapar de él. Tenga en cuenta que entre comillas simples, la barra invertida pierde su significado especial y se trata como un carácter ordinario.

Backslash Escape Sequences
Secuencias de escape de barra invertida

In addition to its role as the escape character, the backslash is used as part of a notation to represent certain special characters called control codes.
Además de su función como carácter de escape, la barra invertida se usa como parte de una notación para representar ciertos caracteres especiales llamados códigos de control.

The first 32 characters in the ASCII coding scheme are used to transmit commands to Teletype-like devices. Some of these codes are familiar (tab, backspace, line feed, and carriage return), while others are not (null, end-of-transmission, and acknowledge).
Los primeros 32 caracteres del esquema de codificación ASCII se utilizan para transmitir comandos a dispositivos similares a teletipos. Algunos de estos códigos son familiares (tabulación, retroceso, avance de línea y retorno de carro), mientras que otros no lo son (nulo, fin de transmisión y reconocimiento).

Table 7-2 lists some of the common backslash escape sequences.
La Tabla 7-2 enumera algunas de las secuencias de escape de barra invertida comunes.

> Table 7-2: Backslash Escape Sequences
> Tabla 7-2: Secuencias de escape de barra invertida

Escape sequence	Meaning

Escape sequence	Meaning
\a	Bell (an alert that causes the computer to beep)
\b	Backspace
\n	Newline; on Unix-like systems, this produces a line feed
\r	Carriage return
\t	Tab

The idea behind this representation using the backslash originated in the C programming language and has been adopted by many others, including the shell.
La idea detrás de esta representación usando la barra invertida se originó en el lenguaje de programación C y ha sido adoptada por muchos otros, incluido el shell.

Adding the -e option to echo will enable interpretation of escape sequences. You can also place them inside $' ' . Here, using the sleep command, a simple program that just waits for the specified number of seconds and then exits, we can create a primitive countdown timer:
Agregar la opción -e a echo permitirá la interpretación de las secuencias de escape. También puede colocarlos dentro de $''. Aquí, usando el comando sleep, un programa simple que solo espera el número especificado de segundos y luego sale, podemos crear un temporizador de cuenta regresiva primitivo:

```
sleep 10; echo -e "Time's up\a"
```

We could also do this:
También podríamos hacer esto:

```
sleep 10; echo "Time's up" $'\a'
```

Summing Up
Resumen

As we move forward with using the shell, we will find that expansions and quoting will be used with increasing frequency, so it makes sense to get a good understanding of the way they work. In fact, it could be argued that they are the most important subjects to learn about the shell. Without a proper understanding of expansion, the shell will always be a source of mystery and confusion, with much of its potential power wasted.
A medida que avancemos en el uso del shell, descubriremos que las expansiones y las citas se usarán con una frecuencia cada vez mayor, por lo que tiene sentido comprender bien la forma en que funcionan. De hecho, se podría argumentar que son los temas más importantes para aprender sobre el caparazón. Sin una comprensión adecuada de la expansión, el caparazón siempre será una fuente de misterio y confusión, con gran parte de su poder potencial desperdiciado.