Part 2

Version Control and Git

Contents

2.1 A note on notation

The text in a yellow box is a set of commands that you can type into the terminal in your virtualbox. However, it is important that you understand that your present working directory is important for some commands. Where possible I have included change-directory (`cd`) commands to make sure you are in the right directory to issue a command that is location sensitive. However you should start to pay attention to your present working directory by using the following command

```
pwd
```

which will report the present working directory, i.e. where the terminal thinks your focus is in relative to the whole file system.

Be alert when you see [SOME CAPITALIZED TEXT IN SQUARE BRACKETS]: in the following there are several places where some part of the text for a Linux terminal command must be supplied by you.

For instance a command that involves your VT NETID will require you to use substitute your actual NETID. When you see `[NETID]` in a yellow box command it means you should ignore the square brackets and replace NETID with your actual VT NETID.

Hopefully when you see `[SOME CAPITAL TEXT IN SQUARE BRACKETS]` it will be clear what text should be used in its place. Don't forget to lose the square brackets !

2.2 Introduction

Git is a distributed source code management system (wiki). Key things to know about Git:

- Git was created by Linus Torvalds to help with team development of his Linux project.
- It is becoming a standard mode for sharing and collaborating on coding projects.
- The word "git" is an english-ism for an unpleasant person and according to the Git wiki page, Torvalds may have named Git after himself. Sigh.
- There are two parts to Git: a server that stores files for a project (repo or repository) and an app that interacts with the server to retrieve and possibly modify the files.
- More than one user can access and modify the contents of a repository if they have sufficient access rights. This allows multiple users to collaborate on a shared project.

In this class, we will use GitHub online repository system to manage all course computer codes. I have already set up a public repository where class example source code and other computer codes will be hosted. You can view this via the GitHub web interface: https://github.com/AliKarakus/me489

In the following we will discuss how to:

- Create a private Git repository on GitHub web interface.
- Install the Git command line application in your virtualbox.
- Create and add a secure shell key (ssh key) to your private Git repo.
- Clone the repository from the GitHub server onto your virtualbox.
- Navigate the local copy of your Git repo on your virtualbox's file system.

- Create and modify files in your local copy of the repo.
- Add your changes to the local repo via the Git command line application.
- Commit your changes to the local repo via the Git command line application.
- Push your changes from the local repo to the GitHub server.

To create a new GitHub repo click on the "New Project" button and follow the instructions to create a private repository. You should name it: me489. Make sure that you specify it should be a "Private" repository and also click the "Initialize repository with a README" box.

2.3 Installing the Git command line tool

Since we will spend most of our time developing code from within the terminal command line it makes sense to interact with the Git app from the command line too.

First things first, if you have not already done so start your virtualbox and launch a terminal. Next you can easily install the Git command line app using the `apt` command:

```
# install the Git command line tool
# using the apt command
# which requires super user privilege provided by super-user-do (sudo)

sudo apt install git
```

Note: useful tutorial on using `apt-get` here.

2.4 Cloning the repo from the GitHub server to your virtualbox using the command line

Now that we have set up the online repo and also set up password-free access it is time to "clone" the repository from the online server to your virtualbox. It is helpful to reflect on this for a few minutes. Right now the virtualbox running on your laptop is completely unaware of anything to do with the existence of the online repository or its contents. We need to use the Git command line tool `git` to copy the contents of the online repository to the disk storage associated with the virtualbox on your laptop. To do this you should first create a directory where your local copy of the repo will reside, then use the `git` command line app to clone the repo

```
# navigate to your home directory
cd ~/

# create a directory where your git repositories will live
```

```
mkdir class
mkdir class/git

# navigate into new directory
cd class/git

# clone the repo
git clone https://github.com/*YourAccount*/me489.git

# check to see if repo was actually cloned
ls
```

If the repo directory does not show up when you type `ls` then seek help. Any number of things may have happened to prevent successful cloning.

2.5 Navigating the local copy of your repo from the terminal command line

The good news is that the local clone is really just a regular Linux directory with sub-directories. You can use the standard terminal commands to navigate within the clone. For instance

```
cd ~/class/git/ME489
ls
mkdir gitClass
cd gitClass
emacs testFile.text
```

When you save that file you will have made a new subdirectory that contains a text file. It is located within the directory structure of the local clone of the online repo. But it is important to recognise that although the file is contained within the directory structure, Git does not consider it part of the repository.

You can query what Git thinks the status of the repo is with the following terminal command (assuming that your current working directory is located within the directory structure of the local clone)

```
git status
```

You should see that the `testFile.text` is included on a list of “Untracked files”.

Go ahead and also check to see if the file shows up under the online interface by pointing your browser. That directory you created should not show up (at least not yet) since we need to explicitly add the file to the repo.

2.6 Git Tutorials

For additional tutorials see for instance the cheat sheets:

- GitHub: https://github.github.com/training-kit/downloads/github-git-cheat-sheet.pdf.
- BitBucket: https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet
- GitLab: https://about.gitlab.com/images/press/git-cheat-sheet.pdf
- KeyCDN: https://www.keycdn.com/blog/git-cheat-sheet
- Nina Jaeschke: https://rogerdudler.github.io/git-guide/files/git_cheat_sheet.pdf
- **Scott Chacon and Ben Straub's priceless text on working with Git** https://git-scm.com/book/en/v2.

You can also use your favorite search engine to find other tutorials on using Git.

2.7 Summary of Git commands so far

In Table 2.1 we give a list of common use cases for the Git command line tool.

Action	`git` commands
Clone repo	`git clone [URL OF REPO]`
Add file change to local repo	`git add [ FILE NAME]`
Commit change to local repo	`git commit -m '[NOTE TO ATTACH TO COMMIT]'`
Push committed changes to remote repo	`git push`
Update local repo with remote changes	`git pull`
Create a new branch	`git branch [NAME OF BRANCH TO CREATE]`
Switch to a different branch	`git checkout [NAME OF BRANCH TO SWITCH TO]`

Table 2.1: Summary of commonly used Git command line tool commands.

In the next lecture we will practice interacting with the online Git repo, present an overview of the branching structure of a git repository, and perhaps even try a merge. Review the cheat sheet prior to the lecture.

2.8 Basic Git Workflow: Edit–Add–Commit

The basic workflow for using Git to track changes to your files is one of *edit–add–commit*:

1. *Edit* your files.
2. *Add* the changes to Git's *index*.

3. *Commit* the changes staged in the index to the repository.

```
$ emacs file.txt
$ git add file.txt
$ git commit
```

Forgetting step 2 (“add”) is a common pitfall, especially for people used to version control systems like CVS and Subversion, which have no analogue. It is good practice to use `git status` before committing to make sure you have added all desired changes to the index:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   file.txt

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:   anotherFile.txt
```

This output shows that our changes to `file.txt` have been added to the index but that those to `anotherFile.txt` have not. Thus, if we next run `git commit`, the changes to `file.txt` will be committed, but those to `anotherFile.txt` will not.

If you add changes to the index and later decide that you do not want them to be part of your commit, you can un-stage them with `git reset`. Continuing the example from above, we can unstage the changes to `file.txt` as follows:

```
$ git reset file.txt
Unstaged changes after reset:
M anotherFile.txt
M file.txt
```

(The `M` indicates that the file was modified.) Now `git status` shows

```
$ git status
On branch master
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:   anotherFile.txt
```

```
modified:   file.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Both files have been modified, but changes to neither have been staged for commit.

To see what changes have been made since the last commit, you can use `git diff`:

```
$ git diff
diff --git a/anotherFile.txt b/anotherFile.txt
index 9ca9a87..7603f14 100644
--- a/anotherFile.txt
+++ b/anotherFile.txt
@@ -1 +1,2 @@
 This is another file.
+This is a new line in the other file.
diff --git a/file.txt b/file.txt
index 4dd1ef7..685e488 100644
--- a/file.txt
+++ b/file.txt
@@ -1 +1,2 @@
 This is a file.
+This is a new line in the file.
```

This outputs the differences in the files in the unified diff format. Lines beginning with a `+` have been added since the last commit; lines beginning with a `-` have been deleted. Note that `git diff` only shows changes that have not been staged (added to the index). To see staged changes, use `git diff --cached`:

```
$ git add file.txt
$ git diff
diff --git a/anotherFile.txt b/anotherFile.txt
index 9ca9a87..9b6f183 100644
--- a/anotherFile.txt
+++ b/anotherFile.txt
@@ -1 +1 @@
-This is another file.
+This is a new line in the other file that replaces the old one.
$ git diff --cached
diff --git a/file.txt b/file.txt
index 4dd1ef7..685e488 100644
--- a/file.txt
+++ b/file.txt
@@ -1 +1,2 @@
 This is a file.
+This is a new line in the file.
```

If you've changed a file but decide that you do not want to keep the changes, you can use `git checkout` to discard them:

```
$ git checkout anotherFile.txt
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

modified:   file.txt
```

Notice that `git status` no longer lists `anotherFile.txt` as modified.

When you run `git commit`, Git will open a text editor (usually `vi` by default) to let you write your commit message. The standard format for a Git commit message is to begin with a single short (less than 80-character) line summarizing the changes in the commit, followed by a blank line, and then more thorough description as needed. Get in the habit of writing good, informative commit messages. Your collaborators (including your future self) will thank you, as good commit messages can make it much easier to trace the history of development of a piece of code by revealing what the developer was thinking when the changes were checked in.

2.9 Looking Through History

Git has a number of commands for viewing the commit history of a repository. The most basic of these is `git log`:

```
$ git log
commit cae7fa52ce9d58aba4f743ff15f6fafb684da4d6
Author: J. D. Hokie <jdh@vt.edu>
Date:   Tue Sep 3 20:23:07 2019 -0400

    Add numbered headers to files.

commit 9759687249220a7e079eeab185c9f10c76775351
Author: J. D. Hokie <jdh@vt.edu>
Date:   Tue Sep 3 20:22:35 2019 -0400

    Add yet another file.

commit 16db20ec863d28215fd9532133bbf6d35234b604
Author: J. D. Hokie <jdh@vt.edu>
Date:   Tue Sep 3 20:21:20 2019 -0400

    Add a new line to the first file.

commit ffef1cf6368a65aec94ff34da53369e32e663191
Author: J. D. Hokie <jdh@vt.edu>
```

```
Date:   Tue Sep 3 16:47:31 2019 -0400

    Add another file.

commit 0bd6c8e9f7370881de6526db9745c54eefbd2caf
Author: J. D. Hokie <jdh@vt.edu>
Date:   Tue Sep 3 15:37:53 2019 -0400

    Initial commit.
```

The history begins with the most recent commit and proceeds backward in time. Each entry shows the commit author's name and e-mail address, a timestamp with the date of creation, and the message that the author wrote when the commit was made.

The long hexadecimal string at the beginning of each log entry is a 160-bit SHA-1 hash that uniquely identifies the commit. These hashes can be used throughout Git to refer to commits. You can abbreviate a hash to its first few characters if that is enough to identify it. As an example, if you want to see the full log entry, including the patch, associated with the third commit above (`16db20ec`), you can do

```
$ git show 16db20ec
commit 16db20ec863d28215fd9532133bbf6d35234b604
Author: J. D. Hokie <jdh@vt.edu>
Date:   Tue Sep 3 20:21:20 2019 -0400

    Add a new line to the first file.

diff --git a/file.txt b/file.txt
index 4dd1ef7..685e488 100644
--- a/file.txt
+++ b/file.txt
@@ -1 +1,2 @@
 This is a file.
+This is a new line in the file.
```

The `git show` command can also show the state of a file at the time of a given commit, e.g.:

```
$ git show 16db20ec:file.txt
```

Just as `git diff` can be used to view changes made since the previous commit, it can also be used to show changes made between two commits. For instance, to see the changes between `16db20ec` and the current commit you can do:

```
$ git diff 16db20ec..HEAD
diff --git a/anotherFile.txt b/anotherFile.txt
```

```
index 9ca9a87..ece730d 100644
--- a/anotherFile.txt
+++ b/anotherFile.txt
@@ -1 +1,3 @@
+FILE 2
+------
 This is another file.
diff --git a/file.txt b/file.txt
index 685e488..a89ec53 100644
--- a/file.txt
+++ b/file.txt
@@ -1,2 +1,4 @@
+FILE 1
+------
 This is a file.
 This is a new line in the file.
diff --git a/yetAnotherFile.txt b/yetAnotherFile.txt
new file mode 100644
index 0000000..a4a93e0
--- /dev/null
+++ b/yetAnotherFile.txt
@@ -0,0 +1,3 @@
+FILE 3
+------
+This is yet another file.
```

The identifier `HEAD` is shorthand for the commit at the head of the currently checked-out branch (see the section on branching below).

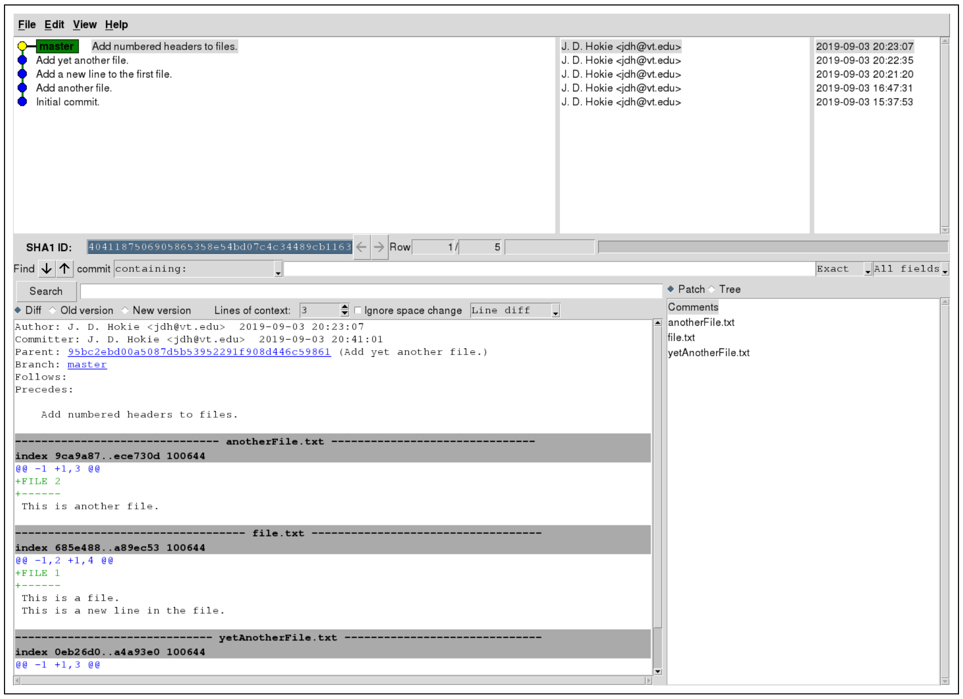
Git's command-line tools for navigating history are useful, but it can also be nice to have a way to view history graphically, especially when the history branches frequently. Git ships with a graphical history viewer written in Tcl called `gitk`.

Each commit on the current branch is displayed with the `HEAD` commit at the top. The text next to each commit in the graphical history window is the first line from the commit message. Below the history window, one can see the currently-selected commit's SHA-1, and below that, the full commit message and diff, along with a list of the files that have been modified.

Note that `gitk` is *not* a graphical interface to Git! Such interfaces do exist, but we discourage their use. Git was designed to be used from the command line, and that is the most popular way to use it. Many of Git's more advanced features are only accessible via the command line. The command line interface to Git has the added advantage of being scriptable, enabling easy automation of development tasks in ways that graphical interfaces do not support.

2.10 Sharing Changes

Unless you work in isolation, at some point, you will need to share the changes you make to a codebase with others. You will also need to be able to receive receive changes others have made and incorporate



```
master  Add numbered headers to files.    J. D. Hokie <jdh@vt.edu>  2019-09-03 20:23:07
Add yet another file.                     J. D. Hokie <jdh@vt.edu>  2019-09-03 20:22:35
Add a new line to the first file.         J. D. Hokie <jdh@vt.edu>  2019-09-03 20:21:20
Add another file.                         J. D. Hokie <jdh@vt.edu>  2019-09-03 16:47:31
Initial commit.                           J. D. Hokie <jdh@vt.edu>  2019-09-03 15:37:53

SHA1 ID: 4041187506905865358e54bd07c4c34489cb1163   Row 1 / 5
Find commit containing:   Exact   All fields
Lines of context: 3   Line diff
```

```
Author: J. D. Hokie <jdh@vt.edu>  2019-09-03 20:23:07
Committer: J. D. Hokie <jdh@vt.edu>  2019-09-03 20:41:01
Parent: 95bc2ebd00a5087d5b53952291f908d446c59861 (Add yet another file.)
Branch: master
Follows:
Precedes:

    Add numbered headers to files.

------------------------------ anotherFile.txt ------------------------------
index 9ca9a87..ece730d 100644
@@ -1 +1,3 @@
+FILE 2
+------
 This is another file.

---------------------------------- file.txt ----------------------------------
index 685e488..a89ec53 100644
@@ -1,2 +1,4 @@
+FILE 1
+------
 This is a file.
 This is a new line in the file.

----------------------------- yetAnotherFile.txt -----------------------------
index 0eb26d0..a4a93e0 100644
@@ -1 +1,3 @@
```

```
Comments
anotherFile.txt
file.txt
yetAnotherFile.txt
```

Figure 2.1: The `gitk` graphical Git history viewer.

them into your own copy of the code. Git's features for sharing changes to code are flexible and powerful and are one of the two major reasons Git has taken the world of open-source development by storm (the other being its facilities for branching, which we introduce briefly in the next section).

In this course, we will stick to the basics. You will share your code with the course instructor and teaching assistants by putting them in the repository you created. To get code from your laptop to repo, the basic command is `git push`:

```
$ git push
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 359 bytes | 359.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/AliKarakus/me489.git
   4041187..d501a5f  master -> master
```

Later in the course, you will also need to know how to get changes from GitHub to your laptop. The basic command for this is `git pull`:

```
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
```

```
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/AliKarakus/me489.git
 * branch            HEAD       -> FETCH_HEAD
Updating 4041187..d501a5f
Fast-forward
 anotherFile.txt | 1 +
 1 file changed, 1 insertion(+)
```

Note that you cannot push new changes to a repository unless you have already pulled the latest version of the files being tracked from it already. If you forget to do this, and if the repository to which you are trying to push your changes has other changes that you do not yet have, you will get an error like this:

```
$ git push
To https://github.com/AliKarakus/me489.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/AliKarakus/me489.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

To fix this, do `git pull` first; then retry `git push`:

```
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/AliKarakus/me489.git
   d501a5f..b932eba  master     -> origin/master
Merge made by the 'recursive' strategy.
 yetAnotherFile.txt | 1 +
 1 file changed, 1 insertion(+)
$ git push
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 580 bytes | 580.00 KiB/s, done.
Total 5 (delta 2), reused 0 (delta 0)
To https://github.com/AliKarakus/me489.git
   b932eba..3a51a4e  master -> master
```

You may be prompted for a commit message for a merge commit.

Notice that in neither of these commands did we specify where to push our data or from where to pull it. The reason is that Git remembers the URL of your repository on GitHub from when you cloned it in the previous lecture. From the perspective of the repository on your laptop, your repository on GitHub repository is what Git calls a *remote repository*, or just a *remote* for short. To see a list of all the remotes of which your current repository is aware, you can use `git remote -v`:

```
$ git remote -v
origin https://github.com/AliKarakus/me489.git (fetch)
origin https://github.com/AliKarakus/me489.git (push)
```

This output shows that you have a remote named `origin` that refers to your repository on GitHub. The two URLs shown are the locations from which Git will retrieve ("fetch") changes and to which Git will send ("push") changes when interacting with this remote. The two URLs are almost always the same; in rare circumstances, it can be beneficial for them to be different.

The `git push` command shown above is actually shorthand for

```
$ git push origin master
```

which instructs Git to push commits from the `master` branch on your laptop's repository to the `master` branch of the repository specified by the `origin` remote, i.e., your GitHub repository. Likewise

```
$ git pull origin master
```

tells git to pull changes from the `master` branch on GitHub into the currently checked-out branch on your laptop.

It is considered good practice to use the long forms of these commands, specifying both the remote and the branch name explicitly. This greatly reduces the likelihood of accidentally sending (receiving) changes to (from) the wrong place when you work with multiple remotes.

2.11 Branching Basics

What do we mean by the `master` branch? In the language of version control systems, a *branch* refers to a line of code development. In this class, you have only one line of development to worry about—that associated with your own coursework—but in large projects with multiple developers, it is common to have tens, hundreds, or even thousands of lines of development that are active at once on the same codebase. Modern version control systems all have facilities for supporting multiple branches.

In Git, branches are particularly simple: they're essentially just references to specific commits. A branch "points" to a commit, and when a new commit is made "to" that branch, two things happen. First, the commit to which the branch currently refers is made the immediate ancestor of the new commit. Second, the branch is updated to point to the new commit.

When a Git repository is first initialized, it comes with a default branch named `master` that is set to the current branch. Many (but not all) software projects that use Git for version control retain this branch and use it to refer to the most recently released or "stable" version of the software or to code that will be in the next release.

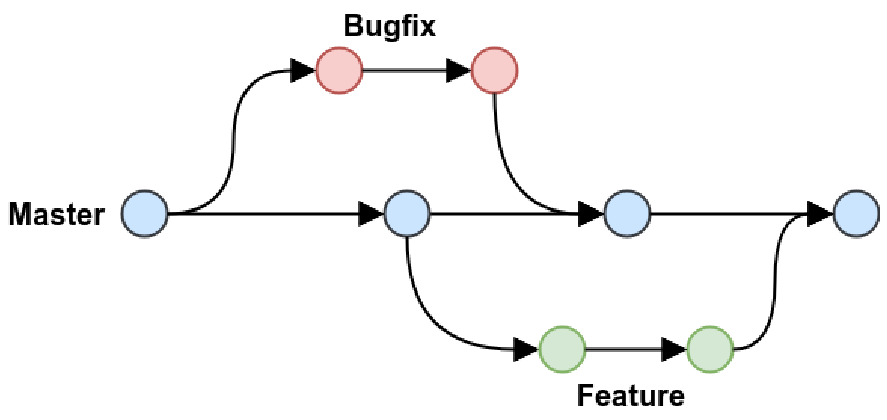


Figure 2.2: Basic branching

As mentioned above, in this class, you have only one line of development, and you will be making all of your commits to your repository's `master` branch, but as real-world codebases frequently have multiple branches, it is useful to know the basics for how to work with branches in Git.

You can see what branch you are currently on with `git status`:

```
$ git status
On branch master
nothing to commit, working tree clean
```

The output clearly states that we are on the `master` branch.

You can see a list of all the branches in your repository with `git branch`

```
$ git branch
  experimental
* master
```

This repository has two branches: `experimental` and `master`. The current branch is indicated by an asterisk (*).

The branch that's currently active is said to be "checked out". To check out a different branch, use `git checkout`:

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

When you do this, the files in your repository will automatically change to whatever state they were in on the commit at the head of that branch.

You can create a new branch and check it out with `git checkout -b`:

```
$ git checkout -b newBranch
Switched to a new branch 'newBranch'
```

The head of the new branch will be whatever commit is currently checked out. If you want it to be somewhere else, you can specify it by its SHA-1:

```
$ git checkout -b anotherBranch 40411875
Switched to a new branch 'anotherBranch'
```

To merge changes from one branch into another, the command is `git merge`. This command merges changes from a given branch into the branch that is currently checked out. To merge changes from a branch named `experimental` into the `master` branch, you do:

```
$ git checkout master
Switched to branch 'master'
$ git merge experimental
Updating 3a3637a..ee3580a
Fast-forward
 file.txt | 1 +
 1 file changed, 1 insertion(+)
```

In this case, the output of `git merge` indicated that this was a *fast-forward* merge. This means that the head of the branch named in `git merge` was a descendent of the checked-out branch into which the merge was made. In this case, the merge operation is trivial: Git just redefines the `master` branch to point to the same commit as the `experimental` one.

For merges that are not fast-forward merges, one must make a *merge commit* to join the history of the two branches. When this happens, Git will prompt you for a commit message. The body of the message will already be filled in; you can replace it if you choose.

What happens if both branches have commits that modify the same file in different ways? In this scenario, Git will try to figure out whether the changes clash with one another and merge them automatically. If it cannot do this, you will get a message indicating that the changes conflict:

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

In this case, you will need to edit the files with conflicting changes, add the changes to the index with git add, and then use git commit to complete the merge. Git will mark the parts of the files showing where there were conflicts that it could not resolve.

2.12 An Extended Example

The following is an example of how Git is used in a typical real-world development scenario. Your work in this class will follow a similar workflow except that as you are working on your own, you can do all your work in a single line of development without branches (or, rather, using only the master branch) if you choose.

Alice's boss has asked her to implement a new feature in their codebase. As the new feature is likely to be unstable, she's been asked to do her development in a new branch to keep it separate from the main codebase being tracked by the project's master branch. Alice creates a new branch based off of master and does some work:

```
$ git checkout -b new-feature master
$ emacs newFeature.txt
$ git add newFeature.txt
$ git commit
```

Alice reports back to her boss, saying that the new feature is finished. Her boss says that before it can be incorporated into the stable master branch, it needs to undergo review by the project lead. Alice pushes her branch to the project's central repository so the lead can see it.

```
$ git push origin new-feature
```

The project lead makes a few small changes and then asks Alice to make some others. Alice first makes sure she's on the branch for her feature and then pulls the lead's changes into her repository. She then does the additional work and pushes it back for further review:

```
$ git checkout new-feature
$ git pull origin new-feature
$ emacs newFeature.txt
$ git add newFeature.txt
$ git commit
$ git push origin new-feature
```

The project lead is now happy with her code and asks Alice to incorporate it into the master branch. She switches to the master branch, makes sure she has pulled any updates that have been pushed to it in the meantime and then merges her feature branch into it with git merge:

```
$ git checkout master
$ git pull origin master
$ git merge new-feature
```

Finally, she pushes the newly updated `master` branch to her company's central repository and deletes the local and remote versions of her feature branch, which are no longer needed:

```
$ git push origin master
$ git push origin :new-feature
$ git branch -d new-feature
```

2.13 Miscellaneous Tips

Git has many useful features—far too many to cover comprehensively in these notes. Here are a few additional features that may be useful to you in this course.

2.13.1 Ignoring Files

Often, you have files that you do not want Git to track. You can get Git to ignore these files by creating a file named `.gitignore` in your repository and adding the files that you do not want tracked to it, one per line. Wildcard patterns are accepted; see `man gitignore` for details about the syntax.

As a basic example, this `.gitignore`

```
*.o
myProgram
```

will ignore the file `myProgram` and all object files ending in with the generated by a compilation process. Why track these when they can be regenerated from the (tracked) source files?

Patterns specified in a `.gitignore` file apply to the directory in which the `.gitignore` is located any any subdirectories. Note that the `.gitignore` file is trackable by Git; this is useful for distributing project-wide ignore rules to all developers.

2.13.2 Tracking Empty Directories

Git tracks files, not directories. You cannot add a directory to a repository unless it has a file in it. If you just want to set up a directory structure and have Git track it, one way to accomplish this is to create a single empty file in the directory, conventionally named `.KEEP` or `.gitkeep`, and commit that file.

2.13.3 Deleting, Moving, and Renaming Files

The important thing to remember when deleting, moving, or renaming files is that each of these operations is a change that must be added to Git's index before it can be committed. For instance, to delete a file, you can use `rm` like usual, followed by `git add`:

```
$ rm file.txt
$ git add file.txt
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

deleted:    file.txt
```

This two-step sequence is a little awkward—it feels weird to `git add` a file that we've deleted—but there is a shortcut; you can use `git rm` instead:

```
$ git rm file.txt
```

Similarly, to move or rename a file, you can use `mv` followed by `git add`, or you can use `git mv` to do both at once.

2.13.4 Undoing a Commit

If you need to undo a commit, the command to use is `git revert`. For example, if commit `b932eba4` turned out to be a mistake, you can do

```
$ git revert b932eba4
```

This command works by creating a new commit that is the "inverse" of the old commit. As such, when you run this command, you will be prompted for a commit message, giving you some a place to explain the need for the reversion.

2.13.5 Getting Help

If you can't remember the options for a given command and need a quick reminder, the Git manual pages are a good place to look. The manual pages are named after the commands with spaces replaced by hyphens; for instance, to retrieve the manual page for `git add`, type `man git-add`.

The top-level manual page (`man git`) has, among other things, a list of all the Git commands, including many that we have not discussed here. Other manual pages of interest include the Git tutorial at `man gittutorial` (and its sequel `man gittutorial-2`), the glossary at `man gitglossary`, and the summary of "everyday" Git commands and workflows at `man giteveryday`.

2.14 Summary of Git Commands

Table 2.2 extends the summary table of Git commands from the previous lecture to include the new commands we have covered above.

Action	`git` commands
Clone repo	`git clone [URL OF REPO]`
Check repo status	`git status`
Add file change to local repo	`git add [FILE NAME]`
Commit change to local repo	`git commit -m '[NOTE TO ATTACH TO COMMIT]'`
Push committed changes to remote repo	`git push`
Update local repo with remote changes	`git pull`
Unstage a staged change	`git reset [FILE]`
Discard changes to a file	`git checkout [CHANGED FILE]`
View unstaged changes since the last commit	`git diff`
View staged changes since the last commit	`git diff --cached`
Delete a tracked file and stage the deletion	`git rm [FILE]`
Move a tracked file and stage the move	`git mv [FILE] [NEW FILE]`
Undo a commit	`git revert [COMMIT]`
View history of the current branch	`git log`
View the message and patch for a commit	`git show [COMMIT]`
View a file as it was on a particular commit	`git show [COMMIT]:[FILE]`
View difference between two commits	`git diff [OLD COMMIT]..[NEW COMMIT]`
Launch Git's built-in graphical history viewer	`gitk`
View remotes associated with local repository	`git remote -v`
Push committed changes to remote repo (full form)	`git push [REMOTE] [BRANCH]`
Update local repo with remote changes (full form)	`git pull [REMOTE] [BRANCH]`
Create a new branch	`git branch [NAME OF BRANCH TO CREATE]`
Switch to a different branch	`git checkout [NAME OF BRANCH TO SWITCH TO]`
Create and switch to a different branch	`git checkout -b [NEW BRANCH] [BRANCH LOCATION]`
Merge given branch into current one	`git merge [BRANCH TO MERGE]`

Table 2.2: Summary of commonly used Git command line tool commands.

2.15 Further Reading

The first three chapters of *Pro Git* by Scott Chacon and Ben Straub, available at https://git-scm.com/book/en/v2 contain all of the information you need for this class and then some. If you are interested in learning how Git works internally, the first five sections of Chapter 10 are also of interest.