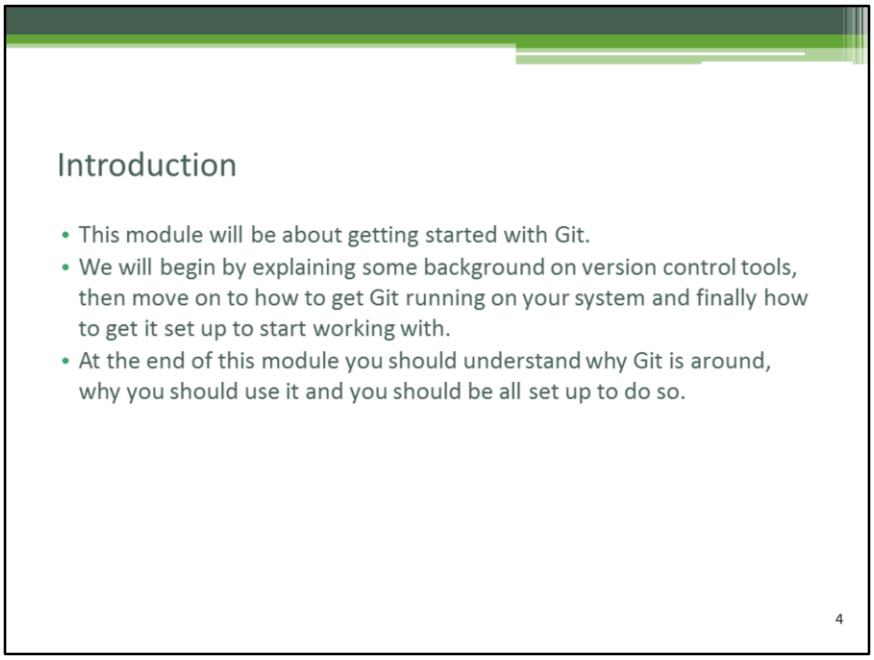
GIT DEVELOPER

Working with GIT

Your gateway into GIT

Table of Contents

Module 1: Getting Started



- How presentation will benefit audience: Adult learners are more interested in a subject if they know how or why it is important to them.
- Presenter's level of expertise in the subject: Briefly state your credentials in this area, or explain why participants should listen to you.

About Version Control

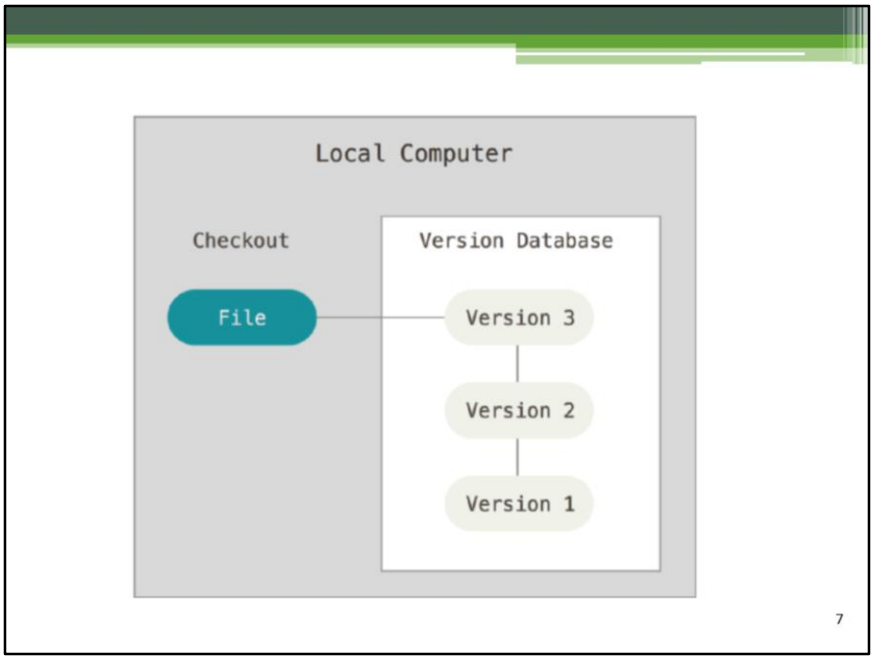
- What is “version control”, and why should you care?
- Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.
- If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use.
- It allows you to revert files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more.
- Using a VCS also generally means that if you screw things up or lose files, you can easily recover.
- In addition, you get all this for very little overhead.

5

Local Version Control Systems

- Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever).
- This approach is very common because it is so simple, but it is also incredibly error prone.
- It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

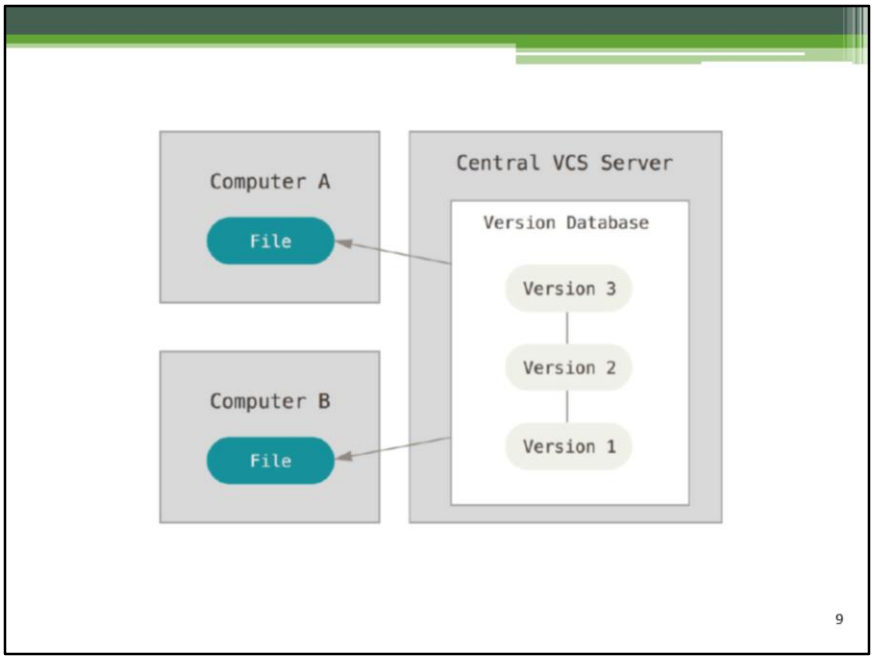
6


Local Computer
Checkout
File
Version Database
Version 3
Version 2
Version 1
7

Centralized Version Control Systems

- The next major issue that people encounter is that they need to collaborate with developers on other systems.
- To deal with this problem, Centralized Version
- Control Systems (CVCSs) were developed. These systems, such as CVS,
- Subversion, and Perforce, have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For
- many years, this has been the standard for version control.

8


Computer A
File
Computer B
File
Central VCS Server
Version Database
Version 3
Version 2
Version 1
9

Centralized Version Control Systems

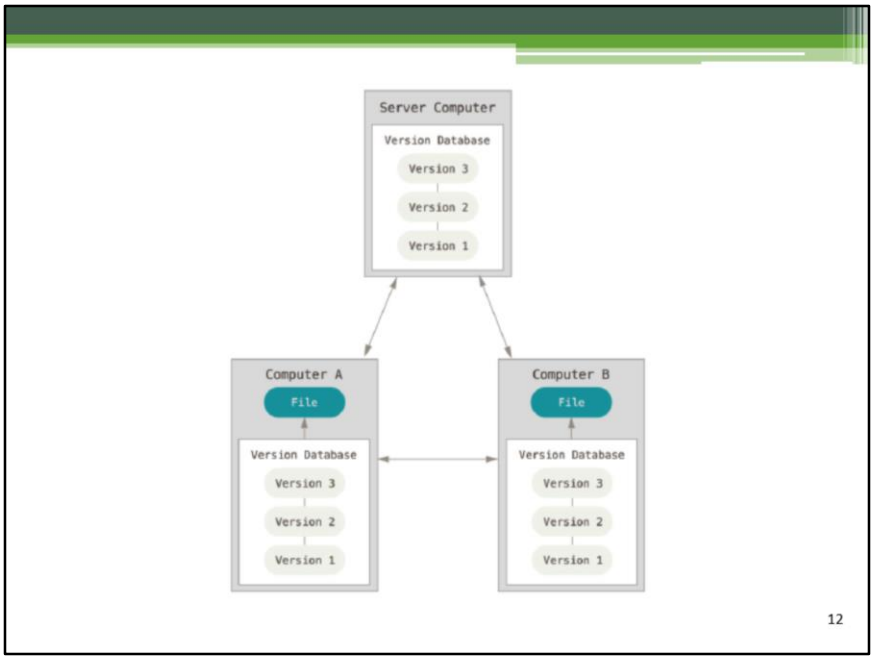
- This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing.
- Administrators have fine-grained control over who can do what; and it's far easier to administer a CVCS than it is to deal with local databases on every client.
- However, this setup also has some serious downsides. The most obvious is the single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they're working on.
- If the hard disk the central database is on becomes corrupted, and proper backups haven't been kept, you lose absolutely everything – the entire history of the project except whatever single snapshots people happen to have on their local machines.
- Local VCS systems suffer from this same problem – whenever you have the entire history of the project in a single place, you risk losing everything.

10

Distributed Version Control Systems

- This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files: they fully mirror the repository.
- Thus if any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it.
- Every clone is really a full backup of all the data.

11


Server Computer
Version Database
Version 3
Version 2
Version 1
Computer A
File
Version Database
Version 3
Version 2
Version 1
Computer B
File
Version Database
Version 3
Version 2
Version 1
12

A Short History of Git

- For most of the lifetime of the Linux kernel maintenance (1991–2002), changes to the software were passed around as patches and archived files.
- In 2002, the Linux kernel project began using a proprietary DVCS called BitKeeper.
- In 2005, the relationship between the community that developed the Linux kernel and the commercial company that developed BitKeeper broke down, and the tool's free-of-charge status was revoked.
- This prompted the Linux development community (and in particular Linus Torvalds, the creator of Linux) to develop their own tool based on some of the lessons they learned while using BitKeeper.

13

A Short History of Git

- Since its birth in 2005, Git has evolved and matured to be easy to use and yet retain these initial qualities.
- It's incredibly fast, it's very efficient with large projects, and it has an incredible branching system for non-linear development.

Git Basics

- So, what is Git in a nutshell? This is an important section to absorb, because if you understand what Git is and the fundamentals of how it works, then using Git effectively will probably be much easier for you.
- As you learn Git, try to clear your mind of the things you may know about other VCSs, such as Subversion and Perforce; doing so will help you avoid subtle confusion when using the tool.
- Git stores and thinks about information much differently than these other systems, even though the user interface is fairly similar, and understanding those differences will help prevent you from becoming confused while using it.

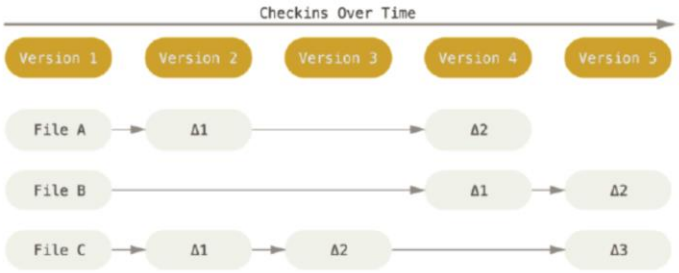
15

Snapshots, Not Differences

- The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data.
- Conceptually, most other systems store information as a list of file-based changes.
- These systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they keep as a set of files and the changes made to each file over time.

16

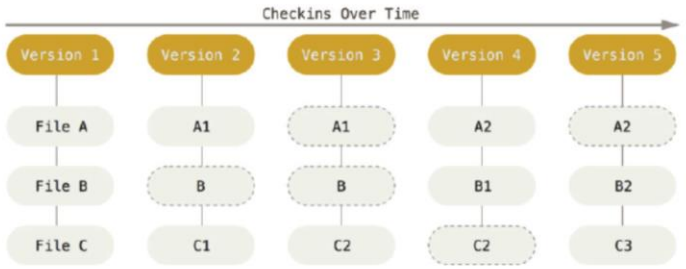
Storing data as changes to a base version of each file.



Snapshots, Not Differences

- Git doesn't think of or store its data this way. Instead, Git thinks of its data more like a set of snapshots of a miniature file system.
- Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot.
- To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored.
- Git thinks about its data more like a **stream of snapshots**.

Storing data as snapshots of the project over time.



19

Snapshots, Not Di erences

- This is an important distinction between Git and nearly all other VCSs.
- It makes Git reconsider almost every aspect of version control that most other systems copied from the previous generation.
- This makes Git more like a mini file system with some incredibly powerful tools built on top of it, rather than simply a VCS.

Nearly Every Operation Is Local

- Most operations in Git only need local files and resources to operate – generally no information is needed from another computer on your network.
- If you're used to a CVCS where most operations have that network latency overhead, this aspect of Git will make you think that the gods of speed have blessed Git with unworldly powers.
- Because you have the entire history of the project right there on your local disk, most operations seem almost instantaneous.

21

Nearly Every Operation Is Local

- For example, to browse the history of the project, Git doesn't need to go out to the server to get the history and display it for you – it simply reads it directly from your local database.
- This means you see the project history almost instantly.
- If you want to see the changes introduced between the current version of a file and the file a month ago, Git can look up the file a month ago and do a local difference calculation, instead of having to either ask a remote server to do it or pull an older version of the file from the remote server to do it locally.

Git Has Integrity

- Everything in Git is check-summed before it is stored and is then referred to by that checksum.
- This means it's impossible to change the contents of any file or directory without Git knowing about it.
- This functionality is built into Git at the lowest levels and is integral to its philosophy. You can't lose information in transit or get file corruption without Git being able to detect it.

Git Has Integrity

- The mechanism that Git uses for this checksumming is called a SHA-1 hash.
- This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git.
- A SHA-1 hash looks something like this:

  24b9da6552252987aa493b52f8696cd6d3b00373

- You will see these hash values all over the place in Git because it uses them so much.
- In fact, Git stores everything in its database not by file name but by the hash value of its contents.

Git Generally Only Adds Data

- When you do actions in Git, nearly all of them only add data to the Git database.
- It is hard to get the system to do anything that is not undoable or to make it erase data in any way.
- As in any VCS, you can lose or mess up changes you haven't committed yet; but after you commit a snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository.

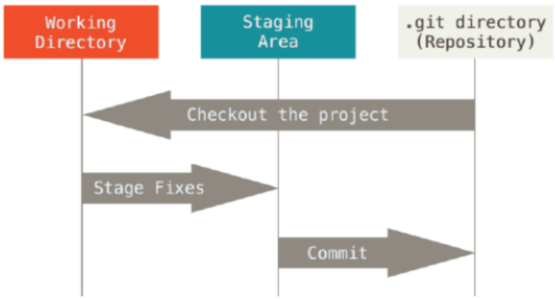
25

The Three States

- This is the main thing to remember about Git if you want the rest of your learning process to go smoothly.
- Git has three main states that your files can reside in: committed, modified, and staged.
- Committed means that the data is safely stored in your local database.
- Modified means that you have changed the file but have not committed it to your database yet.
- Staged means that you have marked a modified file in its current version to go into your next commit snapshot.
- This leads us to the three main sections of a Git project: the Git directory, the working directory, and the staging area.

26

Working directory, staging area, and Git directory.



27

The Three States

- The Git directory is where Git stores the metadata and object database for your project.
- This is the most important part of Git, and it is what is copied when you clone a repository from another computer.
- The working directory is a single checkout of one version of the project.
- These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.
- The staging area is a file, generally contained in your Git directory, that
- stores information about what will go into your next commit.
- It's sometimes referred to as the "index", but it's also common to refer to it as the staging area.

28

The Three States

- The basic Git workflow goes something like this:
  1. You modify files in your working directory.
  2. You stage the files, adding snapshots of them to your staging area.
  3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

- If a particular version of a file is in the Git directory, it's considered committed.
- If it has been modified and was added to the staging area, it is staged.
- And if it was changed since it was checked out but has not been staged, it is modified.

29

The Command Line

- There are a lot of different ways to use Git.
- There are the original command line tools, and there are many graphical user interfaces of varying capabilities.
- For this book, we will be using Git on the command line. For one, the command line is the only place you can run **all** Git commands – most of the GUIs only implement some subset of Git functionality for simplicity.
- If you know how to run the command line version, you can probably also figure out how to run the GUI version, while the opposite is not necessarily true.
- Also, while your choice of graphical client is a matter of personal taste, all users will have the command-line tools installed and available.

30

Installing Git

- If you want to install the basic Git tools on Linux via a binary installer, you can generally do so through the basic package-management tool that comes with your distribution.
- If you're on Fedora for example, you can use yum:

```
$ sudo yum install git-all
```

- If you're on a Debian-based distribution like Ubuntu, try apt-get:

```
$ sudo apt-get install git-all
```

Installing on Windows

- There are also a few ways to install Git on Windows.
- The most official build is available for download on the Git website. Just go to ***http://git-scm.com/download/*** ***win*** and the download will start automatically.
- Note that this is a project called Git for Windows, which is separate from Git itself; for more information on it, go to ***https://git-for-windows.github.io/***.
- Another easy way to get Git installed is by installing GitHub for Windows.
- The installer includes a command line version of Git as well as the GUI. It also works well with Powershell, and sets up solid credential caching and sane CRLF settings.
- We'll learn more about those things a little later, but suffice it to say they're things you want.
- You can download this from the GitHub for Windows website, at ***http://windows.github.com***.

32

First-Time Git Setup

- Now that you have Git on your system, you'll want to do a few things to customize your Git environment.
- You should have to do these things only once on any given computer; they'll stick around between upgrades.
- You can also change them at any time by running through the commands again.
- Git comes with a tool called git config that lets you get and set configuration variables that control all aspects of how Git looks and operates.

First-Time Git Setup

- These variables can be stored in three different places:

1. /etc/gitconfig file: Contains values for every user on the system and all their repositories. If you pass the option --system to git config, it reads and writes from this file specifically.

2. ~/.gitconfig or ~/.config/git/config file: Specific to your user.
You can make Git read and write to this file specifically by passing the --global option.

3. config file in the Git directory (that is, .git/config) of whatever repository you're currently using: Specific to that single repository.

34

First-Time Git Setup

- Each level overrides values in the previous level, so values in .git/config trump those in /etc/gitconfig.
- On Windows systems, Git looks for the .gitconfig file in the $HOME directory (C:\Users\$USER for most people).

35

Your Identity

- The first thing you should do when you install Git is to set your user name and email address.
- This is important because every Git commit uses this information, and it's immutably baked into the commits you start creating:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Your Identity

- Again, you need to do this only once if you pass the --global option, because then Git will always use that information for anything you do on that system.
- If you want to override this with a different name or email address for specific projects, you can run the command without the --global option when you're in that project.

Checking Your Settings

- If you want to check your settings, you can use the git config --list command to list all the settings Git can find at that point:

$ git config –list

```
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

38

Checking Your Settings

- You may see keys more than once, because Git reads the same key from different files (/etc/gitconfig and ~/.gitconfig, for example).
- In this case, Git uses the last value for each unique key it sees.

- You can also check what Git thinks a specific key's value is by typing **git config <key>:**

```
$ git config user.name
John Doe
```

Getting Help

- If you ever need help while using Git, there are three ways to get the manual page (manpage) help for any of the Git commands:

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

- For example, you can get the manpage help for the config command by running:

```
$ git help config
```

Module 1 Lab

- See lab handout

41

Module 2: Git Basics

Getting a Git Repository

- You can get a Git project using two main approaches.
- The first takes an existing project or directory and imports it into Git.
- The second clones an existing Git repository from another server.

Initializing a Repository in an Existing Directory

- If you're starting to track an existing project in Git, you need to go to the project's directory and type:

```
$ git init
```

- This creates a new subdirectory named .git that contains all of your necessary repository files – a Git repository skeleton.
- At this point, nothing in your project is tracked yet.

Initializing a Repository in an Existing Directory

- If you want to start version-controlling existing files (as opposed to an empty directory), you should probably begin tracking those files and do an initial commit.
- You can accomplish that with a few git add commands that specify the files you want to track, followed by a git commit:

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'initial project version'
```

- We'll go over what these commands do in just a minute.
- At this point, you have a Git repository with tracked files and an initial commit.

45

Cloning an Existing Repository

- If you want to get a copy of an existing Git repository – for example, a project you'd like to contribute to – the command you need is git clone.
- If you're familiar with other VCS systems such as Subversion, you'll notice that the command is "clone" and not "checkout".
- This is an important distinction – instead of getting just a working copy, Git receives a full copy of nearly all data that the server has.
- Every version of every file for the history of the project is pulled down by default when you run git clone.
- In fact, if your server disk gets corrupted, you can often use nearly any of the clones on any client to set the server back to the state it was in when it was cloned

Cloning an Existing Repository

- You clone a repository with git clone [url].
- For example, if you want to clone the Git linkable library called libgit2, you can do so like this:

```
$ git clone https://github.com/libgit2/libgit2
```

- That creates a directory named “libgit2”, initializes a .git directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version.
- If you go into the new libgit2 directory, you’ll see the project files in there, ready to be worked on or used.

47

Cloning an Existing Repository

- If you want to clone the repository into a directory named something other than "libgit2", you can specify that as the next command-line option:

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

- That command does the same thing as the previous one, but the target directory is called mylibgit.

Recording Changes to the Repository

- You have a bona fide Git repository and a checkout or working copy of the files for that project.
- You need to make some changes and commit snapshots of those changes into your repository each time the project reaches a state you want to record.
- Remember that each file in your working directory can be in one of two states: tracked or untracked.
- Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged.
- Untracked files are everything else – any files in your working directory that were not in your last snapshot and are not in your staging area.

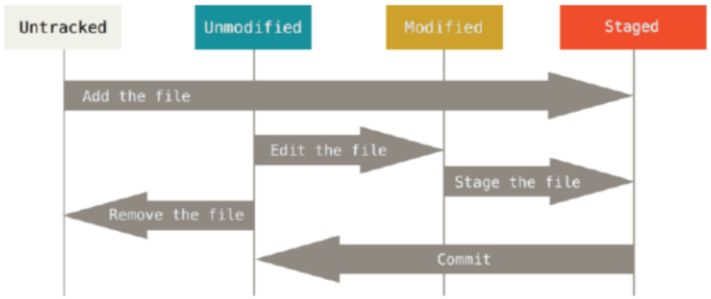
49

Recording Changes to the Repository

- When you first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven't edited anything.
- As you edit files, Git sees them as modified, because you've changed them since your last commit.
- You stage these modified files and then commit all your staged changes, and the cycle repeats.

50

The lifecycle of the status of your files.



51

Checking the Status of Your Files

- The main tool you use to determine which files are in which state is the git status command.
- If you run this command directly after a clone, you should see something like this:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Checking the Status of Your Files

- This means you have a clean working directory – in other words, there are no tracked and modified files.
- Git also doesn't see any untracked files, or they would be listed here.
- Finally, the command tells you which branch you're on and informs you that it has not diverged from the same branch on the server.
- For now, that branch is always "master", which is the default; you won't worry about it here.

Checking the Status of Your Files

- Let's say you add a new file to your project, a simple README file.
- If the file didn't exist before, and you run git status, you see your untracked file like so:

```
$ echo 'My Project' > README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
(use "git add <file>..." to include in what will be committed)
README
nothing added to commit but untracked files present (use "git add" to track)
```

Checking the Status of Your Files

- You can see that your new README file is untracked, because it's under the "Untracked files" heading in your status output.
- Untracked basically means that Git sees a file you didn't have in the previous snapshot (commit); Git won't start including it in your commit snapshots until you explicitly tell it to do so.
- It does this so you don't accidentally begin including generated binary files or other files that you did not mean to include.
- You do want to start including README, so let's start tracking the file.

Tracking New Files

- In order to begin tracking a new file, you use the command git add.
- To begin tracking the REDME file, you can run this:

```
$ git add README
```

- If you run your status command again, you can see that your README file is now tracked and staged to be committed:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
new file: README
```

Tracking New Files

- You can tell that it's staged because it's under the "Changes to be committed" heading.
- If you commit at this point, the version of the file at the time you ran git add is what will be in the historical snapshot.
- You may recall that when you ran git init earlier, you then ran git add (files) – that was to begin tracking files in your directory.
- The git add command takes a path name for either a file or a directory; if it's a directory, the command adds all the files in that directory recursively.

57

Staging Modified Files

- Let's change a file that was already tracked. If you change a previously tracked file called CONTRIBUTING.md and then run your git status command again, you get something that looks like this:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
new file: README
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working directory)
modified: CONTRIBUTING.md
```

Staging Modified Files

- The CONTRIBUTING.md file appears under a section named “Changes not staged for commit” – which means that a file that is tracked has been modified in the working directory but not yet staged.
- To stage it, you run the git add command. git add is a multipurpose command – you use it to begin tracking new files, to stage files, and to do other things like marking merge-conflicted files as resolved.
- It may be helpful to think of it more as “add this content to the next commit” rather than “add this file to the project”.

59

Staging Modified Files

- Let's run git add now to stage the CONTRIBUTING.md file, and then run git status again:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
new file: README
modified: CONTRIBUTING.md
```

Staging Modified Files

- Both files are staged and will go into your next commit.
- At this point, suppose you remember one little change that you want to make in CONTRIBUTING.md before you commit it.
- You open it again and make that change, and you're ready to commit. However, let's run git status one more time:

```
$ vim CONTRIBUTING.md
$ git status

On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
new file: README
modified: CONTRIBUTING.md
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working directory)
modified: CONTRIBUTING.md
```

Staging Modified Files

- Now CONTRIBUTING.md is listed as both staged and unstaged.
- How is that possible?
- It turns out that Git stages a file exactly as it is when you run the git add command.
- If you commit now, the version of CONTRIBUTING.md as it was when you last ran the git add command is how it will go into the commit, not the version of the file as it looks in your working directory when you run git commit.

Staging Modified Files

- If you modify a file a!er you run git add, you have to run git add again to stage the latest version of the file:

```
$ git add CONTRIBUTING.md
$ git status
```

```
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
new file: README
modified: CONTRIBUTING.md
```

Short Status

- While the git status output is pretty comprehensive, it's also quite wordy.
- Git also has a short status flag so you can see your changes in a more compact way.
- If you run git status -s or git status --short you get a far more simplified output from the command:

```
$ git status -s
M README
MM Rakefile
A lib/git.rb
M lib/simplegit.rb
?? LICENSE.txt
```

- New files that aren't tracked have a ?? next to them, new files that have been added to the staging area have an A, modified files have an M and so on.

Short Status

- There are two columns to the output - the le!-hand column indicates the status of the staging area and the right-hand column indicates the status of the working tree.
- So for example in that output, the README file is modified in the working directory but not yet staged, while the lib/simplegit.rb file is modified and staged.
- The Rakefile was modified, staged and then modified again, so there are changes to it that are both staged and unstaged.

Ignoring Files

- Often, you'll have a class of files that you don't want Git to automatically add or even show you as being untracked.
- These are generally automatically generated files such as log files or files produced by your build system.
- In such cases, you can create a file listing patterns to match them named .gitignore.
- Here is an example .gitignore file:

```
$ cat .gitignore

*.[oa]
*~
```

66

Ignoring Files

- The first line tells Git to ignore any files ending in ".o" or ".a" – object and archive files that may be the product of building your code.
- The second line tells Git to ignore all files whose names end with a tilde (~), which is used by many text editors such as Emacs to mark temporary files.
- You may also include a log, tmp, or pid directory; automatically generated documentation; and so on.
- Setting up a .gitignore file before you get going is generally a good idea so you don't accidentally commit files that you really don't want in your Git repository.

Ignoring Files

- The rules for the patterns you can put in the .gitignore file are as follows:

- Blank lines or lines starting with # are ignored.
- Standard glob patterns work.
- You can start patterns with a forward slash (/) to avoid recursivity.
- You can end patterns with a forward slash (/) to specify a directory.
- You can negate a pattern by starting it with an exclamation point (!).

Viewing Your Staged and Unstaged Changes

- If the git status command is too vague for you – you want to know exactly what you changed, not just which files were changed – you can use the git diff command.
- We'll cover git diff in more detail later, but you'll probably use it most often to answer these two questions:

What have you changed but not yet staged?
And what have you staged that you are about to commit?

- Although git status answers those questions very generally by listing the file names, git diff shows you the exact lines added and removed – the patch, as it were.

69

Viewing Your Staged and Unstaged Changes

- Let's say you edit and stage the README file again and then edit the CONTRIBUTING.md file without staging it.
- If you run your git status command, you once again see something like this:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
modified: README
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working directory)
```

Viewing Your Staged and Unstaged Changes

- To see what you've changed but not yet staged, type git diff with no other arguments:

```
$ git diff
diff --git a/CONTRIBUTING.md  b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
.....
.....
```

71

Viewing Your Staged and Unstaged Changes

- That command compares what is in your working directory with what is in your staging area.
- The result tells you the changes you've made that you haven't yet staged.
- If you want to see what you've staged that will go into your next commit, you can use git diff --staged.
- This command compares your staged changes to your last commit:

```
$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project
```

Viewing Your Staged and Unstaged Changes

- It's important to note that git diff by itself doesn't show all changes
- made since your last commit – only changes that are still unstaged. This can be
- confusing, because if you've staged all of your changes, git diff will give you
- no output.

Committing Your Changes

- Now that your staging area is set up the way you want it, you can commit your changes.
- Remember that anything that is still unstaged – any files you have created or modified that you haven't run git add on since you edited them –won't go into this commit.
- They will stay as modified files on your disk.
- In this case, let's say that the last time you ran git status, you saw that everything was staged, so you're ready to commit your changes.
- The simplest way to commit is to type git commit:

```
$ git commit
```

74

Removing Files

- To remove a file from Git, you have to remove it from your tracked files (more
- accurately, remove it from your staging area) and then commit.
- The git rm command does that, and also removes the file from your working directory so you don't see it as an untracked file the next time around.
- If you simply remove the file from your working directory, it shows up under the "Changed but not updated" (that is, unstaged) area of your git status output:

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
(use "git add/rm <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working directory)
deleted: PROJECTS.md
no changes added to commit (use "git add" and/or "git commit -a")
```

Removing Files

- Then, if you run git rm, it stages the file's removal:

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
```

```
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
deleted: PROJECTS.md
```

Moving Files

- Unlike many other VCS systems, Git doesn't explicitly track file movement.
- If you rename a file in Git, no metadata is stored in Git that tells it you renamed the file.
- However, Git is pretty smart about figuring that out a!er the fact – we'll deal with detecting file movement a bit later.
- Thus it's a bit confusing that Git has a mv command.
- If you want to rename a file in Git, you can run something like:

```
$ git mv file_from file_to
```

and it works fine.

- In fact, if you run something like this and look at the status, you'll see that Git considers it a renamed file:

```
$ git mv README.md README
$ git status
```

- On branch master
- Your branch is up-to-date with 'origin/master'.
- Changes to be committed:
- (use "git reset HEAD <file>..." to unstage)
- renamed: README.md -> README

77

Moving Files

- However, this is equivalent to running something like this:

```
$ mv README.md README
$ git rm README.md
$ git add README
```

78

Viewing the Commit History

- After you have created several commits, or if you have cloned a repository with an existing commit history, you'll probably want to look back to see what has happened.
- The most basic and powerful tool to do this is the git log command.
- These examples use a very simple project called "simplegit".
- To get the project, run

```
$ git clone https://github.com/schacon/simplegit-progit
```

Viewing the Commit History

- When you run git log in this project, you should get output that looks something like this:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700
changed the version number
.....
.....
```

Unmodifying a Modified File

- What if you realize that you don't want to keep your changes to the CONTRIBUTING. md file?
- How can you easily unmodify it – revert it back to what it looked like when you last committed (or initially cloned, or however you got it into your working directory)?
- Luckily, git status tells you how to do that, too.
- In the last example output, the unstaged area looks like this:

```
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git checkout -- <file>..." to discard changes in working directory)
modified: CONTRIBUTING.md
```

81

- It tells you pretty explicitly how to discard the changes you've made. Let's do what it says:

```
$ git checkout -- CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
renamed: README.md -> README
```

Adding Remote Repositories

- We've mentioned and given some demonstrations of how the clone command
- implicitly adds the origin remote for you.
- Here's how to add a new remote explicitly.
- To add a new remote Git repository as a shortname you can reference easily, run git remote add <shortname> <url>:

$ git remote

origin

$ git remote add pb https://github.com/paulboone/ticgit

$ git remote -v

origin https://github.com/schacon/ticgit (fetch)
origin https://github.com/schacon/ticgit (push)
pb https://github.com/paulboone/ticgit (fetch)
pb https://github.com/paulboone/ticgit (push)

Adding Remote Repositories

- Now you can use the string pb on the command line in lieu of the whole URL.
- For example, if you want to fetch all the information that Paul has but that you don't yet have in your repository, you can run git fetch pb:

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
```

84

Fetching and Pulling from Your Remotes

- To get data from your remote projects, you can run:

```
$ git fetch [remote-name]
```

- The command goes out to that remote project and pulls down all the data from that remote project that you don't have yet.
- After you do this, you should have references to all the branches from that remote, which you can merge in or inspect at any time.

Pushing to Your Remotes

- When you have your project at a point that you want to share, you have to push it upstream.
- The command for this is simple: git push [remote-name] [branch-name].
- If you want to push your master branch to your origin server(again, cloning generally sets up both of those names for you automatically), then you can run this to push any commits you've done back up to the server:

$ git push origin master

Pushing to Your Remotes

- This command works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime.
- If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected.
- You'll have to fetch their work first and incorporate it into yours before you'll be allowed to push.

Inspecting a Remote

- If you want to see more information about a particular remote, you can use the git remote show [remote-name] command.
- If you run this command with a particular shortname, such as origin, you get something like this:

```
$ git remote show origin
* remote origin
Fetch URL: https://github.com/schacon/ticgit
Push URL: https://github.com/schacon/ticgit
HEAD branch: master
Remote branches:
master tracked
dev-branch tracked
Local branch configured for 'git pull':
master merges with remote master
Local ref configured for 'git push':
master pushes to master (up to date)
```

Inspecting a Remote

- It lists the URL for the remote repository as well as the tracking branch information.
- The command helpfully tells you that if you're on the master branch and you run git pull, it will automatically merge in the master branch on the remote after it fetches all the remote references.
- It also lists all the remote references it has pulled down.

Removing and Renaming Remotes

- You can run git remote rename to change a remote's shortname.
- For instance, if you want to rename pb to paul, you can do so with git remote rename:

```
$ git remote rename pb paul
$ git remote
origin
paul
```

Tagging

- Like most VCSs, Git has the ability to tag specific points in history as being important.
- Typically people use this functionality to mark release points (v1.0, and
- so on).
- In this section, you'll learn how to list the available tags, how to create new tags, and what the different types of tags are.

91

Listing Your Tags

- Listing the available tags in Git is straightforward.
- Just type git tag:

```
$ git tag
v0.1
v1.3
```

- This command lists the tags in alphabetical order; the order in which they appear has no real importance.

Listing Your Tags

- You can also search for tags with a particular pattern.
- The Git source repo, for instance, contains more than 500 tags.
- If you're only interested in looking at the 1.8.5 series, you can run this:

```
$ git tag -l "v1.8.5*"
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
```

93

Creating Tags

- Git uses two main types of tags: lightweight and annotated.
- A lightweight tag is very much like a branch that doesn't change – it's just a pointer to a specific commit.
- Annotated tags, however, are stored as full objects in the Git database.
- They're checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG).
- It's generally recommended that you create annotated tags so you can have all this information; but if you want a temporary tag or for some reason don't want to keep the other information, lightweight tags are available too.

Annotated Tags

- Creating an annotated tag in Git is simple. The easiest way is to specify -a when you run the tag command:

```
$ git tag -a v1.4 -m "my version 1.4"
$ git tag
v0.1
v1.3
v1.4
```

Annotated Tags

- The -m specifies a tagging message, which is stored with the tag.
- If you don't specify a message for an annotated tag, Git launches your editor so you can type it in.
- You can see the tag data along with the commit that was tagged by using the git show command:

```
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date: Sat May 3 20:19:12 2014 -0700
my version 1.4
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700
```

Lightweight Tags

- Another way to tag commits is with a lightweight tag.
- This is basically the commit checksum stored in a file – no other information is kept.
- To create a lightweight tag, don't supply the -a, -s, or -m option:

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

97

Git Aliases

- Before we finish this module on basic Git, there's just one little tip that can make your Git experience simpler, easier, and more familiar: aliases.
- Git doesn't automatically infer your command if you type it in partially.
- If you don't want to type the entire text of each of the Git commands, you can easily set up an alias for each command using git config.
- Here are a couple of examples you may want to set up:

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

98

Module 2 Lab

- See Lab Handout

99

Module 3: Git Branching

Git Branching

- Nearly every VCS has some form of branching support.
- Branching means you diverge from the main line of development and continue to do work without messing with that main line.
- In many VCS tools, this is a somewhat expensive process, often requiring you to create a new copy of your source code directory, which can take a long time for large projects.

101

Branches in a Nutshell

- To really understand the way Git does branching, we need to take a step back and examine how Git stores its data.
- As you may remember from **Module 1**, Git doesn't store data as a series of changesets or differences, but instead as a series of snapshots.
- When you make a commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged.
- This object also contains the author's name and email, the message that you typed, and pointers to the commit or commits that directly came before this commit (its parent or parents): zero parents for the initial commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches.

Branches in a Nutshell

- To visualize this, let's assume that you have a directory containing three files, and you stage them all and commit.
- Staging the files checksums each one (the SHA-1 hash we mentioned in **Module 1**), stores that version of the file in the Git repository (Git refers to them as blobs), and adds that checksum to the staging area:

```
$ git add README test.rb LICENSE
$ git commit -m 'The initial commit of my project'
```

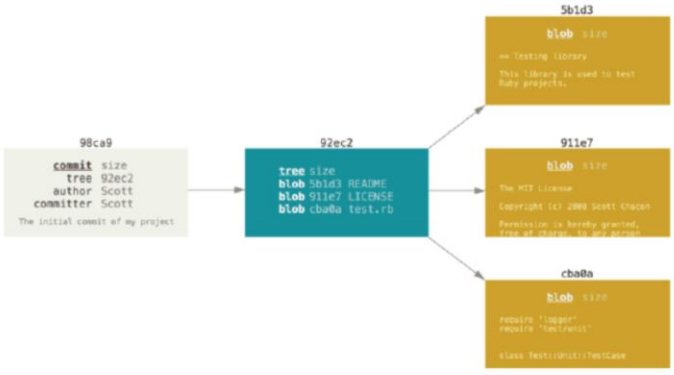
- When you create the commit by running git commit, Git checksums each subdirectory (in this case, just the root project directory) and stores those tree objects in the Git repository.
- Git then creates a commit object that has the metadata and a pointer to the root project tree so it can re-create that snapshot when needed.

103

Branches in a Nutshell

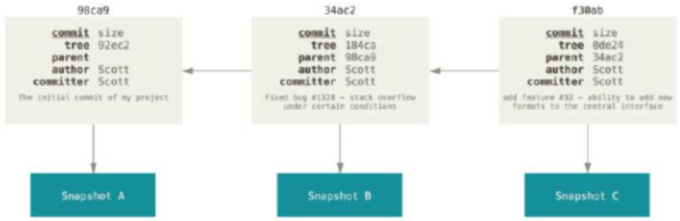
- Your Git repository now contains five objects: one blob for the contents of each of your three files, one tree that lists the contents of the directory and specifies which file names are stored as which blobs, and one commit with the pointer to that root tree and all the commit metadata.

A commit and its tree



105

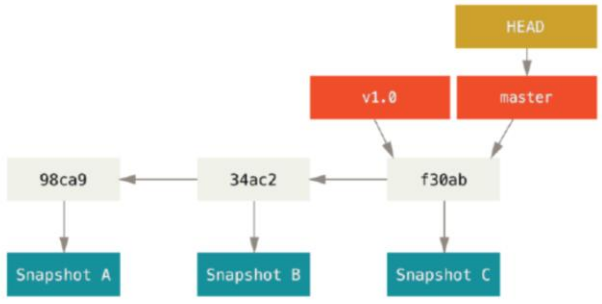
Commits and their parents



Branches in a Nutshell

- A branch in Git is simply a lightweight movable pointer to one of these commits.
- The default branch name in Git is master.
- As you start making commits, you're given a master branch that points to the last commit you made.
- Every time you commit, it moves forward automatically.
- The "master" branch in Git is not a special branch.
- It is exactly like any other branch.
- The only reason nearly every repository has one is that the git init command creates it by default and most people don't bother to change it.

A branch and its commit history



108

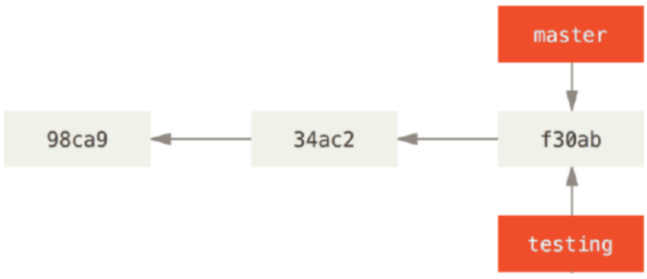
Creating a New Branch

- What happens if you create a new branch?
- Well, doing so creates a new pointer for you to move around.
- Let's say you create a new branch called testing.
- You do this with the git branch command:

```
$ git branch testing
```

- This creates a new pointer to the same commit you're currently on.

Two branches pointing into the same series of commits



110

Creating a New Branch

- How does Git know what branch you're currently on?
- It keeps a special pointer called HEAD.
- Note that this is a lot different than the concept of HEAD in other VCSs you may be used to, such as Subversion or CVS.
- In Git, this is a pointer to the local branch you're currently on. In this case, you're still on master.
- The git branch command only created a new branch – it didn't switch to that branch.

HEAD pointing to a branch

112

Creating a New Branch

- You can easily see this by running a simple git log command that shows you where the branch pointers are pointing.
- This option is called --decorate.

$ git log --oneline --decorate

f30ab (HEAD -> master, testing) add feature #32 - ability to add new formats to the central interface

34ac2 Fixed bug #1328 - stack overflow under certain conditions

98ca9 The initial commit of my project

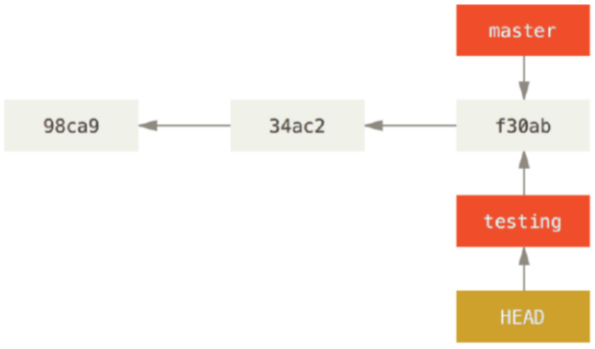
Switching Branches

- To switch to an existing branch, you run the git checkout command.
- Let's switch to the new testing branch:

```
$ git checkout testing
```

- This moves HEAD to point to the testing branch.

HEAD points to the current branch



115

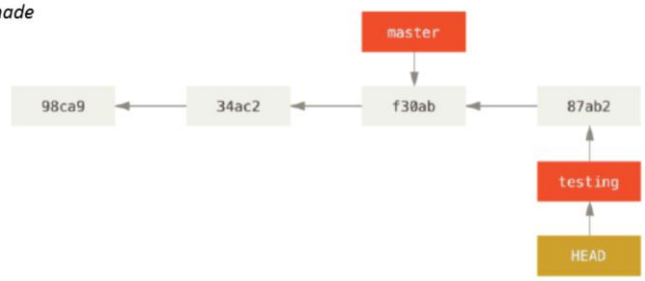
Switching Branches

- What is the significance of that?
- Well, let's do another commit:

```
$ vim test.rb
$ git commit -a -m 'made a change'
```

116

The HEAD branch moves forward when a commit is made



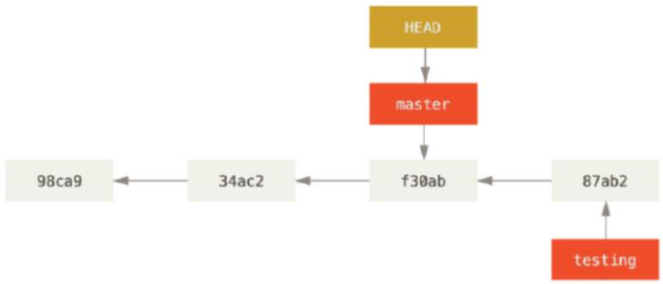
117

Switching Branches

- This is interesting, because now your testing branch has moved forward, but your master branch still points to the commit you were on when you ran git checkout to switch branches. Let's switch back to the master branch:

```
$ git checkout master
```

HEAD moves when you checkout



119

Switching Branches

- That command did two things.
- It moved the HEAD pointer back to point to the master branch, and it reverted the files in your working directory back to the snapshot that master points to.
- This also means the changes you make from this point forward will diverge from an older version of the project.
- It essentially rewinds the work you've done in your testing branch so you can go in a different direction.

Switching Branches

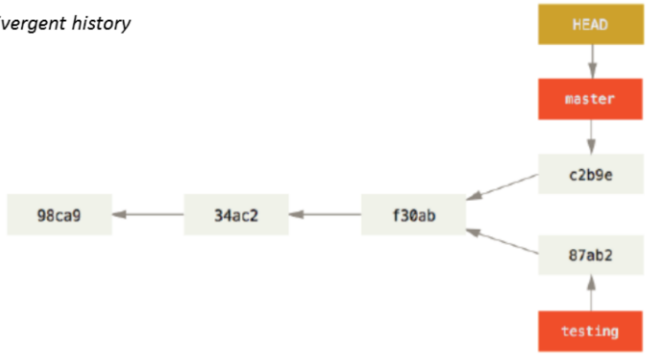
- Let's make a few changes and commit again:

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

- Now your project history has diverged.
- You created and switched to a branch, did some work on it, and then switched back to your main branch and did other work.
- Both of those changes are isolated in separate branches: you can switch back and forth between the branches and merge them together when you're ready.
- And you did all that with simple branch, checkout, and commit commands.

121

Divergent history



122

Basic Branching and Merging

- Let's go through a simple example of branching and merging with a workflow that you might use in the real world. You'll follow these steps:

1. Do work on a web site.
2. Create a branch for a new story you're working on.
3. Do some work in that branch.

- At this stage, you'll receive a call that another issue is critical and you need a hotfix. You'll do the following:

1. Switch to your production branch.
2. Create a branch to add the hotfix.
3. After it's tested, merge the hotfix branch, and push to production.
4. Switch back to your original story and continue working.

123

Basic Branching

- First, let's say you're working on your project and have a couple of commits already.

124

Basic Branching

- You've decided that you're going to work on issue #53 in whatever issue tracking system your company uses.
- To create a branch and switch to it at the same time, you can run the git checkout command with the -b switch:

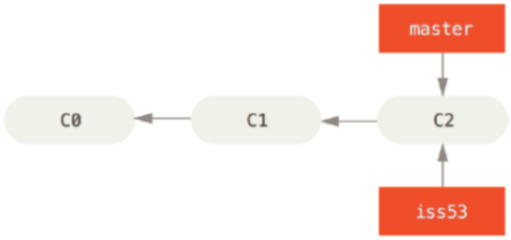
```
$ git checkout -b iss53
```

- Switched to a new branch "iss53"
- This is shorthand for:

```
$ git branch iss53
$ git checkout iss53
```

125

Creating a new branch pointer



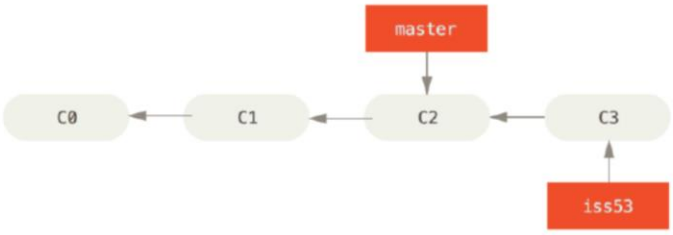
126

Basic Branching

- You work on your web site and do some commits. Doing so moves the iss53 branch forward, because you have it checked out (that is, your HEAD is pointing to it):

```
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
```

The iss53 branch has moved forward with your work



128

Basic Branching

- Now you get the call that there is an issue with the web site, and you need to fix it immediately.
- With Git, you don't have to deploy your fix along with the iss53 changes you've made, and you don't have to put a lot of e ort into reverting those changes before you can work on applying your fix to what is in production.
- All you have to do is switch back to your master branch.
- However, before you do that, note that if your working directory or staging area has uncommitted changes that conflict with the branch you're checking out, Git won't let you switch branches.
- It's best to have a clean working state when you switch branches.

129

Basic Branching

- Let's assume you've committed all your changes, so you can switch back to your master branch:

```
$ git checkout master
Switched to branch 'master'
```

- At this point, your project working directory is exactly the way it was before you started working on issue #53, and you can concentrate on your hotfix.
- This is an important point to remember: when you switch branches, Git resets your working directory to look like it did the last time you committed on that branch.
- It adds, removes, and modifies files automatically to make sure your working copy is what the branch looked like on your last commit to it.

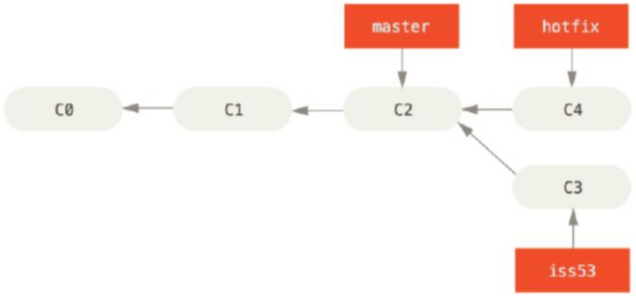
130

Basic Branching

- Next, you have a hotfix to make. Let's create a hotfix branch on which to work until it's completed:

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address‘
[hotfix 1fb7853] fixed the broken email address
1 file changed, 2 insertions(+)
```

Basic Branching



132

Basic Branching

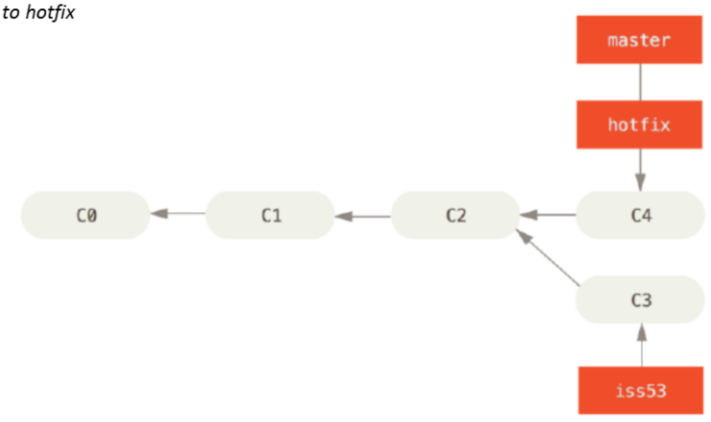
- You can run your tests, make sure the hotfix is what you want, and merge it back into your master branch to deploy to production.
- You do this with the git merge command:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
index.html | 2 ++
1 file changed, 2 insertions(+)
```

Basic Branching

- You'll notice the phrase "fast-forward" in that merge.
- Because the commit C4 pointed to by the branch hotfix you merged in was directly ahead of the commit C2 you're on, Git simply moves the pointer forward.
- To phrase that another way, when you try to merge one commit with a commit that can be reached by following the first commit's history, Git simplifies things by moving the pointer forward because there is no divergent work to merge together – this is called a "fast-forward."
- Your change is now in the snapshot of the commit pointed to by the master branch, and you can deploy the fix.

master is fastforwarded to hotfix



135

Basic Branching

- After your super-important fix is deployed, you're ready to switch back to the work you were doing before you were interrupted.
- However, first you'll delete the hotfix branch, because you no longer need it – the master branch points at the same place.
- You can delete it with the -d option to git branch:

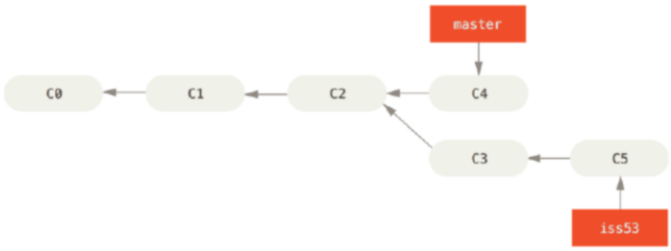
```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

Basic Branching

- Now you can switch back to your work-in-progress branch on issue #53 and continue working on it.

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
1 file changed, 1 insertion(+)
```

Work continues on iss53



138

Basic Branching

- It's worth noting here that the work you did in your hotfix branch is not contained in the files in your iss53 branch.
- If you need to pull it in, you can merge your master branch into your iss53 branch by running git merge master, or you can wait to integrate those changes until you decide to pull the iss53 branch back into master later.

Basic Merging

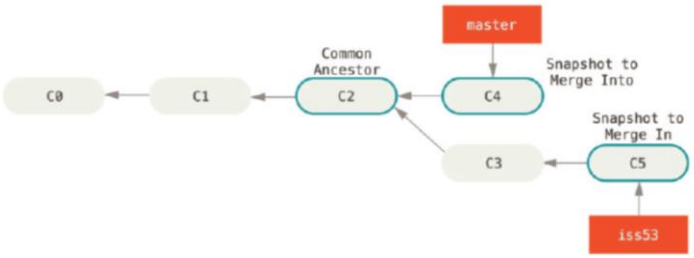
- Suppose you've decided that your issue #53 work is complete and ready to be merged into your master branch.
- In order to do that, you'll merge your iss53 branch into master, much like you merged your hotfix branch earlier.
- All you have to do is check out the branch you wish to merge into and then run the git merge command:

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html | 1 +
1 file changed, 1 insertion(+)
```

Basic Merging

- This looks a bit different than the hotfix merge you did earlier. In this case, your development history has diverged from some older point.
- Because the commit on the branch you're on isn't a direct ancestor of the branch you're merging in, Git has to do some work.
- In this case, Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two.

Three snapshots used in a typical merge

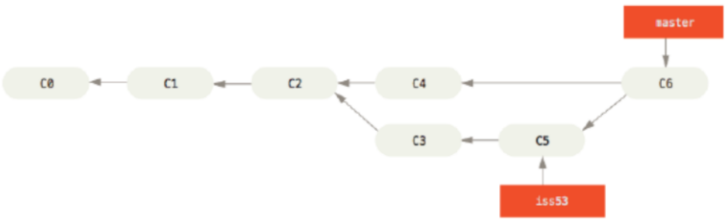


142

Basic Merging

- Instead of just moving the branch pointer forward, Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it.
- This is referred to as a merge commit, and is special in that it has more than one parent.

A merge commit



144

Basic Merging

- It's worth pointing out that Git determines the best common ancestor to use for its merge base; this is different than older tools like CVS or Subversion (before version 1.5), where the developer doing the merge had to figure out the best merge base for themselves.
- This makes merging a heck of a lot easier in Git than in these other systems.

Basic Merging

- Now that your work is merged in, you have no further need for the iss53 branch.
- You can close the ticket in your ticket-tracking system, and delete the branch:

```
$ git branch -d iss53
```

Basic Merge Conflicts

- Occasionally, this process doesn't go smoothly.
- If you changed the same part of the same file differently in the two branches you're merging together, Git won't be able to merge them cleanly.
- If your fix for issue #53 modified the same part of a file as the hotfix, you'll get a merge conflict that looks something like this:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

147

Basic Merge Conflicts

- Git hasn't automatically created a new merge commit. It has paused the process while you resolve the conflict.
- If you want to see which files are unmerged at any point a!er a merge conflict, you can run git status:

```
$ git status
On branch master
You have unmerged paths.
(fix conflicts and run "git commit")
Unmerged paths:
(use "git add <file>..." to mark resolution)
both modified: index.html
no changes added to commit (use "git add" and/or "git commit -a")
```

148

Basic Merge Conflicts

- Anything that has merge conflicts and hasn't been resolved is listed as unmerged.
- Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts.
- Your file contains a section that looks something like this:

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

149

Basic Merge Conflicts

- This means the version in HEAD (your master branch, because that was what you had checked out when you ran your merge command) is the top part of that block (everything above the =======), while the version in your iss53 branch looks like everything in the bottom part.
- In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself.

Basic Merge Conflicts

- For instance, you might resolve this conflict by replacing the entire block with this:

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

- This resolution has a little of each section, and the <<<<<<<, =======, and >>>>>>> lines have been completely removed.
- After you've resolved each of these sections in each conflicted file, run git add on each file to mark it as resolved.
- Staging the file marks it as resolved in Git.

151

Basic Merge Conflicts

- If you want to use a graphical tool to resolve these issues, you can run git mergetool, which fires up an appropriate visual merge tool and walks you through the conflicts:

```
$ git mergetool
This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge
p4merge araxis bc3 codecompare Merging:
index.html
Normal merge conflict for 'index.html':
{local}: modified file
{remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

Basic Merge Conflicts

- After you exit the merge tool, Git asks you if the merge was successful.
- If you tell the script that it was, it stages the file to mark it as resolved for you.
- You can run git status again to verify that all conflicts have been resolved:

```
$ git status
On branch master
All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)
Changes to be committed:
modified: index.html
```

153

Branch Management

- Now that you've created, merged, and deleted some branches, let's look at some branch-management tools that will come in handy when you begin using branches all the time.
- The git branch command does more than just create and delete branches.
- If you run it with no arguments, you get a simple listing of your current branches:

```
$ git branch
iss53
* master
testing
```

Branch Management

- Notice the * character that prefixes the master branch: it indicates the branch that you currently have checked out (i.e., the branch that HEAD points to).
- This means that if you commit at this point, the master branch will be moved forward with your new work. To see the last commit on each branch, you can run git branch -v:

```
$ git branch -v
iss53 93b412c fix javascript issue
* master 7a98805 Merge branch 'iss53'
testing 782fd34 add scott to the author list in the readmes
```

155

Branch Management

- The useful --merged and --no-merged options can filter this list to branches that you have or have not yet merged into the branch you're currently on.
- To see which branches are already merged into the branch you're on, you can run git branch --merged:

```
$ git branch --merged
iss53
* master
```

Branch Management

- Because you already merged in iss53 earlier, you see it in your list.
- Branches on this list without the * in front of them are generally fine to delete with git branch -d; you've already incorporated their work into another branch, so you're not going to lose anything.
- To see all the branches that contain work you haven't yet merged in, you can run git branch --no-merged:

```
$ git branch --no-merged
testing
```

- This shows your other branch.
- Because it contains work that isn't merged in yet, trying to delete it with git branch -d will fail:

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

157