LPI certification 101 exam prep, Part 1

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. About this tutorial

What does this tutorial cover?

Welcome to "Linux fundamentals", the first of four tutorials designed to prepare you for the Linux Professional Institute's 101 exam. In this tutorial, we'll introduce you to **bash** (the standard Linux shell), show you how to take full advantage of standard Linux commands like **ls**, **cp**, and **mv**, explain Linux's permission and ownership model, and much more.

By the end of this tutorial, you'll have a solid grounding in Linux fundamentals and will even be ready to begin learning some basic Linux system administration tasks.

By the end of this *series* of tutorials (eight in all), you'll have the knowledge you need to become a Linux Systems Administrator and will be ready to attain an LPIC Level 1 certification from the Linux Professional Institute if you so choose.

Should I take this tutorial?

This tutorial (Part 1) is ideal for those who are new to Linux, or those who want to review or improve their understanding of fundamental Linux concepts, such as copying files, moving files, and creating symbolic and hard links. Along the way, we'll share plenty of hints, tips, and tricks to keep the tutorial "meaty" and practical, even for those with a good amount of previous Linux experience. For beginners, much of this material will be new, but more experienced Linux users may find this tutorial to be a great way of "rounding out" their fundamental Linux skills.

Also in this series are three other tutorials:

* *Part 2: Basic administration*
* *Part 3: Intermediate administration*
* *Part 4: Advanced administration*

About the author

For technical questions about the content of this tutorial, contact the author, Daniel Robbins, at *drobbins@gentoo.org*.

Residing in Albuquerque, New Mexico, Daniel Robbins is the President/CEO of *Gentoo Technologies, Inc.*, the creator of **Gentoo Linux**, an advanced Linux for the PC, and the **Portage** system, a next-generation ports system for Linux. He has also served as a contributing author for the Macmillan books *Caldera OpenLinux Unleashed*, *SuSE Linux Unleashed*, and *Samba Unleashed*. Daniel has been involved with computers in some fashion since the second grade, when he was first exposed to the Logo programming language as well as a potentially dangerous dose of Pac Man. This probably explains why he has since served as a Lead Graphic Artist at **SONY Electronic Publishing/Psygnosis**. Daniel enjoys spending time with his wife, Mary, and his new baby daughter, Hadassah.

Section 2. Introducing bash

The shell

If you've used a Linux system, you know that when you log in, you are greeted by a prompt that looks something like this:

```
$
```

The particular prompt that you see may look quite different. It may contain your system's hostname, the name of the current working directory, or both. But regardless of what your particular prompt may look like, there's one thing that's certain. The program that printed that prompt is called a "shell", and it's very likely that your particular shell is a program called **bash**.

Are you running bash?

You can check to see if you're running **bash** by typing:

```
$ echo $SHELL
/bin/bash
```

If the above line gave you an error or didn't respond similarly to our example, then you may be running a shell other than **bash**. In that case, most of this tutorial should still apply, but it would be advantageous for you to switch to **bash** for the sake of preparing for the 101 exam. (See Part 2 of this tutorial series for information on changing your shell using the **chsh** command.)

About bash

Bash, an acronym for "Bourne-again shell", is the default shell on most Linux systems. The shell's job is to obey your commands so that you can interact with your Linux system. When you're finished entering commands, you may instruct the shell to **exit** or **logout**, at which point you'll be returned to a login prompt.

By the way, you can also logout by pressing control-D at the **bash** prompt.

Using "cd"

As you've probably found, staring at your **bash** prompt isn't the most exciting thing in the world. So, let's start using **bash** to navigate around our filesystem. At the prompt, type the following (without the **$**):

```
$ cd /
```

We've just told **bash** that you want to work in /, also known as the *root* directory; all the directories on the system form a tree, and / is considered the top of this tree, or the root. **cd**

sets the directory where you are currently working, also known as the "current working directory".

Paths

To see bash's current working directory, you can type:

```
$ pwd
/
```

In the above example, the / argument to **cd** is called a *path*. It tells **cd** where we want to go. In particular, the / argument is an *absolute* path, meaning that it specifies a location relative to the root of the filesystem tree.

Absolute paths

Here are some other absolute paths:

```
/dev
/usr
/usr/bin
/usr/local/bin
```

As you can see, the one thing that all absolute paths have in common is that they begin with /. With a path of /usr/local/bin, we're telling **cd** to enter the / directory, then the usr directory under that, and then local and bin. Absolute paths are always evaluated by starting at / first.

Relative paths

The other kind of path is called a *relative path*. Bash, **cd**, and other commands always interpret these paths relative to the current directory. Relative paths never begin with a /. So, if we're in /usr:

```
$ cd /usr
```

Then, we can use a relative path to change to the /usr/local/bin directory:

```
$ cd local/bin
$ pwd
/usr/local/bin
```

Using ".."

Relative paths may also contain one or more .. directories. The .. directory is a special directory that points to the parent directory. So, continuing from the example above:

```
$ pwd
```

```
/usr/local/bin
$ cd ..
$ pwd
/usr/local
```

As you can see, our current directory is now /usr/local. We were able to go "backwards" one directory, relative to the current directory that we were in.

Using "..", continued

In addition, we can also add .. to an existing relative path, allowing us to go into a directory that's alongside one we are already in, for example:

```
$ pwd
/usr/local
$ cd ../share
$ pwd
/usr/share
```

Relative path examples

Relative paths can get quite complex. Here are a few examples, all without the resultant target directory displayed. Try to figure out where you'll end up after typing these commands:

```
$ cd /bin
$ cd ../usr/share/zoneinfo
```

```
$ cd /usr/X11R6/bin
$ cd ../lib/X11
```

```
$ cd /usr/bin
$ cd ../bin/../bin
```

Now, try them out and see if you got them right. :)

Understanding "."

Before we finish our coverage of **cd**, we need to discuss a few more things. First, there is another special directory called **.**, which means "the current directory". While this directory isn't used with the **cd** command, it's often used to execute some program in the current directory, as follows:

```
$ ./myprog
```

In the above example, the **myprog** executable residing in the current working directory will be executed.

cd and the home directory

If we wanted to change to our home directory, we could type:

```
$ cd
```

With no arguments, cd will change to your home directory, which is /root for the superuser and typically /home/username for a regular user. But what if we want to specify a file in our home directory? Maybe we want to pass a file argument to the **myprog** command. If the file lives in our home directory, we can type:

```
$ ./myprog /home/drobbins/myfile.txt
```

However, using an absolute path like that isn't always convenient. Thankfully, we can use the **~** (tilde) character to do the same thing:

```
$ ./myprog ~/myfile.txt
```

Other users' home directories

Bash will expand a lone **~** to point to your home directory, but you can also use it to point to other users' home directories. For example, if we wanted to refer to a file called fredsfile.txt in fred's home directory, we could type:

```
$ ./myprog ~fred/fredsfile.txt
```

Section 3. Using Linux commands

Introducing "ls"

Now, we'll take a quick look at the **ls** command. Very likely, you're already familiar with **ls** and know that typing it by itself will list the contents of the current working directory:

```
$ cd /usr
$ ls
X11R6      doc         i686-pc-linux-gnu  lib      man          sbin   ssl
bin        gentoo-x86  include            libexec  portage      share  tmp
distfiles  i686-linux  info               local    portage.old  src
```

By specifying the **-a** option, you can see all of the files in a directory, including hidden files -- those that begin with **.**. As you can see in the following example, **ls -a** reveals the **.** and **..** special directory links:

```
$ ls -a
.      bin        gentoo-x86         include  libexec  portage      share  tmp
..     distfiles  i686-linux         info     local    portage.old  src
X11R6  doc        i686-pc-linux-gnu  lib      man      sbin         ssl
```

Long directory listings

You can also specify one or more files or directories on the **ls** command line. If you specify a file, **ls** will show that file only. If you specify a directory, **ls** will show the *contents* of the directory. The **-l** option comes in very handy when you need to view permissions, ownership, modification time, and size information in your directory listing.

Long directory listings, continued

In the following example, we use the **-l** option to display a full listing of my /usr directory.

```
$ ls -l /usr
drwxr-xr-x    7 root     root          168 Nov 24 14:02 X11R6
drwxr-xr-x    2 root     root        14576 Dec 27 08:56 bin
drwxr-xr-x    2 root     root         8856 Dec 26 12:47 distfiles
lrwxrwxrwx    1 root     root            9 Dec 22 20:57 doc -> share/doc
drwxr-xr-x   62 root     root         1856 Dec 27 15:54 gentoo-x86
drwxr-xr-x    4 root     root          152 Dec 12 23:10 i686-linux
drwxr-xr-x    4 root     root           96 Nov 24 13:17 i686-pc-linux-gnu
drwxr-xr-x   54 root     root         5992 Dec 24 22:30 include
lrwxrwxrwx    1 root     root           10 Dec 22 20:57 info -> share/info
drwxr-xr-x   28 root     root        13552 Dec 26 00:31 lib
drwxr-xr-x    3 root     root           72 Nov 25 00:34 libexec
drwxr-xr-x    8 root     root          240 Dec 22 20:57 local
lrwxrwxrwx    1 root     root            9 Dec 22 20:57 man -> share/man
lrwxrwxrwx    1 root     root           11 Dec  8 07:59 portage -> gentoo-x86/
drwxr-xr-x   60 root     root         1864 Dec  8 07:55 portage.old
drwxr-xr-x    3 root     root         3096 Dec 22 20:57 sbin
drwxr-xr-x   46 root     root         1144 Dec 24 15:32 share
drwxr-xr-x    8 root     root          328 Dec 26 00:07 src
drwxr-xr-x    6 root     root          176 Nov 24 14:25 ssl
```

```
lrwxrwxrwx    1 root     root           10 Dec 22 20:57 tmp -> ../var/tmp
```

The first column displays permissions information for each item in the listing. I'll explain how to interpret this information in a bit. The next column lists the number of links to each filesystem object, which we'll gloss over now but return to later. The third and fourth columns list the owner and group, respectively. The fifth column lists the object size. The sixth column is the "last modified" time or "mtime" of the object. The last column is the object's name. If the file is a symbolic link, you'll see a trailing **->** and the path to which the symbolic link points.

Looking at directories

Sometimes, you'll want to look *at* a directory, rather than inside it. For these situations, you can specify the **-d** option, which will tell **ls** to look at any directories that it would normally look inside:

```
$ ls -dl /usr /usr/bin /usr/X11R6/bin ../share
drwxr-xr-x    4 root     root           96 Dec 18 18:17 ../share
drwxr-xr-x   17 root     root          576 Dec 24 09:03 /usr
drwxr-xr-x    2 root     root         3192 Dec 26 12:52 /usr/X11R6/bin
drwxr-xr-x    2 root     root        14576 Dec 27 08:56 /usr/bin
```

Recursive and inode listings

So you can use **-d** to look *at* a directory, but you can also use **-R** to do the opposite -- not just look inside a directory, but recursively look inside all the directories inside that directory! We won't include any example output for this option (since it's generally voluminous), but you may want to try a few **ls -R** and **ls -Rl** commands to get a feel for how this works.

Finally, the **-i** ls option can be used to display the *inode numbers* of the filesystem objects in the listing:

```
$ ls -i /usr
   1409 X11R6        314258 i686-linux           43090 libexec        13394 sbin
   1417 bin            1513 i686-pc-linux-gnu     5120 local          13408 share
   8316 distfiles      1517 include                776 man            23779 src
     43 doc            1386 info                 93892 portage        36737 ssl
  70744 gentoo-x86     1585 lib                   5132 portage.old       784 tmp
```

Understanding inodes, part 1

Every object on a filesystem is assigned a unique index, called an inode number. This might seem trivial, but understanding inodes is essential to understanding many filesystem operations. For example, consider the **.** and **..** links that appear in every directory. To fully understand what a **..** directory actually is, we'll first take a look at /usr/local's inode number:

```
$ ls -id /usr/local
   5120 /usr/local
```

The /usr/local directory has an inode number of 5120. Now, let's take a look at the inode number of /usr/local/bin/..:

```
$ ls -id /usr/local/bin/..
   5120 /usr/local/bin/..
```

Understanding inodes, part 2

As you can see, /usr/local/bin/.. has the same inode number as /usr/local! Here's how can we come to grips with this shocking revelation. In the past, we've considered /usr/local to be *the* directory itself. Now, we discover that inode 5120 is in fact *the* directory, and we have found two directory entries (called "links") that point to this inode. Both /usr/local and /usr/local/bin/.. are *links* to inode 5120. Although inode 5120 exists in only one place on disk, multiple things link to it.

Understanding inodes, part 3

In fact, we can see the total number of times that inode 5120 is referenced by using the **ls -dl** command:

```
$ ls -dl /usr/local
drwxr-xr-x    8 root     root          240 Dec 22 20:57 /usr/local
```

If we take a look at the second column from the left, we see that the directory /usr/local (inode 5120) is referenced eight times. On my system, here are the various paths that reference this inode:

```
/usr/local
/usr/local/.
/usr/local/bin/..
/usr/local/games/..
/usr/local/lib/..
/usr/local/sbin/..
/usr/local/share/..
/usr/local/src/..
```

mkdir

Let's take a quick look at the **mkdir** command, which can be used to create new directories. The following example creates three new directories, tic, tac, and toe, all under /tmp:

```
$ cd /tmp
$ mkdir tic tac toe
```

By default, the **mkdir** command doesn't create parent directories for you; the entire path up to the next-to-last element needs to exist. So, if you want to create the directories won/der/ful, you'd need to issue three separate **mkdir** commands:

```
$ mkdir won/der/ful
```

```
mkdir: cannot create directory `won/der/ful': No such file or directory
$ mkdir won
$ mkdir won/der
$ mkdir won/der/ful
```

mkdir -p

However, mkdir has a handy **-p** option that tells mkdir to create any missing parent directories, as can be seen here:

```
$ mkdir -p easy/as/pie
```

All in all, pretty straightforward. To learn more about the **mkdir** command, type **man mkdir** to read the manual page. This will work for nearly all commands covered here (such as **man ls**), except for **cd**, which is built-in to **bash**.

touch

Now, we're going to take a quick look at the **cp** and **mv** commands, used to copy, rename, and move files and directories. To begin this overview, we'll first use the **touch** command to create a file in /tmp:

```
$ cd /tmp
$ touch copyme
```

The **touch** command updates the "mtime" of a file if it exists (recall the sixth column in **ls -l** output). If the file doesn't exist, then a new, empty file will be created. You should now have a /tmp/copyme file with a size of zero.

echo and redirection

Now that the file exists, let's add some data to the file. We can do this using the **echo** command, which takes its arguments and prints them to standard output. First, the echo command by itself:

```
$ echo "firstfile"
firstfile
```

echo and redirection

Now, the same **echo** command with output redirection:

```
$ echo "firstfile" > copyme
```

The greater-than sign tells the shell to write **echo**'s output to a file called copyme. This file will be created if it doesn't exist, and will be overwritten if it does exist. By typing **ls -l**, we can

see that the copyme file is 10 bytes long, since it contains the word **firstfile** and the newline character:

```
$ ls -l copyme
-rw-r--r--    1 root     root           10 Dec 28 14:13 copyme
```

cat and cp

To display the contents of the file on the terminal, use the **cat** command:

```
$ cat copyme
firstfile
```

Now, we can use a basic invocation of the **cp** command to create a copiedme file from the original copyme file:

```
$ cp copyme copiedme
```

Upon investigation, we find that they are truly separate files; their inode numbers are different:

```
$ ls -i copyme copiedme
  648284 copiedme   650704 copyme
```

mv

Now, let's use the **mv** command to rename "copiedme" to "movedme". The inode number will remain the same; however, the filename that points to the inode will change.

```
$ mv copiedme movedme
$ ls -i movedme
  648284 movedme
```

A moved file's inode number will remain the same as long as the destination file resides on the same filesystem as the source file. We'll take a closer look at filesystems in Part 3 of this tutorial series.

Section 4. Creating links and removing files

Hard links

We've mentioned the term *link* when referring to the relationship between directory entries and inodes. There are actually two kinds of links available on Linux. The kind we've discussed so far are called *hard links*. A given inode can have any number of hard links, and the inode will persist on the filesystem until all the hard links disappear. New hard links can be created using the **ln** command:

```
$ cd /tmp
$ touch firstlink
$ ln firstlink secondlink
$ ls -i firstlink secondlink
  15782 firstlink    15782 secondlink
```

Hard links, continued

As you can see, hard links work on the inode level to point to a particular file. On Linux systems, hard links have several limitations. For one, you can only make hard links to files, not directories. That's right; even though **.** and **..** are system-created hard links to directories, you (not even as the "root" user) aren't allowed to create any of your own.

The second limitation of hard links is that they can't span filesystems. This means that you can't create a link from /usr/bin/bash to /bin/bash if your / and /usr directories exist on separate filesystems.

Symbolic links

In practice, symbolic links (or "symlinks") are used more often than hard links. Symlinks are a special file type where the link refers to another file by name, rather than directly to the inode. Symlinks do not prevent a file from being deleted; if the target file disappears, then the symlink will just be unusable, or "broken".

Symbolic links, continued

A symbolic link can be created by passing the **-s** option to **ln**.

```
$ ln -s secondlink thirdlink
$ ls -l firstlink secondlink thirdlink
-rw-rw-r--    2 agriffis agriffis        0 Dec 31 19:08 firstlink
-rw-rw-r--    2 agriffis agriffis        0 Dec 31 19:08 secondlink
lrwxrwxrwx    1 agriffis agriffis       10 Dec 31 19:39 thirdlink -> secondlink
```

Symbolic links can be distinguished in **ls -l** output from normal files in three ways. First, notice that the first column contains an **l** character to signify the symbolic link. Second, the size of the symbolic link is the number of characters in the target (**secondlink** in this case). Third, the last column of the output displays the target filename.

Symlinks in-depth, part 1

Symbolic links are generally more flexible than hard links. You can create a symbolic link to any type of filesystem object, including directories. And because the implementation of symbolic links is based on paths (not inodes), it's perfectly fine to create a symbolic link that points to an object on another filesystem. However, this fact can also make symbolic links tricky to understand.

Symlinks in-depth, part 2

Consider a situation where we want to create a link in /tmp that points to /usr/local/bin. Should we type this:

```
$ ln -s /usr/local/bin bin1
$ ls -l bin1
lrwxrwxrwx    1 root     root           14 Jan  1 15:42 bin1 -> /usr/local/bin
```

Or alternatively:

```
$ ln -s ../usr/local/bin bin2
$ ls -l bin2
lrwxrwxrwx    1 root     root           16 Jan  1 15:43 bin2 -> ../usr/local/bin
```

Symlinks in-depth, part 3

As you can see, both symbolic links point to the same directory. However, if our second symbolic link is ever moved to another directory, it will be "broken" because of the relative path:

```
$ ls -l bin2
lrwxrwxrwx    1 root     root           16 Jan  1 15:43 bin2 -> ../usr/local/bin
$ mkdir mynewdir
$ mv bin2 mynewdir
$ cd mynewdir
$ cd bin2
bash: cd: bin2: No such file or directory
```

Because the directory /tmp/usr/local/bin doesn't exist, we can no longer change directories into bin2; in other words, bin2 is now broken.

Symlinks in-depth, part 4

For this reason, it is sometimes a good idea to avoid creating symbolic links with relative path information. However, there are many cases where relative symbolic links come in handy. Consider an example where you want to create an alternate name for a program in /usr/bin:

```
# ls -l /usr/bin/keychain
-rwxr-xr-x    1 root     root        10150 Dec 12 20:09 /usr/bin/keychain
```

Symlinks in-depth, part 5

As the root user, you may want to create an alternate name for "keychain", such as "kc". In this example, we have root access, as evidenced by our bash prompt changing to "#". We need root access because normal users aren't able to create files in /usr/bin. As root, we could create an alternate name for keychain as follows:

```
# cd /usr/bin
# ln -s /usr/bin/keychain kc
```

Symlinks in-depth, part 6

While this solution will work, it will create problems if we decide that we want to move both files to /usr/local/bin:

```
# mv /usr/bin/keychain /usr/bin/kc /usr/local/bin
```

Because we used an *absolute* path in our symbolic link, our **kc** symlink is still pointing to /usr/bin/keychain, which no longer exists -- another broken symlink. Both relative and absolute paths in symbolic links have their merits, and you should use a type of path that's appropriate for your particular application. Often, either a relative or absolute path will work just fine. In this case, the following example would have worked:

```
# cd /usr/bin
# ln -s keychain kc
# ls -l kc
lrwxrwxrwx    1 root     root            8 Jan  5 12:40 kc -> keychain
```

rm

Now that we know how to use **cp**, **mv**, and **ln**, it's time to learn how to remove objects from the filesystem. Normally, this is done with the **rm** command. To remove files, simply specify them on the command line:

```
$ cd /tmp
$ touch file1 file2
$ ls -l file1 file2
-rw-r--r--    1 root     root            0 Jan  1 16:41 file1
-rw-r--r--    1 root     root            0 Jan  1 16:41 file2
$ rm file1 file2
$ ls -l file1 file2
ls: file1: No such file or directory
ls: file2: No such file or directory
```

rmdir

To remove directories, you have two options. You can remove all the objects inside the directory and then use **rmdir** to remove the directory itself:

```
$ mkdir mydir
$ touch mydir/file1
$ rm mydir/file1
$ rmdir mydir
```

rm and directories

Or, you can use the *recursive force* options of the **rm** command to tell rm to remove the directory you specify, as well as all objects contained in the directory:

```
$ rm -rf mydir
```

Generally, **rm -rf** is the preferred method of removing a directory tree. Be very careful when using **rm -rf**, since its power can be used for both good and evil. :)

Section 5. Introducing wildcards

Introducing wildcards

In your day-to-day Linux use, there are many times when you may need to perform a single operation (such as **rm**) on many filesystem objects at once. In these situations, it can often be cumbersome to type in many files on the command line:

```
$ rm file1 file2 file3 file4 file5 file6 file7 file8
```

Introducing wildcards, continued

To solve this problem, you can take advantage of Linux's built-in wildcard support. This support, also called "globbing" (for historical reasons), allows you to specify multiple files at once by using a *wildcard pattern*. Bash and other Linux commands will interpret this pattern by looking on disk and finding any files that match it. So, if you had files file1 through file8 in the current working directory, you could remove these files by typing:

```
$ rm file[1-8]
```

Or if you simply wanted to remove all files whose names begin with file, you could type:

```
$ rm file*
```

Understanding non-matches

Or if you wanted to list all the filesystem objects in /etc beginning with **g**, you could type:

```
$ ls -d /etc/g*
/etc/gconf  /etc/ggi  /etc/gimp  /etc/gnome  /etc/gnome-vfs-mime-magic  /etc/gpm  /etc/
```

Now, what happens if you specify a pattern that doesn't match any filesystem objects? In the following example, we try to list all the files in /usr/bin that begin with **asdf** and end with **jkl**:

```
$ ls -d /usr/bin/asdf*jkl
ls: /usr/bin/asdf*jkl: No such file or directory
```

Understanding non-matches, continued

Here's what happened. Normally, when we specify a pattern, that pattern matches one or more files on the underlying filesystem, and **bash** *replaces the pattern with a space-separated list of all matching objects* . However, when the pattern doesn't produce any matches, **bash** *leaves the argument, wildcards and all, as is.* So when "ls" can't find the file /usr/bin/asdf*jkl, it gives an error. The operative rule here is that *glob patterns are expanded only if they match objects in the filesystem*.

Wildcard syntax: *

Now that we understand *how* globbing works, let's review wildcard syntax. You can use several special characters for wildcard expansion; here's one:

```
*
```

***** will match zero or more characters. It means "anything can go here". Examples:

* **/etc/g*** matches all files in /etc that begin with **g**.
* **/tmp/my*1** matches all files in /tmp that begin with **my** and end with **1**.

Wildcard syntax: ?

```
?
```

? matches any single character. Examples:

* **myfile?** matches any file whose name consists of **myfile** followed by a single character.
* **/tmp/notes?txt** would match both **/tmp/notes.txt** and **/tmp/notes_txt**, if they exist.

Wildcard syntax: []

```
[]
```

This wildcard is like a **?**, but allows more specificity. To use this wildcard, place any characters you'd like to match inside the **[]**. The resultant expression will match a single occurrence of any of these characters. You can also use **-** to specify a range, and even combine ranges. Examples:

* **myfile[12]** will match **myfile1** and **myfile2**. The wildcard will be expanded as long as at least one of these files exists in the current directory.
* **[Cc]hange[Ll]og** will match **Changelog**, **ChangeLog**, **changeLog**, and **changelog**. As you can see, using bracket wildcards can be useful for matching variations in capitalization.
* **ls /etc/[0-9]*** will list all files in /etc that begin with a number.
* **ls /tmp/[A-Za-z]*** will list all files in /tmp that begin with an upper or lower-case letter.

Wildcard syntax: [!]

```
[!]
```

The **[!]** construct is similar to the **[]** construct, except rather than matching any characters inside the brackets, it'll match any character, as long as it is *not* listed between the **[!** and **]**. Examples:

* **rm myfile[!9]** will remove all files named **myfile** plus a single character, except for **myfile9**.

Wildcard caveats

Here are some caveats to watch out for when using wildcards. Since **bash** treats wildcard-related characters (**?**, **[**, **]**, *****) specially, you need to take special care when typing in an argument to a command that contains these characters. For example, if you want to create a file that contains the string **[fo]***, the following command may not do what you want:

```
$ echo [fo]* > /tmp/mynewfile.txt
```

If the pattern **[fo]*** matches any files in the current working directory, then you'll find the names of those files inside /tmp/mynewfile.txt rather than a literal **[fo]*** like you were expecting. The solution? Well, one approach is to surround your characters with single quotes, which tell **bash** to perform absolutely no wildcard expansion on them:

```
$ echo '[fo]*' > /tmp/mynewfile.txt
```

Wildcard caveats, continued

Using this approach, your new file will contain a literal **[fo]*** as expected. Alternatively, you could use backslash escaping to tell **bash** that **[**, **]** and ***** should be treated literally rather than as wildcards:

```
$ echo \[fo\]\* > /tmp/mynewfile.txt
```

Both approaches will work identically. Since we're talking about backslash expansion, now would be a good time to mention that in order to specify a literal ****, you can either enclose it in single quotes as well, or type **** instead (it will be expanded to ****).

Single vs. double quotes

Note that double quotes will work similarly to single quotes, but will still allow **bash** to do some limited expansion. Therefore, single quotes are your best bet when you are truly interested in passing literal text to a command. For more information on wildcard expansion, type **man 7 glob**. For more information on quoting in **bash**, type **man 8 glob** and read the section titled **QUOTING**. If you're planning to take the LPI exams, consider this a homework assignment. :)

Section 6. Resources and feedback

Resources and homework

Congratulations; you've reached the end of "Linux fundamentals". I hope that this tutorial has helped you to firm up your foundational Linux knowledge. Please join us in our next tutorial on "Basic administration," where we'll build on the foundation laid here, covering topics like regular expressions, ownership and permissions, user account management, and more. The other tutorials in this series are:

* *Part 2: Basic administration*
* *Part 3: Intermediate administration*
* *Part 4: Advanced administration*

And remember, by completing this tutorial series, you'll be prepared to attain your LPIC Level 1 Certification from the Linux Professional Institute. Speaking of LPIC certification, if this is something you're interested in, then we recommend that you study the following resources, which augment the material covered in this tutorial:

In the *Bash by example* article series on developerWorks, I show you how to use **bash** programming constructs to write your own **bash** scripts. This bash series (particularly Parts 1 and 2) will be good preparation for the LPIC Level 1 exam:

* *Bash by example, Part 1: Fundamental programming in the Bourne-again shell*
* *Bash by example, Part 2: More bash programming fundamentals*
* *Bash by example, Part 3: Exploring the ebuild system*

I highly recommend the *Technical FAQ for Linux users*, a 50-page in-depth list of frequently asked Linux questions, along with detailed answers. The FAQ itself is in PDF (Adobe Acrobat) format. If you're a beginning or intermediate Linux user, you really owe it to yourself to check this FAQ out.

If you're not too familiar with the **vi** editor, I strongly recommend that you check out my *Vi intro -- the cheat sheet method tutorial* . This tutorial will give you a gentle yet fast-paced introduction to this powerful text editor. Consider this must-read material if you don't know how to use **vi**.

Your feedback

I look forward to getting your feedback on this tutorial. Additionally, you are welcome to contact me directly at *drobbins@gentoo.org*.

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats

from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.