Bash 101

Jordan Cleal

Bash 101

Bash is the default shell on most Linux systems and has been the standard on macOS for years. It is the glue that holds together everything from automated deployment pipelines to simple daily scripts. Learning Bash means learning to automate the repetitive, chain tools together, and understand what is actually happening when you run a command.

What you will need: a terminal. On macOS and Linux, Bash (or a compatible shell) is already installed. On Windows, install WSL2 and use the Ubuntu terminal - everything in this series will work the same way.

What this series covers:

- Part 1 - The basics: what a shell is, your first script, variables, reading user input, arithmetic, and conditional logic
- Part 2 - Setup: configuring your shell environment, setting Bash as your default, understanding `.bashrc` vs `.bash_profile`, managing your `$PATH`, and writing aliases

What you will be able to do by the end:

Write shell scripts that automate real tasks - moving files, checking conditions, prompting for input, and making your terminal environment work the way you want it to. You will understand what a script is doing when you read it, and you will be comfortable writing your own from scratch.

Select a section from the sidebar to get started.

Why Bash

Bash is the universal glue of automation. It is on every Linux system, runs in every CI pipeline, and is the default shell in almost every Docker image. You cannot avoid it - so you should understand it.

Bash is a good fit for:

- Wrapping CLI tools (`git`, `docker`, `aws`, `kubectl`)
- CI pipeline steps
- Bootstrap and setup scripts
- File manipulation, renaming, moving
- Anything that is mostly gluing commands together

Reach for Python or Go instead when you have:

- Complex data structures
- HTTP requests or JSON parsing
- Scripts over ~200 lines
- Anything that needs proper error types
- Anything another team has to maintain

The rule: if you are fighting Bash to express an idea, it is telling you to use something else.

The script that breaks CI at 2am

```
#!/bin/bash
# deploy.sh - the one that looked fine for 6 months

ENV=$1
aws s3 sync ./dist s3://my-bucket-$ENV
aws cloudfront create-invalidation --distribution-id $CF_ID --paths "/*"
echo "Done"
```

What happens when `ENV` is empty? When `aws s3 sync` fails? When `$CF_ID` is unset?

This script silently succeeds even when the deploy failed. It exits 0. CI shows green. Production is broken.

Every one of these problems has the same fix - and it is mostly one line at the top.

Start every script with these three things

```bash
#!/usr/bin/env bash
set -euo pipefail

# Your script here
```

- **`set -e`** - exit immediately on any command that returns non-zero. No more "the deploy failed but the script printed Done".
- **`set -u`** - treat unset variables as errors. `$UNDEFINED` becomes a fatal error, not an empty string.
- **`set -o pipefail`** - a pipeline fails if *any* command in it fails. Without this, `false | true` exits 0.

These three flags together turn Bash from "silently wrong" to "loudly correct". Almost every Bash incident comes from one of these being absent.

`#!/usr/bin/env bash` not `#!/bin/bash`

```bash
#!/bin/bash
# Hardcoded path - works on Linux,
# breaks on macOS with Homebrew bash,
# breaks in Docker images where bash lives somewhere else
```

```bash
#!/usr/bin/env bash
# Finds the first 'bash' on $PATH
# Works everywhere - local, CI, Docker
# Picks up Homebrew bash on macOS
```

Use `#!/usr/bin/env bash` by default. Only use `#!/bin/bash` when you specifically need a known version at a known path.

Always quote your variables

```bash
# Without quotes: word-splitting and glob expansion
FILE="my file.txt"
cp $FILE /tmp/       # → cp my file.txt /tmp/  (two args, breaks)
cp "$FILE" /tmp/     # → cp "my file.txt" /tmp/ (one arg, correct)

# Double-quote every variable expansion
echo "$HOME"
ls "$DIR"
rm -rf "$TMPDIR"
```

Unquoted variables with spaces silently split into multiple words. Files with spaces break scripts that look fine in testing.

Exception: `[[ ]]` and `$(( ))` do not require quoting inside them. Everything else: quote it.

Arguments and defaults

```bash
#!/usr/bin/env bash
set -euo pipefail

# Positional arguments with defaults
ENV="${1:-}"                    # empty default - validate below
REGION="${2:-ap-southeast-2}"   # real default
```

```bash
# Validate required args
if [[ -z "$ENV" ]]; then
  echo "Usage: $0 <environment> [region]" >&2
  exit 1
fi

# One-liner guard - fail with a clear message if unset or empty
: "${AWS_PROFILE:?AWS_PROFILE must be set}"
: "${IMAGE_TAG:?IMAGE_TAG must be set}"
```

`${VAR:?message}` is one line per required variable and gives a clear error. Better than a wall of `if`-then-`fi` blocks.

Variables and scope

```bash
# Convention: UPPER_CASE for env vars / globals, lower_case for locals
AWS_PROFILE="my-profile"
readonly SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

function deploy() {
  local env="$1"            # local scope - lowercase
  local region="${2:-ap-southeast-2}"
  local timestamp
  timestamp="$(date +%Y%m%d-%H%M%S)"

  echo "Deploying to $env/$region at $timestamp"
}
```

`local` prevents function variables leaking into global scope. Always use it inside functions - without it, every variable is global.

`readonly` makes a variable immutable. Use it for constants and computed paths that should never change after assignment.

Conditionals with `[[ ]]`

```bash
# Use [[ ]] not [ ] - it's safer and more featureful
if [[ "$ENV" == "prod" ]]; then
  echo "deploying to production"
fi

# String tests
[[ -z "$VAR" ]]       # true if empty
[[ -n "$VAR" ]]       # true if non-empty
[[ "$A" == "$B" ]]    # string equality

# File tests
[[ -f "$FILE" ]]      # exists and is a regular file
[[ -d "$DIR" ]]       # exists and is a directory
[[ -x "$BINARY" ]]    # exists and is executable

# Combining conditions
if [[ -f "$CONFIG" && -r "$CONFIG" ]]; then
  source "$CONFIG"
fi
```

`[[ ]]` does not word-split or glob-expand, handles empty variables gracefully, and supports `==` pattern matching and `=~` regex.

Functions: structure your script

```
#!/usr/bin/env bash
set -euo pipefail

#   helpers
log()  { echo "[$(date +%H:%M:%S)] $*"; }
die()  { echo "ERROR: $*" >&2; exit 1; }
info() { echo "  → $*"; }

#   functions
check_deps() {
  local deps=("aws" "jq" "curl")
  for dep in "${deps[@]}"; do
    command -v "$dep" &>/dev/null || die "required tool not found: $dep"
  done
}

deploy() {
  local env="$1"
  log "Starting deploy to $env"
  # ...
}

#   main
main() {
  check_deps
  deploy "${1:?usage: $0 <env>}"
}

main "$@"
```

The `log/die/info` helpers are the first thing to add to any non-trivial script. The `main() + main "$@"` pattern makes the script importable - you can `source` it without executing it.

Traps and cleanup

```
#!/usr/bin/env bash
set -euo pipefail

TMPDIR_WORK="$(mktemp -d)"

cleanup() {
  rm -rf "$TMPDIR_WORK"
  log "cleaned up $TMPDIR_WORK"
}
trap cleanup EXIT

on_error() {
  local exit_code=$?
  local line_number=${BASH_LINENO[0]}
  echo "ERROR: command failed with exit $exit_code at line $line_number" >&2
}
trap on_error ERR
```

`trap cleanup EXIT` is Bash's equivalent of `defer` - it always runs, even if the script exits early due to `set -e`. Temp files are always cleaned up.

The production-ready template

```
#!/usr/bin/env bash
set -euo pipefail
```

```bash
readonly SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

log()  { echo "[$(date +%H:%M:%S)] $*"; }
die()  { echo "ERROR: $*" >&2; exit 1; }

cleanup() { : ; }
trap cleanup EXIT

check_deps() {
  for dep in "$@"; do
    command -v "$dep" &>/dev/null || die "missing: $dep"
  done
}

main() {
  local env="${1:?usage: $(basename "$0") <env>}"
  check_deps aws jq
  : "${AWS_PROFILE:?AWS_PROFILE must be set}"
  log "deploying to $env"
  # ... actual work
}

main "$@"
```

This template gives you safe defaults, cleanup on exit, dependency checks, validated args, and clear errors. Everything that turned the 2am incident into a silent success is now a loud failure.

Exercises: build one real script, step by step

Each exercise adds to the same file. By the end you have a production-safe script you can commit. Use a real automation task - not a toy example. A deploy step, a repo setup script, a cleanup job.

Exercise 1 - Safe skeleton (~5 min)

Create a new file. Pick a name that describes what it will do (`deploy.sh`, `setup.sh`, `cleanup.sh`).

```bash
#!/usr/bin/env bash
set -euo pipefail

log()  { echo "[$(date +%H:%M:%S)] $*"; }
die()  { echo "ERROR: $*" >&2; exit 1; }

main() {
  log "starting"
  echo "hello from $(basename "$0")"
  log "done"
}

main "$@"
```

```
chmod +x your-script.sh
./your-script.sh
```

Test `set -u`: add `echo "$UNDEFINED_VAR"` after `log "starting"`. Run it - it should fail with `unbound variable`, not print an empty line. Remove it after.

Test `set -e`: add `false` and confirm the script aborts before printing "done". Remove it after.

Exercise 2 - Arguments & validation (~8 min)

Keep everything from Exercise 1. Add argument handling to `main()`:

```bash
main() {
  # Required argument - fails with usage message if missing
  local env="${1:?usage: $(basename "$0") <environment> [region]}"
  # Optional argument with default
  local region="${2:-ap-southeast-2}"

  # Validate the value, not just presence
  case "$env" in
    dev|staging|prod) ;;
    *) die "unknown environment: $env (expected: dev|staging|prod)" ;;
  esac

  # Guard: refuse to run in prod without explicit confirmation
  if [[ "$env" == "prod" ]]; then
    read -r -p "Deploying to PROD - are you sure? [y/N] " confirm
    [[ "$confirm" == "y" ]] || die "aborted"
  fi

  log "targeting $env in $region"
}
```

Verify:

```bash
./your-script.sh
# your-script.sh: 1: usage: your-script.sh <environment> [region]

./your-script.sh badenv
# ERROR: unknown environment: badenv (expected: dev|staging|prod)

./your-script.sh dev
# [10:31:00] targeting dev in ap-southeast-2

./your-script.sh prod
# Deploying to PROD - are you sure? [y/N] n
# ERROR: aborted
```

Exercise 3 - Functions & structure (~8 min)

Keep everything from Exercise 2. Add a dependency check and extract real work into functions:

```bash
# Add above main()
check_deps() {
  local deps=("$@")
  for dep in "${deps[@]}"; do
    command -v "$dep" &>/dev/null \
      || die "required tool not found: $dep - please install it"
  done
}

do_the_thing() {
  local env="$1"
  local region="$2"
  log "doing the thing in $env/$region"
  # your real work here
}

main() {
  local env="${1:?usage: $(basename "$0") <environment> [region]}"
  local region="${2:-ap-southeast-2}"

  check_deps aws jq   # ← tools your script actually uses
  do_the_thing "$env" "$region"
}
```

Replace `aws jq` with whatever tools your script actually calls. No fake deps.

Exercise 4 - Cleanup & error handling (~7 min)

Keep everything from Exercise 3. Add a trap for cleanup and error context:

```
# Add near the top, after set -euo pipefail
TMPDIR_WORK=""

cleanup() {
  if [[ -n "$TMPDIR_WORK" ]]; then
    rm -rf "$TMPDIR_WORK"
    log "cleaned up $TMPDIR_WORK"
  fi
}
trap cleanup EXIT

on_error() {
  local line="${BASH_LINENO[0]}"
  echo "ERROR: script failed at line $line" >&2
}
trap on_error ERR

# In main() or do_the_thing(), create and use a temp dir:
TMPDIR_WORK="$(mktemp -d)"
log "working in $TMPDIR_WORK"
```

`trap cleanup EXIT` fires on every exit - success, failure, or Ctrl-C. Temp files are always cleaned up.

Verify:

```
./your-script.sh dev
# [10:33:00] working in /tmp/tmp.abc123
# [10:33:00] doing the thing in dev/ap-southeast-2
# [10:33:00] cleaned up /tmp/tmp.abc123

# Add 'false' inside do_the_thing(), run again:
# [10:33:01] working in /tmp/tmp.xyz456
# ERROR: script failed at line 42
# [10:33:01] cleaned up /tmp/tmp.xyz456  ← trap still ran
```

Exercise 5 - Wire in real work (~7 min)

Replace the placeholder `do_the_thing` with the actual commands your script needs to run.

If you are writing a deploy script:

```
deploy() {
  local env="$1"
  : "${AWS_PROFILE:?AWS_PROFILE must be set}"
  log "syncing to s3..."
  aws s3 sync ./dist "s3://my-bucket-$env/" --delete
  log "invalidating CDN..."
  aws cloudfront create-invalidation \
    --distribution-id "$CF_ID" \
    --paths "/*"
}
```

If you are writing a setup script:

```
setup() {
  local target_dir="$1"
  log "creating directory structure..."
  mkdir -p "$target_dir"/{src,tests,docs}
```

```
  log "copying config templates..."
  cp -r "$SCRIPT_DIR/templates/." "$target_dir/"
  log "initialising git..."
  git -C "$target_dir" init -q
}
```

Do not copy these verbatim - use your actual task. The point is to leave with something real.

Final check:

```
./your-script.sh              # → usage error
./your-script.sh badenv       # → validation error
./your-script.sh dev          # → runs, cleans up
./your-script.sh dev us-west-2  # → runs with override

head -3 your-script.sh
# #!/usr/bin/env bash
# set -euo pipefail

wc -l your-script.sh    # aim for under 150 lines
```

What you built

- `set -euo pipefail` - fails loudly, never silently
- Validated args - clear usage errors, not mysterious crashes
- Dependency checks - tells you what is missing before it fails
- Functions - named, local-scoped, testable chunks
- Trap cleanup - temp files gone, always
- `main "$@"` - importable, not side-effectful on source

References: shellcheck.net · Google Shell Style Guide