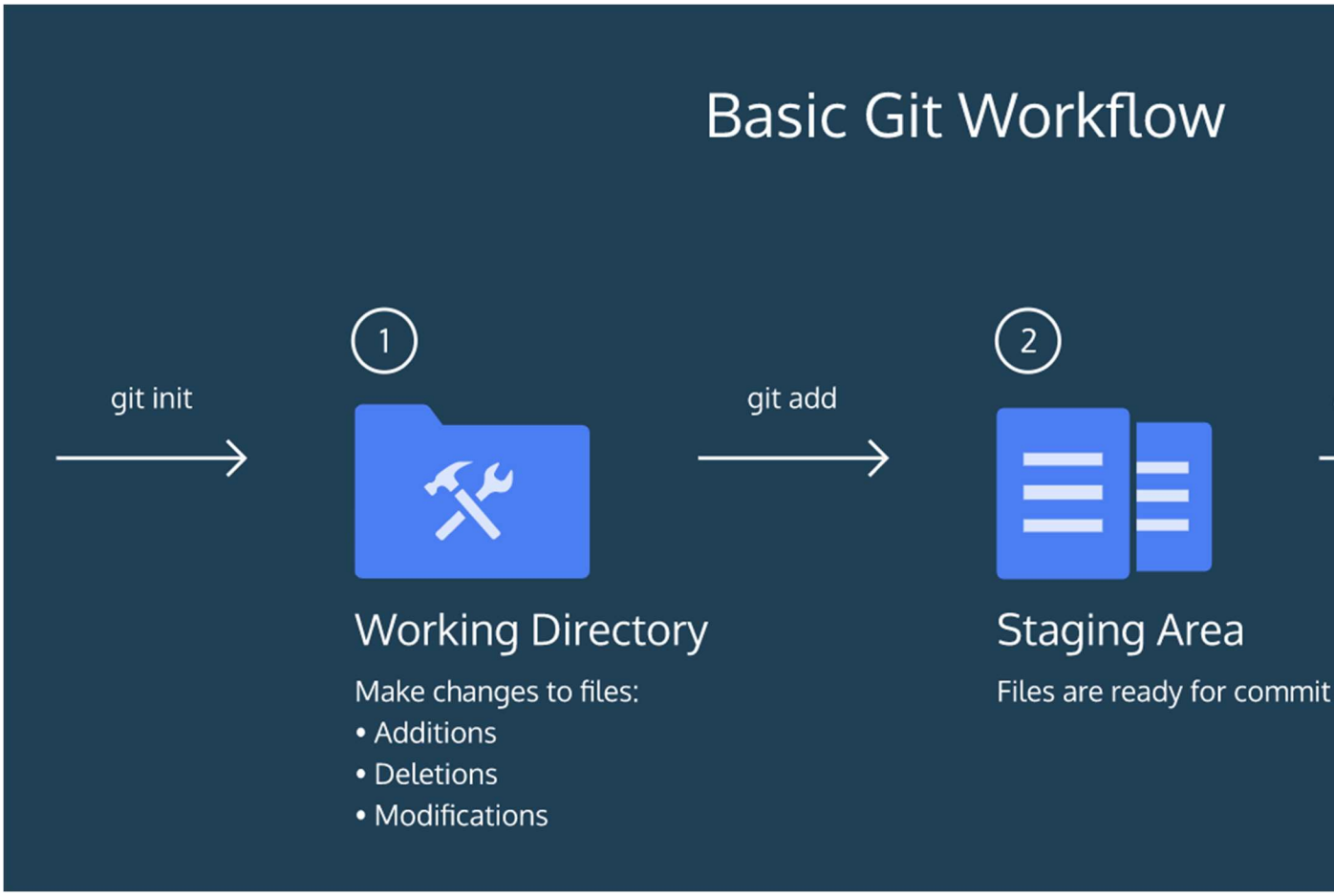


- `git init` creates a new Git repository
- `git status` inspects the contents of the working directory and staging area
- `git add` adds files from the working directory to the staging area
- `git diff` shows the difference between the working directory and the staging area
- `git commit` permanently stores file changes from the staging area in the repository
- `git log` shows a list of all previous commits

git init

The word `init` means *initialize*. The command sets up all the tools Git needs to begin tracking changes made to the project.

A Git project can be thought of as having three parts:

1. A *Working Directory*: where you'll be doing all the work: creating, editing, deleting and organizing files
2. A *Staging Area*: where you'll list changes you make to the working directory
3. A *Repository*: where Git permanently stores those changes as different *versions* of the project

The Git workflow consists of editing files in the working directory, adding files to the staging area, and saving changes to a Git repository. In Git, we save changes with a *commit*, which we will learn more about in this lesson.

git status

As you write the screenplay, you will be changing the contents of the working directory. You can check the status of those changes with:

```
git status
```

git add

In order for Git to start tracking **scene-1.txt**, the file needs to be added to the staging area.

We can add a file to the staging area with:

```
git add filename
```

git diff

Imagine that we type another line in **scene-1.txt**. Since the file is tracked, we can check the differences between the working directory and the staging area with:

```
git diff filename
```

Here, `filename` is the actual name of the file. If the name of my file was **changes.txt** the command would be

```
git diff changes.txt
```

git commit

A *commit* is the last step in our Git workflow. A commit permanently stores changes from the staging area inside the repository.

`git commit` is the command we'll do next. However, one more bit of code is needed for a commit: the *option* `-m` followed by a message.

git log

Often with Git, you'll need to refer back to an earlier version of a project. Commits are stored chronologically in the repository and can be viewed with:

```
git log
```

- A 40-character code, called a *SHA*, that uniquely identifies the commit. This appears in orange text.
- The commit author (you!)
- The date and time of the commit
- The commit message
- `git checkout HEAD filename`: Discards changes in the working directory.

- `git reset HEAD filename`: Unstages file changes in the staging area.
- `git reset commit_SHA`: Resets to a previous commit in your commit history.

Additionally, you learned a way to add multiple files to the staging area with a single command:

```
git add filename_1 filename_2
```

Backtracking Intro

When working on a Git project, sometimes we make changes that we want to get rid of. Git offers a few eraser-like features that allow us to undo mistakes during project creation. In this lesson, we'll learn some of these features.

head commit

In Git, the commit you are currently on is known as the `HEAD` commit. In many cases, the most recently made commit is the `HEAD` commit.

To see the `HEAD` commit, enter:

```
git show HEAD
```

The output of this command will display everything the git log command displays for the `HEAD` commit, plus all the file changes that were committed.

git checkout

What if you decide to change the ghost's line in the working directory, but then decide you wanted to discard that change?

You could rewrite the line how it was originally, but what if you forgot the exact wording? The command

```
git checkout HEAD filename
```

will restore the file in your working directory to look exactly as it did when you last made a commit.

```
git reset HEAD filename
```

This command *resets* the file in the staging area to be the same as the `HEAD`commit. It does not discard file changes from the working directory, it just removes them from the staging area.

git reset II

Creating a project is like hiking in a forest. Sometimes you take a wrong turn and find yourself lost.

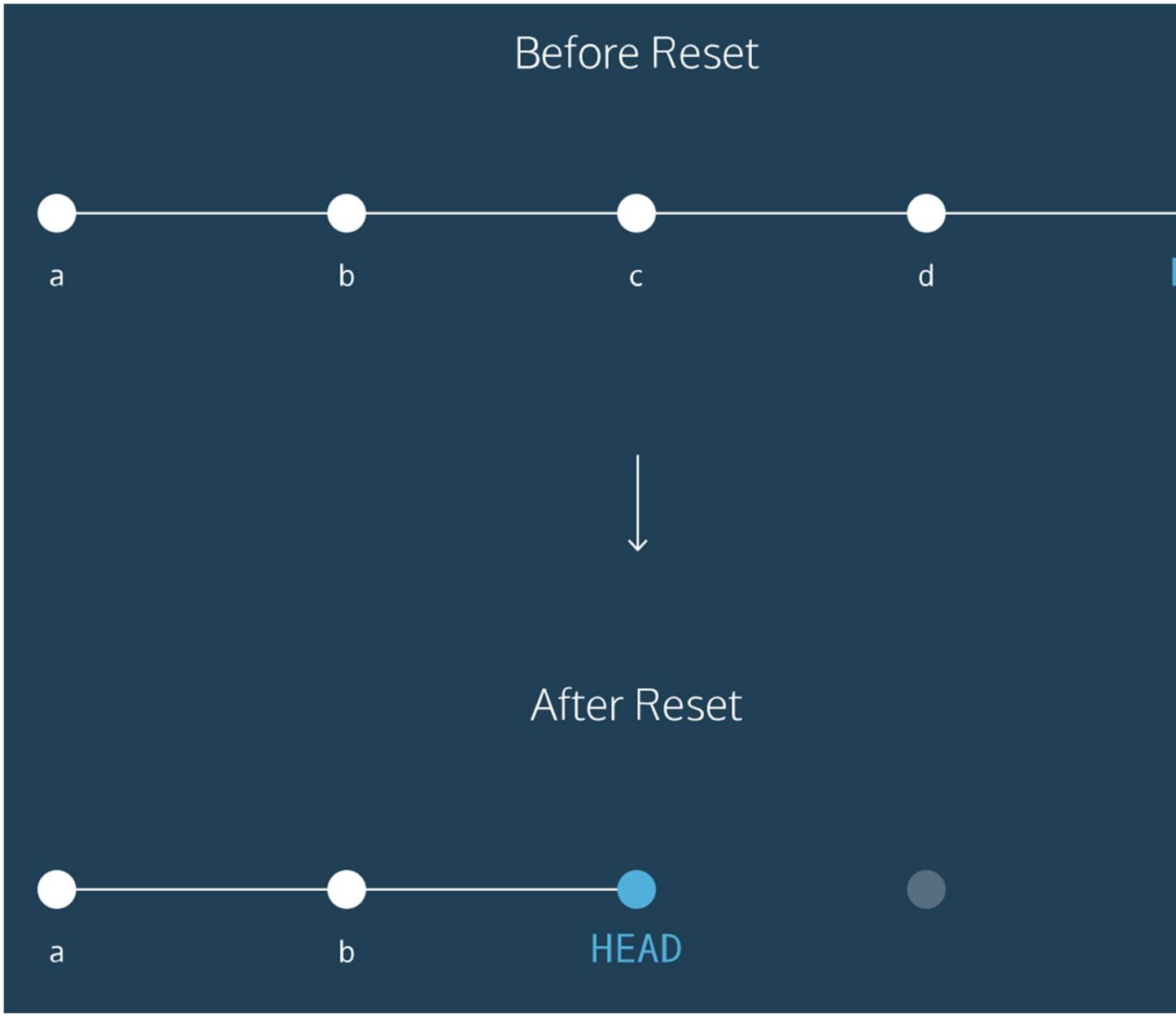
Just like retracing your steps on that hike, Git enables you to rewind to the part before you made the wrong turn. You can do this with:

```
git reset commit_SHA
```

This command works by using the first 7 characters of the SHA of a previous commit. For example, if the SHA of the previous commit is `5d692065cf51a2f50ea8e7b19b5a7ae512f633ba`, use:

```
git reset 5d69206
```

`HEAD` is now set to that previous commit.



Before reset:

- `HEAD` is at the *most recent commit*

After resetting:

- `HEAD` goes to a *previously made commit* of your choice
- The gray commits are no longer part of your project

- You have in essence rewound the project's history