

Shell Utilities

LinuXpert Systems

No.5, Dr. Ambedkar St, Rathinapuram,

Nanganallur, Chennai – 600 061.

Phone: 98841 65649, 90808 01382

E-mail: baskar@linuxpert.in

TABLE OF CONTENTS

1. echo 1

2. printf 8

3. alias 14

4. history 20

5. clear 27

6. env 30

7. export 36

8. source 43

1. echo

The `echo` command is a shell built-in command (also available as an external command on some systems) that **prints text, variables, or command output to the terminal or to a file**.

It is one of the most commonly used commands in shell scripting.

Syntax

```
echo [OPTION]... [STRING]...
```

- **OPTION** – Modifies the behaviour of `echo`.
- **STRING** – The text, variables, or expressions to display.

Basic Examples

1. Print text

```
echo Hello
```

Output:

```
Hello
```

---

2. Print multiple words

```
echo Hello World
```

Output:

```
Hello World
```

---

3. Print special characters

```
echo "Linux is awesome!"
```

Output:

```
Linux is awesome!
```

Printing Variables

Example

```
name="John"
```

```
echo $name
```

Output

```
John
```

or

```
echo "Hello $name"
```

Output

```
Hello John
```

Display Environment Variables

```
echo $HOME
```

Output

```
/home/user
```

```
echo $PATH
```

Displays your PATH environment variable.

```
echo $USER
```

Displays the current username.

Escape Characters

Normally,

```
echo "Line1\nLine2"
```

Output

```
Line1\nLine2
```

To interpret escape sequences, use **`-e`**.

```
echo -e "Line1\nLine2"
```

Output

```
Line1
Line2
```

Common Escape Sequences

Escape	Meaning
`\n`	New line
`\t`	Horizontal tab
`\\`	Backslash
`\"`	Double quote
`\a`	Alert (bell)
`\b`	Backspace
`\r`	Carriage return

Example

```
echo -e "Name:\tJohn"
```

Output

```
Name:	John
```

Suppress the Newline

Normally `echo` adds a newline at the end.

```
echo "Hello"
echo "World"
```

Output

```
Hello
World
```

Using `-n`:

```
echo -n "Hello "
echo "World"
```

Output

```
Hello World
```

Redirect Output to a File

Overwrite a file:

```
echo "Hello Linux" > file.txt
```

Append to a file:

```
echo "Another Line" >> file.txt
```

Command Substitution

```
echo "Today is $(date)"
```

Example Output

```
Today is Mon Jul 27 15:30:42 IST 2026
```

Arithmetic

```
echo $((10 + 5))
```

Output

```
15
```

Quotes

Double Quotes

Variables are expanded.

```
name="Alice"
```

```
echo "Hello $name"
```

Output

```
Hello Alice
```

Single Quotes

Variables are treated literally.

```
echo 'Hello $name'
```

Output

```
Hello $name
```

Display Multiple Variables

```
user="alice"
city="London"
```

```
echo "$user lives in $city"
```

Output :

```
alice lives in London
```

Using Wildcards

```
echo *.txt
```

Output

```
file1.txt file2.txt notes.txt
```

The shell expands `*.txt` before `echo` runs.

Useful in Shell Scripts

```
#!/bin/bash

echo "Starting Backup..."

echo "Backup Complete!"
```

Common Options

Option	Description
`-n`	Do not print the trailing newline
`-e`	Enable interpretation of backslash escapes
`-E`	Disable interpretation of backslash escapes (often the default)

Practical Examples

Display the current shell:

```
echo $SHELL
```

Display the hostname:

```
echo $HOSTNAME
```

Create a simple configuration file:

```
echo "PORT=8080" > app.conf
echo "DEBUG=true" >> app.conf
```

Show today's date:

```
echo "Current Date: $(date)"
```

Print coloured text (ANSI escape codes):

```
echo -e "\e[32mSuccess\e[0m"
```

Exit Status

- `0` – Command executed successfully.
- Non-zero – Rare for `echo`; errors typically occur only in unusual circumstances (for example, write failures such as a full disk or broken pipe).

You can check the exit status with:

```
echo "Hello"
echo $?
```

Output:

```
Hello

0
```

Notes

- `echo` is usually a **shell built-in**, making it very fast.
- Different shells (`bash`, `zsh`, `dash`, etc.) may implement `echo` slightly differently, especially regarding escape sequences. For portable shell scripts, `printf` is generally preferred over `echo` when you need predictable formatting.

Summary

Feature	Description
Purpose	Display text, variables, and command output
Shell Built-in	Yes (in most shells)
Print Variables	`echo $VAR`
Enable Escapes	`echo -e`
No Newline	`echo -n`
Redirect Output	`>` or `>>`
Script Usage	Display messages, debug output, generate files

2. printf

The **`printf`** command formats and prints text to the terminal or another output destination. Unlike `echo`, **`printf`** gives you precise control over formatting, making it the preferred choice in shell scripts.

Syntax

```
printf FORMAT [ARGUMENT]...
```

- `FORMAT` – A format string containing plain text and format specifiers.
- `ARGUMENT` – Values that replace the format specifiers.

Common Format Specifiers

Specifier	Description	Example
`%s`	String	`"Hello"`
`%d`	Decimal integer	`25`
`%i`	Integer	`25`
`%f`	Floating-point number	`3.14`
`%x`	Hexadecimal (lowercase)	`ff`
`%X`	Hexadecimal (uppercase)	`FF`
`%o`	Octal	`755`
`%c`	Single character	`A`
`%%`	Print a % character	`%`

Common Escape Sequences

Escape	Meaning
`\n`	New line
`\t`	Horizontal tab
`\\`	Backslash (\)
`\"`	Double quote
`\r`	Carriage return
`\b`	Backspace

Examples

1. Print simple text

```
printf "Hello World\n"
```

Output:

```
Hello World
```

---

2. Print a string

```
printf "Name: %s\n" "Alice"
```

Output:

```
Name: Alice
```

---

3. Print integers

```
printf "Age: %d\n" 25
```

Output:

```
Age: 25
```

---

4. Print floating-point numbers

```
printf "Pi = %.2f\n" 3.14159
```

Output:

```
Pi = 3.14
```

`%.2f` means print the number with **2 decimal places**.

5. Multiple values

```
printf "%s is %d years old.\n" "John" 30
```

Output:

```
John is 30 years old.
```

---

6. Width formatting

```
printf "|%10s|\n" "Linux"
```

Output:

```
|     Linux|
```

Right-aligned in a field of 10 characters.

Left-aligned:

```
printf "|%-10s|\n" "Linux"
```

Output:

```
|Linux     |
```

---

7. Zero padding

```
printf "%05d\n" 42
```

Output:

```
00042
```

---

8. Print hexadecimal

```
printf "%x\n" 255
```

Output:

```
ff
```

9. Print octal

```
printf "%o\n" 64
```

Output:

```
100
```

---

10. Print a percent sign

```
printf "Completed: 75%%\n"
```

```
Output:
```

```
Completed: 75%
```

---

11. Reuse the format string

```
printf "%s\n" one two three
```

```
Output:
```

```
one
two
three
```

```
The format string is reused automatically for additional arguments.
```

---

12. Save formatted output to a variable (Bash)

```
printf -v message "User: %s, ID: %d" "Alice" 1001
```

```
echo "$message"
```

```
Output:
```

```
User: Alice, ID: 1001
```

```
The -v option stores the formatted output in a variable instead of printing it.
```

`printf` vs `echo`

Feature	`printf`	`echo`
Automatic newline	No	Yes
Supports formatted output	Yes	No
Predictable behaviour	Yes	Can vary between shells
Alignment and padding	Yes	No
Floating-point formatting	Yes	No

Example:

```
echo "Hello"
```

Output:

```
Hello
```

```
printf "Hello"
```

Output:

```
Hello
```

Notice that `printf` does **not** automatically add a newline.

Common Options

Option	Description
`-v variable`	Store output in a shell variable instead of printing it

Exit Status

- `0` – Success
- Non-zero – Invalid format string or other error

Real-world Uses

- Display formatted reports in shell scripts.
- Align columns of output.
- Generate configuration files.
- Create log messages.
- Format numbers with leading zeros.
- Build strings without using external tools.

Summary

- `printf` prints **formatted output**.
- It provides much greater control than `echo`.
- It supports strings, numbers, floating-point values, alignment, padding, hexadecimal, octal, and escape sequences.
- It does **not** automatically append a newline, so include `\n` when needed.
- It is the recommended command for producing reliable, portable output in Bash scripts.

3. alias

The **`alias`** command is a **Bash built-in** used to create **shortcuts** for longer or frequently used commands. An alias replaces a command name with another command or a sequence of commands, making it quicker and easier to type.

Syntax

```
alias [name='command']
```

To remove an alias:

```
unalias alias_name
```

Purpose

- Create command shortcuts.
- Save typing for long commands.
- Add default options to commands.
- Customize your shell environment.

Basic Examples

1. List all aliases

```
alias
```

Example output:

```
alias ll='ls -alF'
alias la='ls -A'
alias l='ls -CF'
```

---

2. Display a specific alias

```
alias ll
```

Output:

```
alias ll='ls -alF'
```

3. Create an alias

```
alias ll='ls -l'
```

Now:

```
ll
```

is equivalent to:

```
ls -l
```

---

4. Alias with multiple options

```
alias la='ls -lah'
```

Running:

```
la
```

actually executes:

```
ls -lah
```

---

5. Alias with another command

```
alias cls='clear'
```

Now:

```
cls
```

clears the terminal.

---

6. Alias with multiple commands

```
alias update='sudo apt update && sudo apt upgrade'
```

Running:

```
update
```

performs both commands.

7. Override a command

```
alias rm='rm -i'
```

Now every `rm` asks for confirmation before deleting files.

Example:

```
rm file.txt
```

Output:

```
rm: remove regular file 'file.txt'?
```

---

8. Remove an alias

```
unalias ll
```

Now `ll` no longer exists.

---

9. Remove all aliases

```
unalias -a
```

---

10. Temporarily bypass an alias

Suppose:

```
alias ls='ls --color=auto'
```

To run the original `ls`:

```
\ls
```

or

```
command ls
```

Commonly Used Aliases

```
alias ll='ls -l'
alias la='ls -la'
alias l='ls -CF'
alias cls='clear'
alias h='history'
alias c='cat'
alias grep='grep --color=auto'
alias df='df -h'
alias du='du -sh'
alias mkdir='mkdir -pv'
alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
```

Make Aliases Permanent

Aliases created at the command line are **temporary** and disappear when you close the terminal.

To make them permanent, add them to your shell configuration file.

For Bash:

```
nano ~/.bashrc
```

Add:

```
alias ll='ls -lah'
alias cls='clear'
```

Save the file, then reload it:

```
source ~/.bashrc
```

Check if a Command is an Alias

Use:

```
type ls
```

Example output:

```
ls is aliased to `ls --color=auto'
```

or

```
type ll
```

Output:

```
ll is aliased to `ls -lah'
```

Limitations

- Aliases are expanded only in **interactive shells** by default.
- They cannot accept positional arguments (`$1`, `$2`, etc.).
- For more complex behaviour, use a **shell function** instead.

Example:

```
greet() {
    echo "Hello, $1"
}
```

Run:

```
greet Alice
```

Output:

```
Hello, Alice
```

Exit Status

- **0** – Alias command executed successfully.
- **Non-zero** – An error occurred (for example, invalid alias syntax).

Summary

Feature	Description
Command	`alias`
Type	Bash built-in
Purpose	Create shortcuts for commands
List aliases	`alias`
Show one alias	`alias name`
Create alias	`alias ll='ls -l'`
Remove alias	`unalias ll`
Remove all aliases	`unalias -a`
Permanent location	`~/.bashrc` (Bash)
Accepts arguments	No (use shell functions instead)

Quick Example

```
$ alias ll='ls -lah'
$ ll
```

Equivalent command:

```
ls -lah
```

The `alias` command is one of the simplest and most useful Bash features for improving productivity by replacing long or frequently used commands with short, memorable names.

4. history

The **`history`** command displays the list of commands that have been executed in the current shell session and those saved from previous sessions. It is one of the most useful commands for recalling, reusing, and auditing previously executed commands.

Syntax

```
history [options] [number]
```

- **options** – Modify the output or behavior.
- **number** – Display only the last *number* of commands.

What `history` does

The shell keeps a record of commands you execute. The `history` command lets you:

- View previously executed commands
- Re-run old commands
- Search for commands
- Delete history entries
- Save or clear command history

Basic Examples

1. Display command history

```
history
```

Example:

```
1  pwd
2  ls
3  cd /etc
4  cat passwd
5  history
```

2. Show the last 10 commands

```
history 10
```

Example:

```
96  df -h
97  free -h
98  ip addr
99  uname -r
100 history
```

---

3. Execute a command from history

Run command number 25:

```
!25
```

Example:

```
$ !25
ls -l /etc
```

---

4. Repeat the previous command

```
!!
```

Example:

```
sudo !!
```

If the previous command was:

```
apt update
```

then

```
sudo !!
```

becomes

```
sudo apt update
```

5. Repeat the last command beginning with a word

```
!systemctl
```

Runs the most recent command that starts with:

```
systemctl
```

---

6. Search history interactively

Press

```
Ctrl + R
```

Then type:

```
ssh
```

Example:

```
(reverse-i-search)`ssh': ssh user@server
```

Press **Enter** to execute it.

Common Options

Option	Description
`history`	Display all history
`history 20`	Show last 20 commands
`history -c`	Clear current history list
`history -d N`	Delete entry number N
`history -a`	Append current session to history file
`history -w`	Write history to the history file
`history -r`	Read history file and append it
`history -n`	Read new history lines not already read

Examples

Show last 5 commands

```
history 5
```

Delete command number 42

```
history -d 42
```

Clear history

```
history -c
```

To also overwrite the saved history file:

```
history -c
history -w
```

Save current history immediately

```
history -a
```

Reload history file

```
history -r
```

Where history is stored

For Bash, command history is usually saved in:

```
~/.bash_history
```

View it with:

```
cat ~/.bash_history
```

History Variables

Variable	Purpose
`HISTFILE`	History file location
`HISTSIZE`	Number of commands kept in memory
`HISTFILESIZE`	Maximum commands stored in the history file
`HISTCONTROL`	Controls which commands are saved
`HISTIGNORE`	Ignore commands matching specific patterns

Example:

```
echo $HISTSIZE
```

Example output:

```
1000
```

Useful Commands

Find commands containing a word

```
history | grep ssh
```

Execute the previous `ls` command

```
!ls
```

Display history with line numbers

```
history
```

Output:

```
101  pwd
102  ls
103  cd /tmp
104  vim file.txt
```

Save history before logging out

```
history -a
```

Exit Status

- **0** – Success
- **Non-zero** – Error (for example, an invalid option)

Practical Uses

- Recover previously used commands.
- Avoid retyping long commands.
- Audit commands executed in a shell.
- Troubleshoot by reviewing command history.
- Quickly repeat administrative tasks.

Tips

- Use **`Ctrl + R`** for fast interactive searching—it is often quicker than scrolling through `history`.
- Commands prefixed with a space may not be saved if `HISTCONTROL=ignorespace` is set.
- By default, Bash writes the history file when the shell exits. Use `history -a` to save the current session without logging out.

Summary

Feature	Description
Command	`history`
Purpose	Display and manage shell command history
History file	`~/.bash_history`
Interactive search	`Ctrl + R`
Repeat previous command	`!!`
Execute history entry	`!number`
Delete entry	`history -d N`
Clear history	`history -c`

The `history` command is an essential Bash built-in that helps you view, search, reuse, and manage previously executed commands, making command-line work faster and more efficient.

5. clear

The **`clear`** command is used to **clear the terminal screen**, removing all previously displayed text from view. It does **not** delete command history or stop running programs—it simply provides a clean terminal display.

Syntax

```
clear
```

How it works

When you run:

```
clear
```

the terminal is instructed to erase the visible contents of the screen and move the cursor to the top-left corner.

Example

Before:

```
$ ls
Documents  Downloads  Pictures

$ pwd
/home/bharath

$ date
Mon Jul 27 15:40:22 IST 2026
```

Run:

```
clear
```

After:

```
$
```

The previous output is no longer visible on the screen, although it may still be available in the terminal's scrollback buffer.

Common Uses

- Clean up a cluttered terminal.
- Start with a fresh screen before running new commands.
- Improve readability during demonstrations or presentations.
- Separate the output of different tasks.

Options

`clear` has very few commonly used options.

Option	Description
`-T <terminal>`	Specify the terminal type instead of using the `TERM` environment variable.
`-V`	Display the version information.
`-x`	Do not clear the terminal's scrollback buffer (behavior depends on the implementation).

Example:

```
clear -V
```

Related Keyboard Shortcut

Instead of typing `clear`, you can usually press:

```
Ctrl + L
```

This performs the same function in most shells, including Bash and Zsh.

Difference Between `clear` and `reset`

`clear`	`reset`
Clears only the visible screen.	Reinitializes the terminal.
Fast and lightweight.	Fixes a corrupted or garbled terminal.
Does not change terminal settings.	Restores terminal settings to their defaults.

Exit Status

- `0` – Command executed successfully.
- Non-zero – An error occurred (rare, such as an invalid terminal type).

Notes

- `clear` does **not** delete your shell command history (`history` command).
- It does **not** terminate running processes.
- In many terminal emulators, you can still scroll up to see previous output unless the scrollback buffer is also cleared.

Summary

- **Command:** `clear`
- **Purpose:** Clears the terminal screen.
- **Requires root privileges:** No
- **Affects files:** No
- **Common shortcut:** `Ctrl + L`
- **Typical use case:** Clean the terminal before continuing work.

6. env

The **env** command is used to **display, modify, or run a program with specific environment variables**. Environment variables are key-value pairs that store information used by the shell and programs, such as the user's home directory, current language, executable search path, and more.

Syntax

```
env [OPTION]... [NAME=VALUE]... [COMMAND [ARG]...]
```

Main Uses

1. Display all environment variables.
2. Run a command with modified environment variables.
3. Run a command with a completely clean environment.

---

1. Display Environment Variables

Simply running:

```
env
```

Example output:

```
USER=bharath
HOME=/home/bharath
PATH=/usr/local/bin:/usr/bin:/bin
SHELL=/bin/bash
LANG=en_US.UTF-8
PWD=/home/bharath
```

Common Environment Variables

Variable	Description
HOME	User's home directory
USER	Current username
PATH	Directories searched for executables
PWD	Current working directory
SHELL	Current shell
LANG	Language and locale
HOSTNAME	System hostname
TERM	Terminal type

2. Run a Command with a Temporary Variable

```
env NAME=value command
```

Example:

```
env GREETING=Hello bash -c 'echo $GREETING'
```

Output:

```
Hello
```

The variable exists **only for that command**.

---

3. Run with a Clean Environment

```
env -i command
```

Example:

```
env -i bash
```

This starts Bash with almost no environment variables.

Verify:

```
env -i env
```

Output:

(No variables are shown.)

4. Set Multiple Variables

```
env VAR1=value1 VAR2=value2 command
```

Example:

```
env NAME=John AGE=25 bash -c 'echo "$NAME is $AGE years old"'
```

Output:

```
John is 25 years old
```

---

5. Ignore Existing Variables

```
env -i HOME=/tmp bash
```

Only the specified variable (`HOME`) is available.

---

6. Find Executable Using PATH

```
env ls
```

This works because `env` searches for `ls` using the `PATH` environment variable before executing it.

Common Options

Option	Description
`-i`, `--ignore-environment`	Start with an empty environment
`-u VARIABLE`	Remove a variable before running the command
`--help`	Display help
`--version`	Show version information

Examples

Print only the PATH variable

```
env | grep PATH
```

Output:

```
PATH=/usr/local/bin:/usr/bin:/bin
```

Remove a Variable

```
env -u HOME bash -c 'echo $HOME'
```

Output:

The `HOME` variable is unavailable for that command.

Change PATH Temporarily

```
env PATH=/usr/bin ls
```

The `ls` command runs using the specified `PATH` value only.

Run Python with a Temporary Variable

```
env DEBUG=1 python3 app.py
```

The program receives:

```
DEBUG=1
```

Difference Between `env` and `printenv`

Feature	`env`	`printenv`
Display environment variables		
Run a command		✘
Set temporary variables		✘
Remove variables		✘

Difference Between `env` and `export`

`env`	`export`
Temporary for one command	Persistent in the current shell session
Doesn't modify the current shell	Modifies the current shell environment
Useful for testing	Useful for setting shell variables

Example:

```
env TEST=123 bash -c 'echo $TEST'
```

The variable disappears after the command finishes.

Whereas:

```
export TEST=123
echo $TEST
```

The variable remains available in the current shell until it is unset or the shell exits.

Exit Status

Exit Code	Meaning
`0`	Command executed successfully
`1`	Error occurred
`126`	Command found but not executable
`127`	Command not found

Real-World Uses

- View the current environment variables.
- Test applications with different environment settings.
- Run a command without inheriting the current shell's environment.
- Temporarily change variables like `PATH`, `HOME`, or `LANG`.
- Remove sensitive variables before executing a program.

Summary

- `env` displays the current environment variables.
- It can set temporary environment variables for a single command.
- It can remove variables or start a command with a completely clean environment.
- It is commonly used in shell scripting, software testing, and system administration to control a program's execution environment.

7. export

The **`export`** command is a **Bash built-in** used to make **shell variables available to child processes** (subshells and programs launched from the current shell). It is commonly used to set **environment variables**.

Syntax

```
export VARIABLE=value
```

or

```
export VARIABLE
```

or

```
export -p
```

What does `export` do?

Normally, variables created in a shell are **local to that shell**.

Example:

```
NAME="John"
```

Only the current shell knows about `NAME`.

If you run:

```
export NAME
```

or

```
export NAME="John"
```

then every program started from this shell can access `NAME`.

Basic Examples

1. Create and export a variable

```
export NAME="Bharath"
```

Check it:

```
echo $NAME
```

Output

```
Bharath
```

---

2. View exported variables

```
export -p
```

Example output

```
declare -x HOME="/home/user"
declare -x LANG="en_US.UTF-8"
declare -x PATH="/usr/local/bin:/usr/bin"
```

---

3. Export an existing variable

```
VERSION=2.5
export VERSION
```

Equivalent to

```
export VERSION=2.5
```

Why is `export` needed?

Suppose you create a variable.

```
MYVAR="Hello"
```

Now run another shell:

```
bash
```

Inside the new shell:

```
echo $MYVAR
```

Output

```
(empty)
```

Because the variable wasn't exported.

Now exit and try:

```
export MYVAR="Hello"
bash
echo $MYVAR
```

Output

```
Hello
```

The variable is inherited by the child shell.

Common Environment Variables

PATH

```
export PATH=$PATH:/opt/myprogram/bin
```

Adds a directory to the executable search path.

JAVA_HOME

```
export JAVA_HOME=/usr/lib/jvm/java-21-openjdk
```

Applications can use:

```
echo $JAVA_HOME
```

EDITOR

```
export EDITOR=vim
```

Many programs use this as the default text editor.

LANG

```
export LANG=en_US.UTF-8
```

Sets the system locale for programs launched from the shell.

Useful Options

`export -p`

Displays all exported variables.

```
export -p
```

`export -n`

Removes the export attribute from a variable.

```
export -n NAME
```

The variable still exists in the current shell but will no longer be passed to newly started child processes.

`export -f`

Exports a shell function.

Example:

```
hello() {
    echo "Hello"
}

export -f hello
```

A child Bash shell can then use the `hello` function.

Practical Examples

Example 1: Temporary environment variable

```
export DEBUG=1
./myprogram
```

The program can read the `DEBUG` variable.

Example 2: Set Python module path

```
export PYTHONPATH=/home/user/project
```

Python searches this directory for modules.

Example 3: Add a custom command directory

```
export PATH=$PATH:$HOME/bin
```

Now executables in `~/bin` can be run without specifying their full path.

Example 4: Set a proxy

```
export HTTP_PROXY=http://proxy.example.com:8080
```

Some command-line tools will use this proxy automatically.

Make an Export Permanent

Using `export` in the terminal only affects the current session.
To make it permanent, add it to your shell's startup file:

```
nano ~/.bashrc
```

Add:

```
export PATH=$PATH:$HOME/bin
export JAVA_HOME=/usr/lib/jvm/java-21-openjdk
```

Reload the configuration:

```
source ~/.bashrc
```

Difference Between Variable and Exported Variable

Variable	Exported Variable
Available only in the current shell	Available in the current shell and child processes
`NAME=John`	`export NAME=John`
Not inherited by child shells	Inherited by child shells
Used internally by the shell	Used by programs and child shells

Exit Status

- `0` – Export successful.
- Non-zero – Invalid option or invalid identifier.

Example:

```
export 123NAME=value
```

Output:

```
bash: export: `123NAME=value': not a valid identifier
```

Common Use Cases

- Setting environment variables.
- Configuring application settings.
- Modifying the `PATH`.
- Defining `JAVA_HOME`, `PYTHONPATH`, or `LD_LIBRARY_PATH`.
- Passing configuration values to scripts and programs.

Summary

- `export` is a **Bash built-in** command.
- It marks variables (or functions) to be inherited by child processes.
- Without `export`, variables remain local to the current shell.
- It is most commonly used to configure the environment for programs and scripts.
- Changes made with `export` are temporary unless added to a shell startup file such as `~/.bashrc`.

8. source

What is `source`?

`source` is a **shell built-in command** used to read and execute commands from a file **in the current shell environment**.

It is mainly used to:

- Apply changes made in shell configuration files
- Load environment variables
- Run scripts without creating a new shell process

The command is also written as:

```
. filename
```

The dot (`.`) is an alternative form of `source`.

Syntax

```
source filename
```

or

```
. filename
```

Example 1: Loading environment variables

Create a file:

```
nano myvars.sh
```

Add:

```
export NAME="Bharath"
export CITY="Chennai"
```

Now load it:

```
source myvars.sh
```

Check:

```
echo $NAME
```

Output:

```
Bharath
```

The variables are now available in your current terminal.

Example 2: Reloading `.bashrc`

When you modify your Bash configuration:

```
nano ~/.bashrc
```

Example addition:

```
alias ll='ls -l'
```

Normally, the change applies only after opening a new terminal.

Instead, run:

```
source ~/.bashrc
```

Now the new alias works immediately:

```
ll
```

Example 3: Running a script in the current shell

Create:

```
test.sh
```

Content:

```
VAR="Hello"
```

Run normally:

```
./test.sh
```

Check:

```
echo $VAR
```

Output:

```
(empty)
```

Because the script ran in a **separate shell**.

Now run:

```
source test.sh
```

Check:

```
echo $VAR
```

Output:

```
Hello
```

Because `source` executes it inside the **current shell**.

Difference between `source` and executing a script

Command	Shell used	Changes remain?
`./script.sh`	New shell	No
`bash script.sh`	New Bash process	No
`source script.sh`	Current shell	Yes

Common uses

1. Reload Bash configuration

```
source ~/.bashrc
```

2. Activate Python virtual environment

```
source venv/bin/activate
```

3. Load application settings

```
source config.env
```

Check if `source` is a built-in command

```
type source
```

Output:

```
source is a shell builtin
```

Important points

- `source` does **not create a new process**.
- It modifies the **current shell session**.
- It is useful for environment variables, aliases, functions, and shell settings.
- If a sourced script contains `exit`, it can close your current terminal session.

Simple definition

`source` executes a shell script inside the current shell so that variables, aliases, and functions defined in that script remain available.