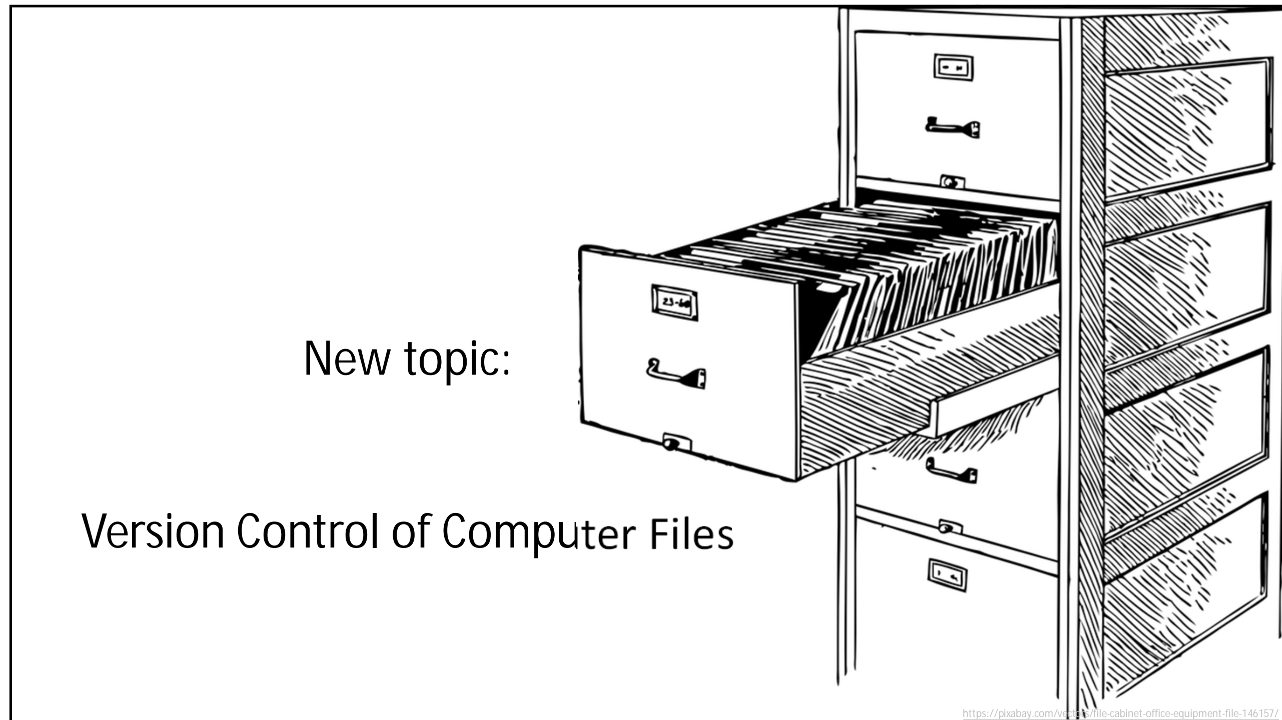


1

Version Control of Computer Files

Typically a project gets big with a lot of related files, all being worked on by a team of people. After a while there are lots of versions of similar files, often having the same filenames. It can become a mess.

A cloud service such as Google Drive, Sky Drive, Dropbox, iCloud, Amazon Cloud Drive, etc. might come to mind, but they are inadequate since they do not keep track of what is up-to-date. They only provide central storage and multiple user access.

Google Docs, OnlyOffice, Dropbox Paper, MS Office Online might come to mind since these allow several editors to work in the same document simultaneously (with multiple cursors active simultaneously). But these are restricted to their office applications. You cannot work in a practical way on software in these programs.

The answer to all these dilemmas we have *version control*, a more sophisticated concept that has a long learning curve but a big payback. All electronics design companies use it in one form or another.

Manual version control was the norm until about 15 years ago. Manual version control means a human secretary keeps a written record of what files are current and maintains archival backups. Modern systems involve way too many files for this to remain practical.

There are two basic types of automated version control, *centralized* or *distributed*.

2

MANUAL Version Control of Computer Files

The repository may be in the cloud or on some person's local storage, but it functions the same anyway.
This requires a repository from which to ***check-out, modify, check-in*** (also known as *lock-modify-unlock)* various files.
A copy remains in the repository (to prevent loss), but only one copy at a time—one check-out at a time—is allowed.

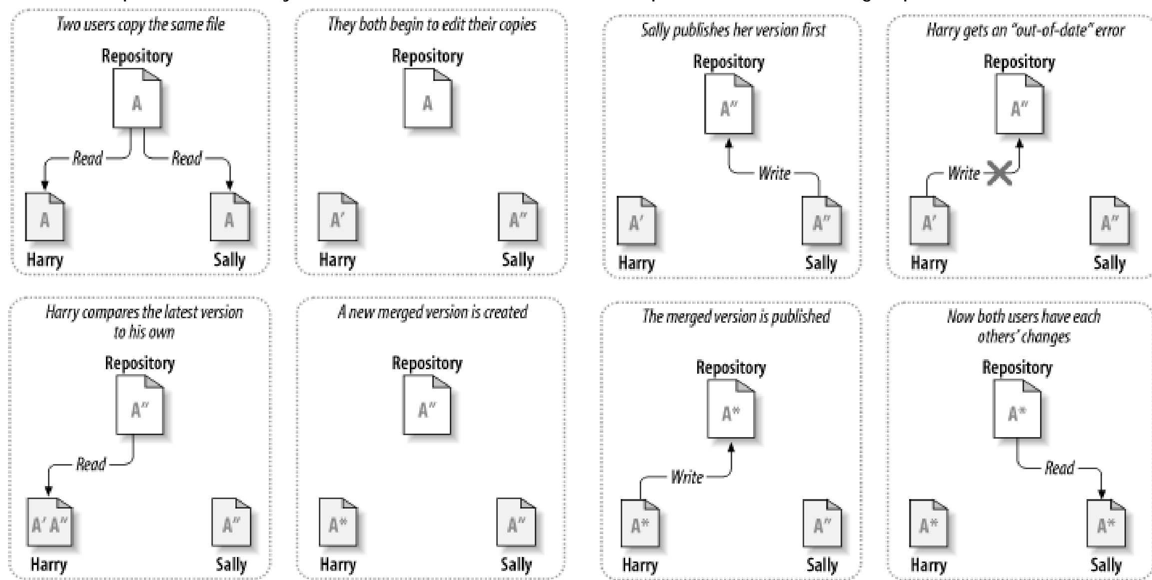
Suppose Albert and Betty are working on a project. Both of them want to edit the Zed file in the repository.
They will have to take turns. Say Albert goes first.
Albert checks out Zed from the repository.
That is, he marks the copy in the repository as "read only" and makes a copy for himself to edit.
Other users may not edit the file. The "read only" status of this file will never be removed.
Albert edits his copy on his computer, then commits his modified file back into the repository.
Committing a file means to re-name the old file in the repository and place the newly edited file in the repository.
Now Betty repeats the process to insert her edits into the zed file.
If they are working on different files, say Albert works on Zed and Betty works on Yikes, then they may work simultaneously.

POTENTIAL PROBLEMS
--Lots of manual labor to check files in and out of the repository. Many chances for error.
--Administrative issues: Albert might forget to check his file back in, needlessly delaying Betty.
--May cause needless delay: What if Albert wants to edit near the beginning of the file and Betty near the end?
Their work would not be mutually conflicting if only there was a way to manage this.
--File check-outs do not necessarily give the integrity desired.
Suppose Albert makes an edit in the beginning of the file but tells nobody and checks the file back in.
Suppose Betty makes an edit without reading the whole file. Suppose her edit depends on a feature that Albert changed. Betty's edits break the file. *With any system, human communication really counts!*
Suppose Zed depends on Yikes. Albert modifies Zed, Betty modifies Yikes and breaks Zed.

3

Computerized Version Control of Computer Files—Centralized or Distributed

Enables a ***copy-modify-merge*** structure—it would be too difficult without a computer
The computer can reliably detect an "out-of-date" error and supervise a manual merge operation.



4

<u>DISTRIBUTED</u> Version Control of Computer Files

All clients have a *local repository* of the entire project.
Any or all of the local repositories can be shared with others.
Typically everyone on the project shares their repository with everyone.
Other people's repositories appear as *remote* repositories to you.
A local repository is just a folder or directory on your computer that has a special file in it (e.g. in our case, *.git)

Use regular OS commands (copy, delete, move, etc) and any software you like
(MS-Word, Photoshop, whatever) to manipulate any file (except *.git)

You will usually have a mutual understanding that one particular repository is "the main repository"

Each team member acts such that the main repository contains the most up-to-date versions of everything.
The main repository gets used like a server, but it has no special privileges or capabilities.
Any repository may be hosted on any OS, via any hardware, or even in the cloud.

Always uses a peer-to-peer structure.

Popular distributed version control programs

- *Git*
- *Mercurial*, also known as *hg*
- *Bazaar*

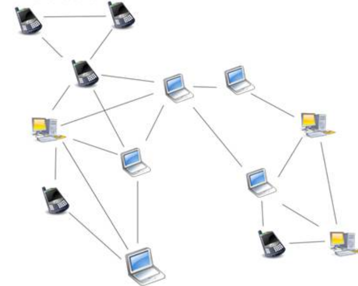


Illustration used by permission CCO1.0 https://en.wikipedia.org/wiki/File:Unstructured_peer-to-peer_network_diagram.png

5

<u>CENTRALIZED</u> Version Control of Computer Files

All clients depend on a central *repository* of the entire project.
The version control software runs on a server. (Client-side software may also be needed, but not always.)
The server software manages the check-out check-in process just described on the previous slide.

Popular centralized version control programs

- *Concurrent Versions Systems* (a.k.a *CVS*, open source)
- *Subversion* (a.k.a *SVN*, open source)
- *Visual SourceSafe* (Microsoft)
- *IBM Rational Clear-Case* (IBM)
- *PVCS* (Micro Focus)

6

Git, a Distrubuted Version Control Program

Git was initially developed on the Unix operating system. (Linux is a variant of Unix—essentially a clone.)
Git's primary interface is a command-line type. (It could have been a GUI, but it is a CLI, and generally it is good!)
The Raspberry Pi runs the "Raspbian" OS, a version of Linux. This also has a CLI
Thus we will be getting familiar with CLI type software.
Git will run on a Raspberry Pi, but we will first learn it and usually use it on Windows.

How does Git Work?
1.) You install a version of Git on your computer. For each type of OS (Windows, OS-X, Linux, etc.) there are several versions of Git to choose from. They are interchangeable and talk pretty well to each other. We will use "GitSCM for Windows." This is called "Git Bash" after it is installed. (As are many other versions of Git!)

2.) You use Git to set up ("init") a "repository" which is a specially enabled folder on your computer.
This folder has a ".git" file in it which you should not touch or delete.
(Note, unless specifically stated otherwise, "local repository" and "repository" mean the same thing.)

3.) You use the repository (special folder) the way you normally would, being sure that all the files you care about For the project are in the repository or a sub-folder of the repository. Initially these files have the status of "untracked." In other words, the ".git" file has no record of them, only the OS's file directory is managing them.

7

Git, a Distrubuted Version Control Program

How does Git Work? (continued)
4.) You may convert any or all files in the repository to "staged" files (add). These files are now logged in the Git system. Appropriate data to do the monitoring is recorded in the .git file. (Generally, staged files are there for your reference, but you are not actively editing them at the time.) These files are given the status of "tracked—staged."

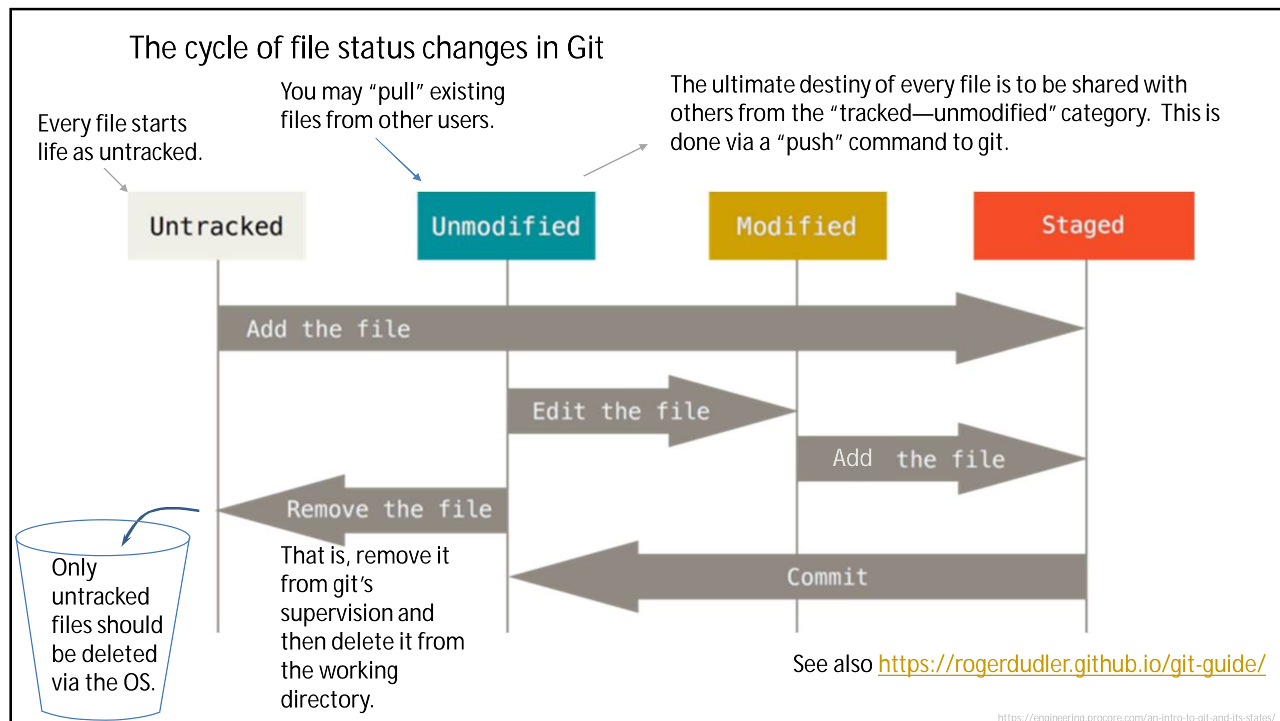
5.) You may take a "snapshot" of your project ("commit"). All the staged files will go into the snapshot. The data to maintain the snapshot is recorded in the .git file. Immediately after taking a snapshot, all the files included in the snapshot are given the status of "tracked—unmodified." A snapshot essentially archives a named or numbered version of your project that you can roll back to at any time.

6.) You may (continue) editing files in your repository. (But never edit .git) When you edit a file its status changes from "tracked—unmodified" to "tracked—modified." These files will not automatically be included in the next snapshot. (You might not be finished editing them. You have to tell Git when you are done editing the file.)

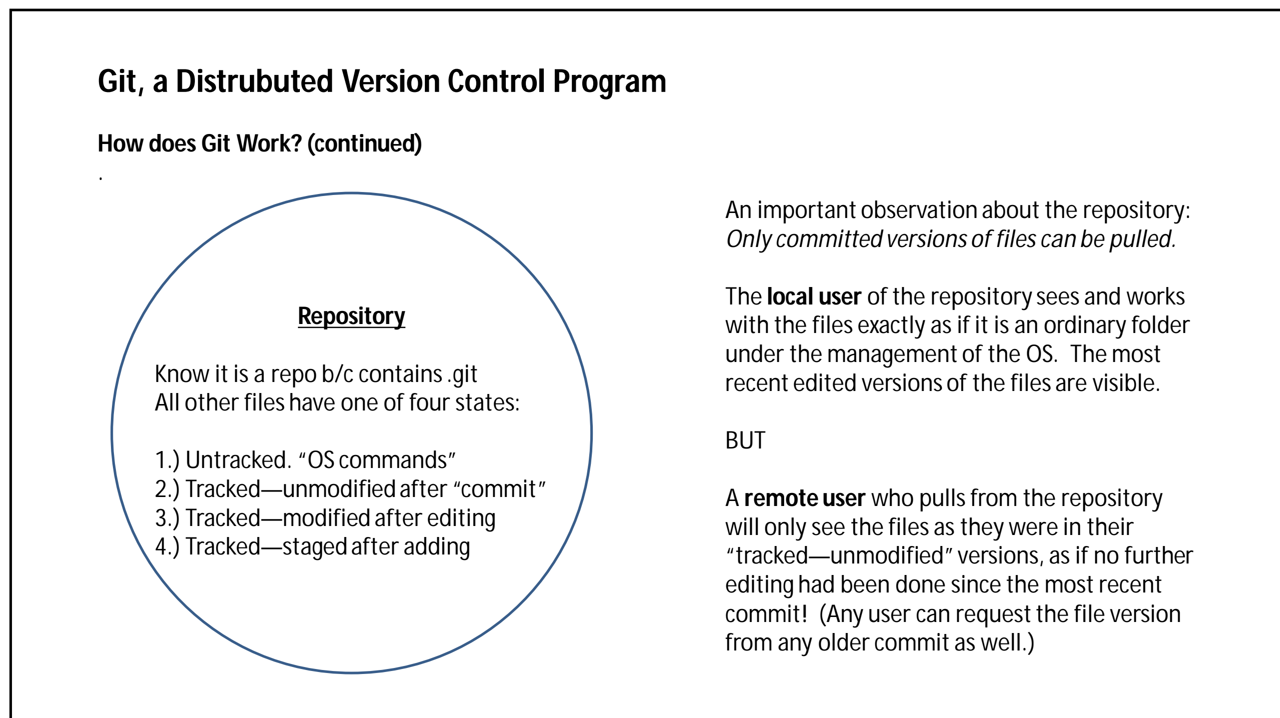
7.) Before creating yet another snapshot including all edits you will usually want to convert "tracked—modified" files to "tracked—staged" files. You will use the "add" command (again) to do that.

8.) Go back to step 5. . . Steps 5, 6, and 7 represent the normal workflow. Note that when you add a new file to a project you need to stage (add) it before it will be tracked! Usually if you add a previously untracked file you also want to snapshot (commit) it immediately. Failing that you will not be able to roll back your first round of edits.

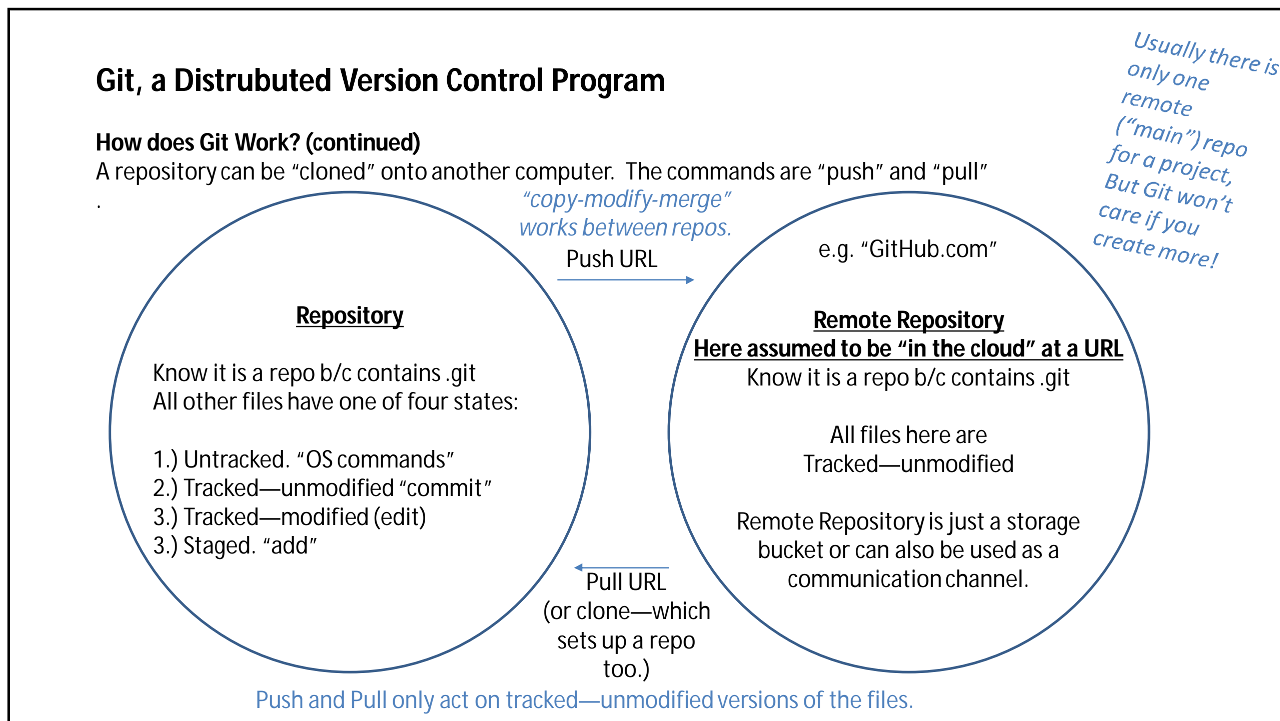
8



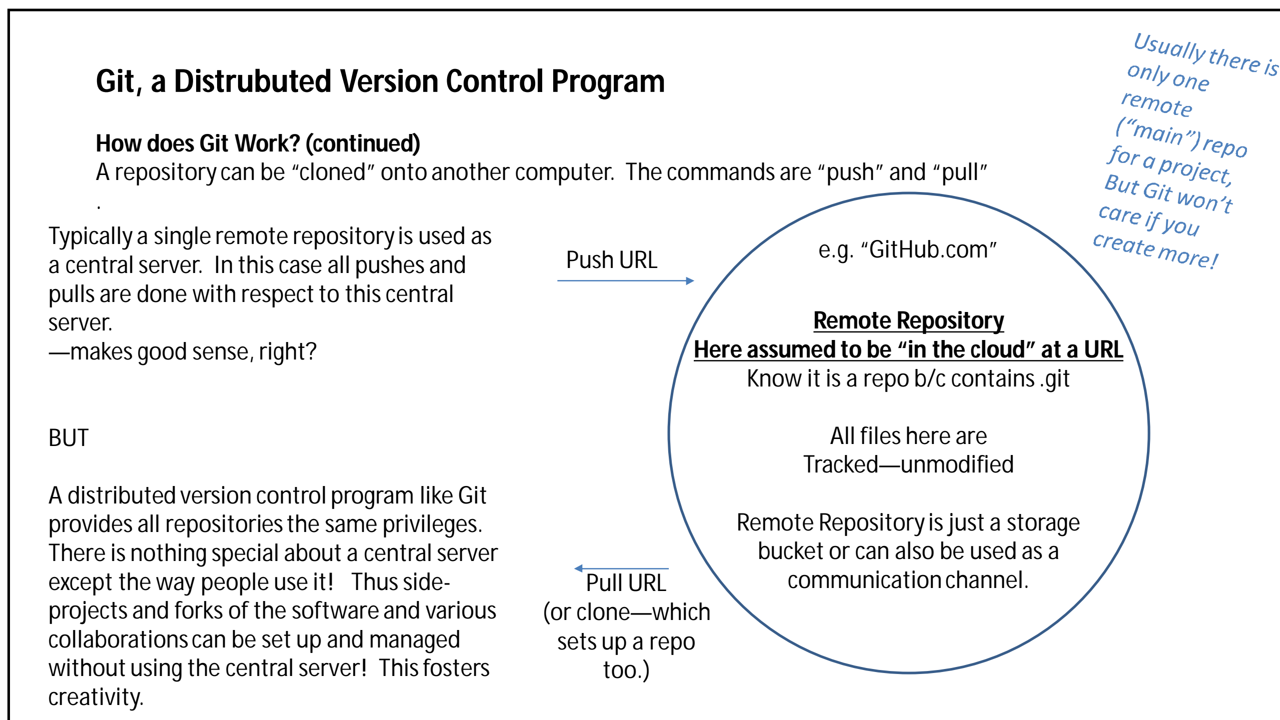
9



10



11



12

A key difference between centralized and distributed version control systems now becomes apparent:

In a centralized system the **central repository is the only known-good copy of the project**, save for the most recent edits that have not yet been committed. All software builds must be done from the central repository. There is going to be lots of network traffic to and from the repository. On bigger projects network delays can slow down file access. You better have good backups of the central repository because if it is corrupted or lost, the whole project is lost—save for the checked-out files.

In a distributed system users typically start their work by cloning the main repository. **Every client has local copies of all the files and complete privileges with all the files.** Network traffic is low. File accesses are fast. The discipline of periodically pushing edits back to the main repository and pulling those into client repositories keeps everyone up-to-date. If any repository is corrupted or lost, it can be recovered from any other repository—sans the most recent local edits in the corrupted or lost repository.

13

Git—Branch

In the bad old days, before version control software could help us. . .

Suppose you have some software project that is essentially working but missing a few little features yet.
As you work on these few little features you do not want to accidentally break the code so. . .
Before going to work on the new feature you set up a new folder and copy all the project files into it.
Then you develop the new feature in that separate folder.
Finally, when you are sure the new feature works, you find the new files and changed files and go back to
The original folder and insert these new files and make any other needed changes.
This way the original code is available to fall back on at any time up to the final commit of the "few little features."

Git uses the concept of a "branch" to manage this kind of work, but. . .
A branch avoids wholesale copying of all the project files—it only keeps track of changes and additions relative to the master set of files. (Relative to the "master branch.")
A "branch" is essentially a pointer (a name) to a particular snapshot (commit) of the project.
Edits to existing files and new files are now associated with that branch.
If you go back to the master branch, all these edits in the branch are ignored. The master remains available.
When the "little features" are ready, you can merge the branch into the master.
This is better than the bad old days because now you can roll back along either the branch or the master.
You can have many branches in play at once and roll back on any of them.

14

Demonstrated setting up a Git Repo

Unix/Linix commands used were. . .

pwd	Print working directory
cd <filepath>	Change directory to <filepath>
ls	List all files in the current directory
Ls –l	List-long. List all files in the current directory giving file details.
git init	Convert the current directory to a Git repository.
git status	Report on files that are untracked, staged, or modified.

Demonstration:
Created folder c:\Practice_Repo (case sensitive in Linux)
Made up file ABC.txt (untracked)
Status shows it is untracked.
Added the file to the repo
Status shows it is staged.
Made up file 123.txt (empty)
Status shows 123.txt as untracked.
Commit
Status shows 123.txt remains untracked

15