# Answers/Hints for Labs in Learn Git in a Month of Lunches

Rick Umali

June 22, 2016

# Introduction

This document contains lab answers (or hints that will help you get to the answers). This document is available from:

http://tech.rickumali.com/learngit/

Please send any feedback to the book's forums, which I actively monitor:

https://forums.manning.com/forums/learn-git-in-a-month-of-lunches

# Chapter 1

No lab!

# Chapter 2

No lab!

# Chapter 3

1. Git configuration can be found in one of three places: /etc/gitconfig, $HOME/.gitconfig or .git/config.

   There is more information on this in Chapter 20, though that chapter may prove difficult at this stage, especially if you are new.

   `git config help` has complete information.

2. `git help git`

3. `git rebase`

4. Directed acyclic graph. See `git help glossary`.

5. Try `git help tutorial`.

6. Try it! They should.

7. To scroll one line at a time in less, use the up or down arrow keys, or the j or k keys. (There are other key combinations, and `less --help` lists them all.)

# Chapter 4

1. You should see `fatal: bad default revision 'HEAD'`. I have seen reports of the error `fatal: your current branch 'master' does not have any commits yet`.

2. Following the steps in this exercise, `git status` should give:

   On branch master

   Initial commit

   Changes to be committed: (use "git rm –cached ..." to unstage)

   ```
   new file:   file.txt
   ```

   Changes not staged for commit: (use "git add ..." to update what will be committed) (use "git checkout – ..." to discard changes in working directory)

   ```
   modified:   file.txt
   ```

   You should see file.txt listed twice.

3. The End of Line (EOL, aks newline) is the bane of computing. If you're on a Windows, most files end lines with CR+LF. If you're on a Unix/Linux machine (including Mac), most files end with LF. The Wikipedia page for Newline describes other variants.

   If you spend all your time on one platform, you will not encounter this issue. However, with Git, you may find yourself collaborating with platforms that are different from yours. Git attempts to be smart about this, with the core.autocrlf and core.safecrlf configuration settings.

   Read this post:

   http://stackoverflow.com/q/170961/10030

   on Stack Overflow for some detailed posts for how to handle EOL. This post contains recommendations for managing EOL settings, but if you're a beginner, I recommend skimming this material for now. Out of the box, Git will try its best to manage EOL properly.

# Chapter 5

1. file.txt will appear in the "Unstaged Changes" text pane.

2. `git citool` is `git gui`, with the argument `citool` (i.e. `git gui citool`). For more details, type `git gui --help`, and look at the COMMANDS section.

3. file.txt will appear in the “Staged Changes” text pane.

4. Following the instructions, you should see file.txt appear in both windows (see figure 1).

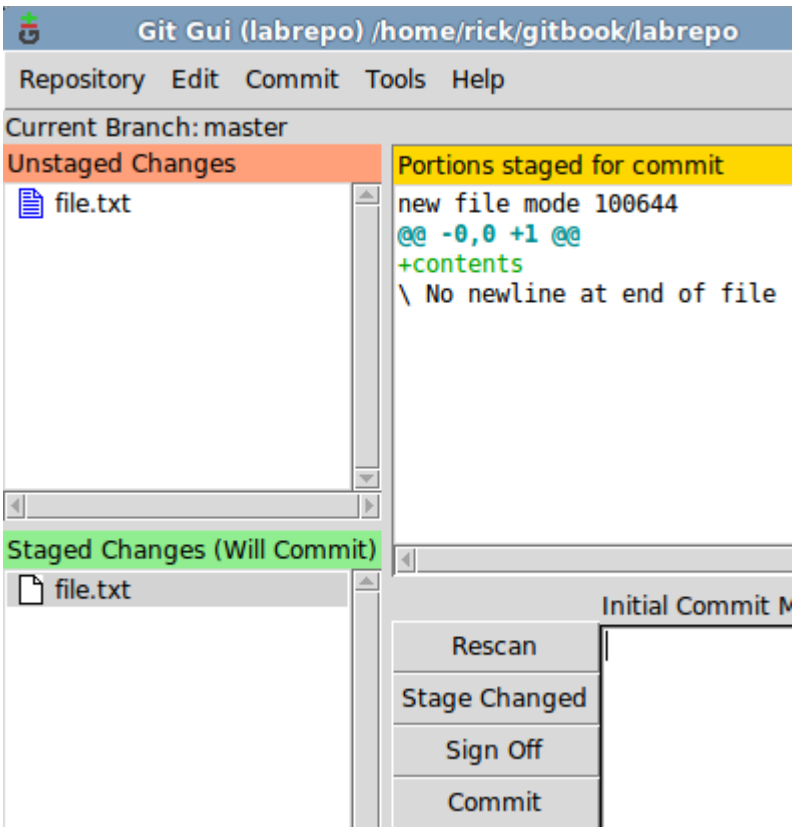



Figure 1: file.txt in both places

5. I’m afraid this is a bit of a trick question. The exercise has you adding a change (with `git add`), then adding another change on top of the same file. However, the instructions actually overwrite the first change with the second!

   The best way to save both is to run `git commit`. This will save the first version of the file. Then `git add` and `git commit` on the second change. This saves the second version of the file.

6. The chapter has us getting to the `gitk` program via `git gui`. It may be somewhat of a surprise that you can get to the `gitk` without `git gui`, by typing `gitk` by itself.

`gitk` has no `--help` switch, but you can read about the command line arguments you can pass into it by typing `git help gitk`.

# Chapter 6

## Ch 6.4.1

1. Instead of `--staged`, use `--cached`. See `git diff --help`.
2. `git add -n`
3. `cat -n FILENAME` will display the contents of FILENAME with line numbers.
4. `git log --format=oneline --abbrev-commit`
5. `--all`. Go figure!

## Ch 6.4.2

To see the command to get rid of a change that you've added to Git via `git add`, use `git status`.

## Ch 6.4.3

In your favorite editor, create a file readme.txt in the math directory. It can contain text or it can be empty. Now type:

```
git add readme.txt
git commit -m "Adding readme.txt"
```

# Chapter 7

## Ch 7.5.1

The instructions for this lab got truncated for some reason! This exercise was to make a Git repository containing the file lorem-ipsum.txt, and then to introduce some changes to that file (copying the contents of loreum-ipsum-change.txt on top of lorem-ipsum.txt).

If you decipher the instructions, the output from `git diff` will look like the following:

```
diff --git a/lorem-ipsum.txt b/lorem-ipsum.txt
index dd194e9..b3ee228 100644
--- a/lorem-ipsum.txt
+++ b/lorem-ipsum.txt
@@ -1,7 +1,7 @@
 Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc facilisis
 venenatis felis, scelerisque consectetur lacus viverra in. Phasellus
 scelerisque pretium nibh, ac porttitor massa vestibulum non. Aenean ornare
-convallis enim, et semper neque sagittis ut. Proin blandit dui vitae quam
+convallis enim, et change here. Proin blandit dui vitae quam
 fermentum, non lacinia lectus molestie. Vivamus tempus viverra dui. Cras in
 luctus felis. Pellentesque dictum tortor ipsum, id iaculis felis ultrices ut.
 Nunc id velit dignissim, suscipit est ac, euismod libero. Phasellus fringilla,
@@ -10,8 +10,7 @@ mi sagittis odio. Donec fermentum sem vel neque pulvinar eleifend.

 Nullam posuere condimentum rutrum. Vivamus vitae elit a tellus interdum
 accumsan. Duis pretium viverra condimentum. Curabitur dapibus aliquet semper.
-Aliquam pretium nunc aliquam pellentesque molestie. Suspendisse potenti. In a
-erat sed ante rutrum commodo. Fusce congue nunc aliquet tellus dignissim
+Aliquam pretium nunc deleted below molestie. Suspendisse potenti. In a
 fringilla.

 Cras non odio gravida, blandit risus sed, tempor lectus. Donec vel congue
@@ -28,6 +27,7 @@ sem convallis, nec vulputate ante ullamcorper. Nam laoreet, sapien in
 elementum, ante massa vehicula eros, non luctus ipsum leo a metus.

 Nunc velit lacus, pellentesque sit amet libero ac, rutrum facilisis est. Ut
+added a new line here do not be afraid added new line here do not be afraid
 aliquam justo sit amet ornare condimentum. Curabitur nibh est, vehicula at
 sodales non, posuere in quam. Quisque ornare rutrum arcu ut porttitor. Nunc
 blandit justo eget nisl accumsan, eu auctor urna pretium. Proin non condimentum
```

If your repository can produce the above `git diff` output, you're ready to tackle the questions in the exercise.

If you're stuck, the answers are below. See if you can work backwards from this!

1. There are three hunks (they are delimited by `@@` at the start of the line).
2. The second hunk's `@@` string should be "-10,8 +10,7". If you stare at the difference, you'll see that a line was deleted. To stage and commit this change, use:

   ```
   git add -p
   ```

   At this point, you'll be prompted with "Stage this hunk [y,n,q,a,d,/,K,j,J,g,e,?]?" for each hunk. For the first hunk, reply with 'n'. For the second hunk, reply with 'y'. For the third hunk, reply with 'n'.

At this point, `git status` will show that the file lorem-ipsum.txt has staged and unstaged changes. Commit the staged changes by typing `git commit -m "2nd hunk"`.

3. To check out the latest code, `git status` will suggest `git checkout -- <file>...` to discard changes in the working directory. Translate this to `git checkout -- lorem-ipsum.txt`.

   After typing this command, `git status` should report that the working directory is clean, with nothing to commit.

4. This step is a repeat of the setup instructions, but this time the file in our repository is different.

5. There are two hunks.

6. To commit the entire file, there's no need to select each hunk one by one. Just type `git add lorem-ipsum.txt`, then `git commit -m "All hunks"`.

## Ch 7.5.2

Deleting files in Git require `git rm`. In this exercise, I have you perform a `git rm`, but then I encourage you to revert the changes, following the `git status` commands. However, as I write this, reverting the file requires two commands: `git reset` and `git checkout`. Each time `git status` shows us the syntax.

My command line session for doing this follows:

```
% ls
lorem-ipsum.txt

% git rm lorem-ipsum.txt
rm 'lorem-ipsum.txt'

% ls

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

        deleted:    lorem-ipsum.txt


% git reset HEAD lorem-ipsum.txt
Unstaged changes after reset:
```

```
D       lorem-ipsum.txt

% ls

% git status
On branch 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:    lorem-ipsum.txt

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

% git checkout -- lorem-ipsum.txt

% ls
lorem-ipsum.txt

% git status
On branch master
nothing to commit, working directory clean
```

## Ch 7.5.3

The word 'form' is used in the Git documentation to indicate the pattern of the arguments.

The `git checkout -- math.sh` form is formally described in the `git checkout` as `git checkout [-p|--patch] [<tree-ish>] [--] <pathspec>...` Notice that the brackets indicate optional arguments. In our command, we omit the `--patch` and `[tree-ish]`. Knowing this form leads you to the right section of the manual to read.

The `git reset math.sh` form is described in `git reset` as `git reset [-q] [<tree-ish>] [--] <paths>...`. In the command we give, we omit the `-q` and the [tree-ish] arguments.

Both commands specify `--` as an optional argument. For both our commands, we omit `--`. These double-dashes are used to separate the tree-ish from the paths. The book doesn't discuss tree-ish, or a use-case involving `--`, but you can visit this Stack Overflow URL for more.

http://stackoverflow.com/q/13321458/10030

# Chapter 8

`git log` is a command that people should feel completely comfortable experimenting with. It's read-only, and it doesn't do any writing to the repository!

## Ch 8.5.1

1. `git log --reverse`

   In this chapter, we learned that each commit points back to its parent. `git log` has a `--parents` switch that helps facilitate viewing the parents. To me, this suggests that it is *much* easier for Git to generate the default `git log` listing, than it is to generate the listing required by the `--reverse` switch. This is more apparent on a very large repository. For the Git source code, `git log --reverse` pauses for a few seconds before displaying the first commit.

2. `git log -N` (where N is any number)

   This limits the `git log` output to N commits. `-N` is the shortest form of this command, which can also be expressed as `-n N` or `--max-count=N`.

3. `git log --relative-date`

4. `git log --pretty=oneline --abbrev-commit`

## Ch 8.5.2

Below is a session that follows the instructions in this exercise.

```
% echo "new line" >> readme.txt

% git add readme.txt

% git commit -m "One new line"
[master 0ce6b5a] One new line
 1 file changed, 2 insertions(+)

% git log -1
commit 0ce6b5aa98e69fc5f34122ff479fa68842a52987
Author: Rick Umali <rickumali@gmail.com>
Date:   Thu Nov 5 20:10:45 2015 -0500

    One new line

% git commit --amend -m "Fixed commit" -m "Second paragraph" -m "Wall of text"
```

```
[master bf176fd] Fixed commit
 Date: Thu Nov 5 20:10:45 2015 -0500
 1 file changed, 2 insertions(+)

% git log -1
commit bf176fd3acab541a84f1f6a35137b9ae61d789a2
Author: Rick Umali <rickumali@gmail.com>
Date:   Thu Nov 5 20:10:45 2015 -0500

    Fixed commit

    Second paragraph

    Wall of text
```

Given the output above, you can see that SHA1 ID is different from the first time you performed the commit. `git commit --amend` causes the SHA1 IDs to change!

## Ch 8.5.3

1. The first three commands return the third ancestor from the most recent commit on master.

   If your log looks like this (on the master branch):

   ```
   % git log --oneline -4
   c0fd19e A small update to readme.
   a853319 Adding printf.
   f646129 Adding two numbers.
   6a476a7 Renaming c and d.
   ```

   The command `git rev-parse master~3` returns:

   ```
   6a476a73bbb608c4efa508b7a13509b38f8c93f7
   ```

   Note that the rev-parse returns the full SHA1 ID. To mimic the display of `--oneline`, pass `--short` to the rev-parse command.

   Both `git show` commands will show the same output, because `master@{3}` and `master^^^` are synonyms.

2. The `git rev-parse :/"Removed a and b."` will search the commit logs, and return the SHA1 ID of the commit that contains the string "Removed a and b." This is a nifty trick, and we'll see a variant of this in Chapter 15.

3. Yes! Type `git rev-parse four_files_galore`, and you will see the SHA1 ID that this tag points to.

4. Yes, all the funky syntax of `git rev-parse` work on tag names as well. Type `git rev-parse four_files_galore^^`, and it will return the SHA1 ID that is commits before four_files_galore.

## Ch 8.5.4

Below is a session that follows the instructions in this exercise.

```
% git checkout four_files_galore
Note: checking out 'four_files_galore'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 84965d0... Adding four empty files.

% ls
a  b  c  d  math.sh

% echo -n "commit" >> d

% git add d

% git commit -m "One commit to d"
[detached HEAD 399cd3f] One commit to d
 1 file changed, 1 insertion(+)

% git checkout master
Warning: you are leaving 1 commit behind, not connected to
any of your branches:

  399cd3f One commit to d

If you want to keep it by creating a new branch, this may be a good time
to do so with:
```

```
 git branch <new-branch-name> 399cd3f

Switched to branch 'master'
```

From that last `git checkout` command, Git noticed that you were leaving a stray commit behind. In the next chapter, we'll show a better way for creating commits so that they're not strays!

## Ch 8.5.5

To delete a tag, use the `-d` switch to `git tag`. A session showing off this would look like the following.

```
% git branch
  another_fix_branch
* master
  new_feature

% git log --oneline -3
c0fd19e A small update to readme.
a853319 Adding printf.
f646129 Adding two numbers.

% git tag -m "Next to last" penultimate master^^

% git rev-parse penultimate
5edd4a3238c288c329126f0144a28df238ef275c

% git show penultimate
tag penultimate
Tagger: Rick Umali <rickumali@gmail.com>
Date:   Thu Nov 5 20:54:00 2015 -0500

Next to last

commit f6461290964d974e807d0fd3dbf1c041270dc3a4
Author: Rick Umali <rickumali@gmail.com>
Date:   Wed Nov 4 20:17:25 2015 -0500

    Adding two numbers.

diff --git a/math.sh b/math.sh
index 5bb7f63..dab42fb 100644
--- a/math.sh
```

```
+++ b/math.sh
@@ -1,3 +1,5 @@
-# Comment
+# Add a and b
 a=1
 b=1

% git tag
four_files_galore
penultimate

% git tag -d penultimate
Deleted tag 'penultimate' (was 5edd4a3)

% git tag
four_files_galore
```

We incorporated the funky syntax to pick a commit that is two commits behind master. If it wasn't clear earlier, notice that the `git rev-parse penultimate` command returns the SHA1 ID of the tag, but `git show` shows only the SHA1 ID of the commit that the tag points to.

## Ch 8.5.6

This exercise has you running a script that creates a repository that contains many commits. This repository provides another test area to try out the techniques from this chapter.

1. The very first commit says "The very first commit. Hi!" It should be five days old, meaning if you ran the script on October 8, the date of this first commit would be October 3.

2. This question required you to use `git rev-parse :/"ubiquitous"`. See 8.5.3 of this PDF to refresh your memory!

3. `git log --author=rgu`

   The translator for the Japanese version of the book offered the above concise solution.

   When I originally wrote this answer, I found the answer by using `git log`, and within the pager command, searching for the e-mail address, using "/". Remember that Git displays most of its output inside of a pager if it detects that the output will be more than a screen-ful of lines. You can search this output using the pager.

   You could also combine `git log` and the UNIX `grep` command to search for the answer! Try: `git log | grep -B 1 rgu@freeshell`.

4. `git log --since=yesterday`

5. Your `gitk` output should look something like figure 2, examining the most recent commit. Clicking on any file should show its contents in the various panes.

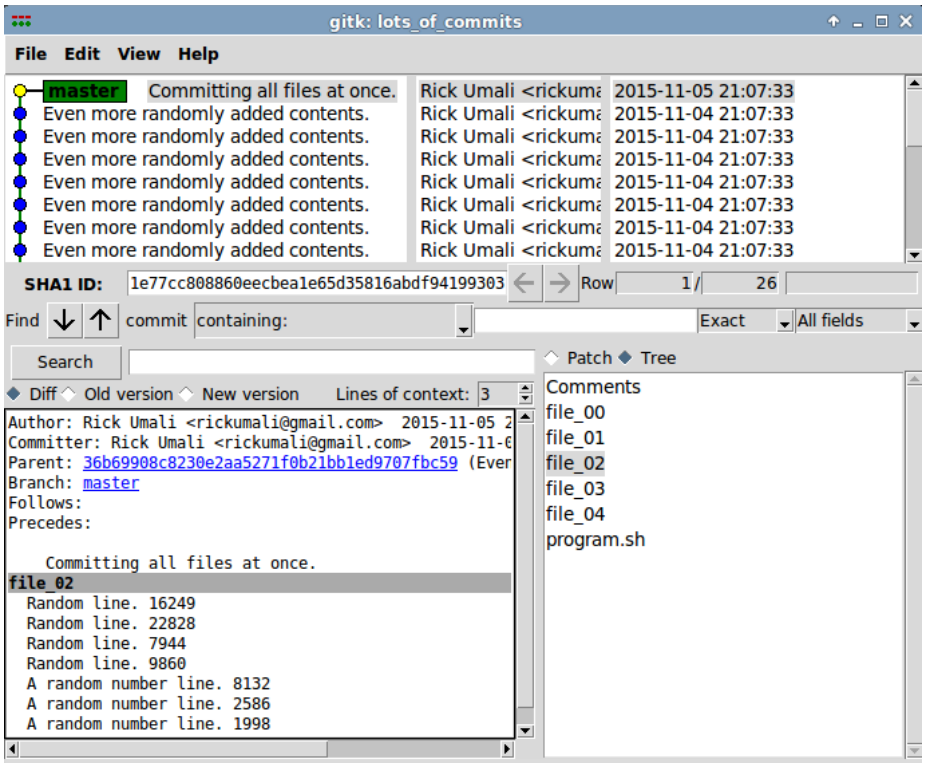


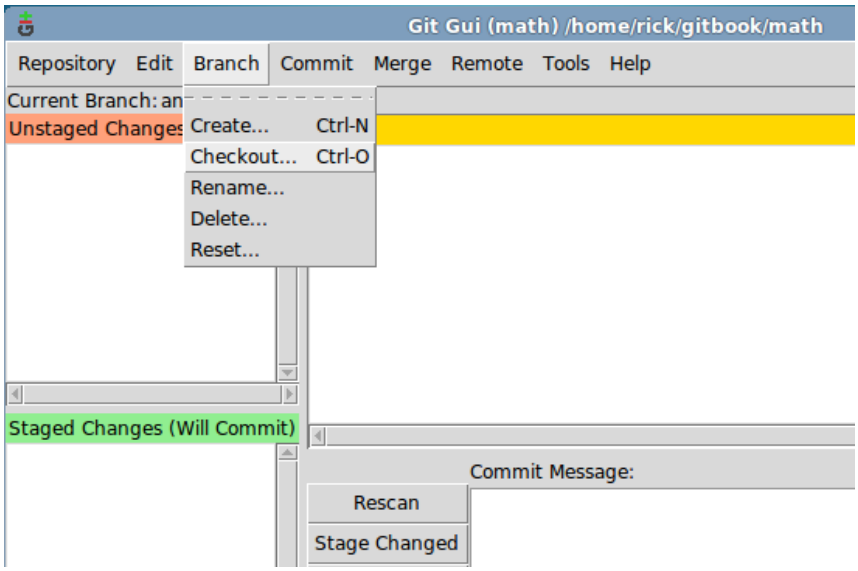

Figure 2: gitk and the lots_of_commits repository

# Chapter 9

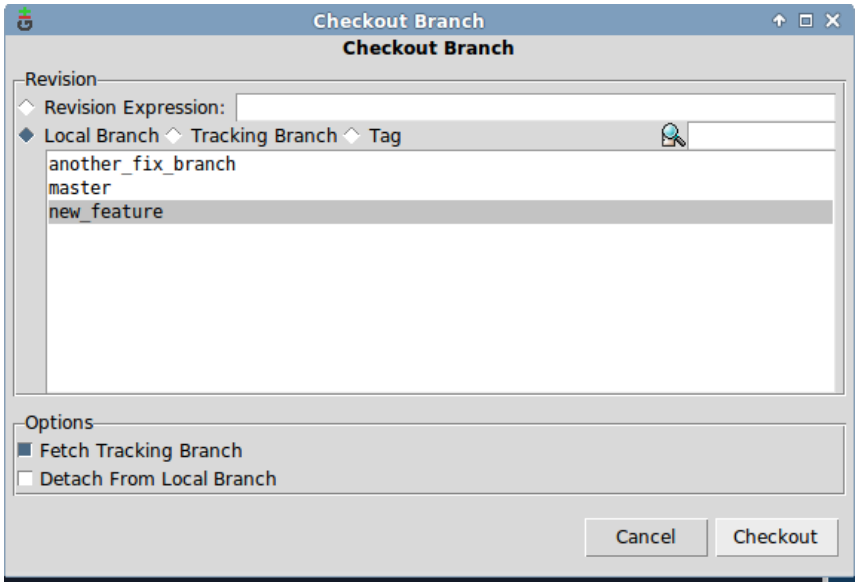
## Ch 9.5.1

1. Git GUI has a decent mechanism to check out branches.

   First, you select the Branch menu item, as in the following figure.




Then, on the resulting branch selector dialog window, the next figure, select "new_feature_branch", and click the Checkout button.

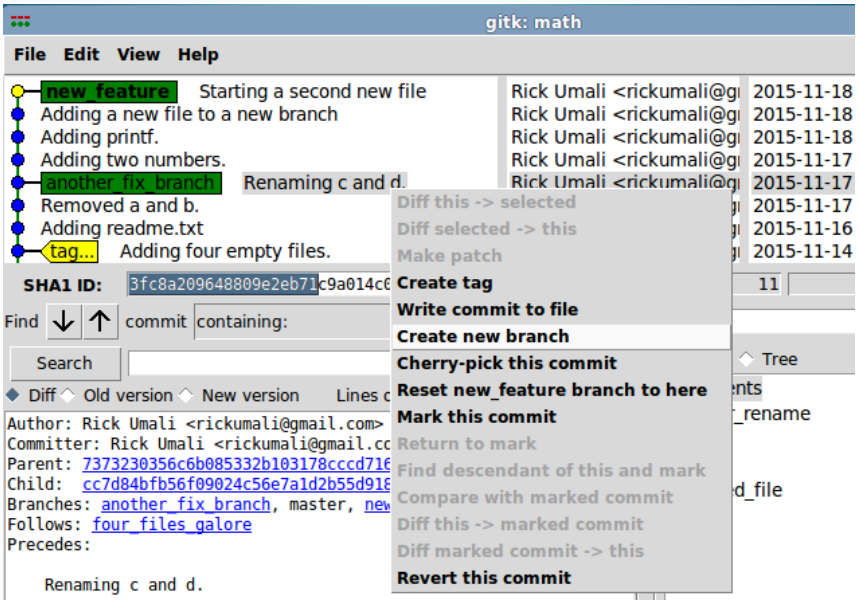



After doing the above steps, your repository will be in the new_feature_branch branch.

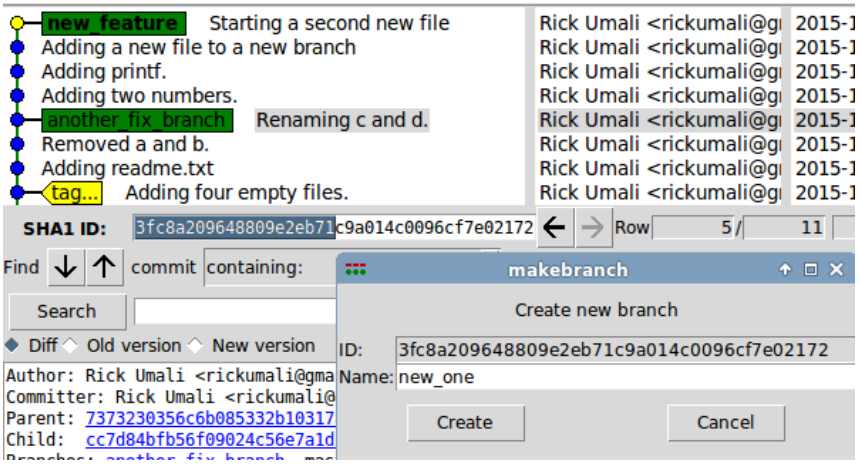
2. `gitk` has a mechanism for making branches off of an existing branch.

   First, use the right-mouse button click (context menu) on the commit text of the branch that you want to use (in this exercise, it was 'another_fix_branch'), as shown in the next figure. (Selecting the context

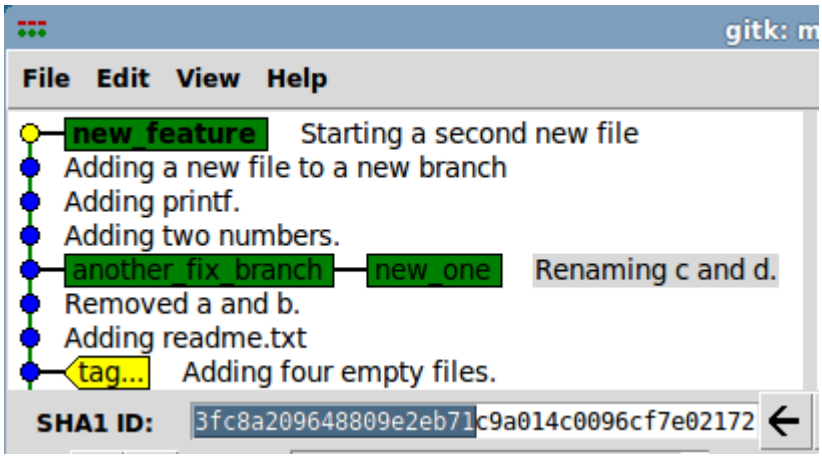
menu by clicking on the branch name presents a different menu.)



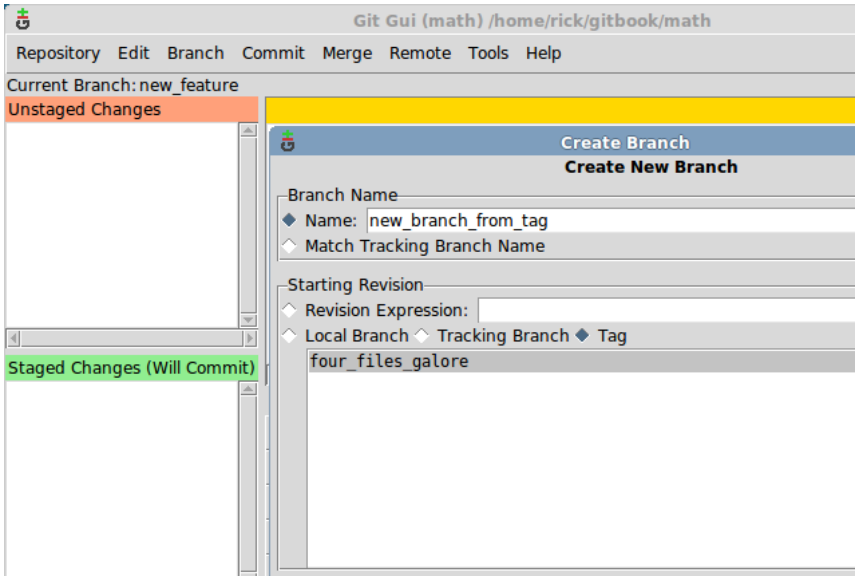

From the context menu, select "Create a new branch". A smaller window appears, allowing you to enter the name of a branch (see the following picture). I've entered "new_one."




After you press "Create", the new branch will appear in `gitk`, alongside another_fix_branch (as shown in the next figure).

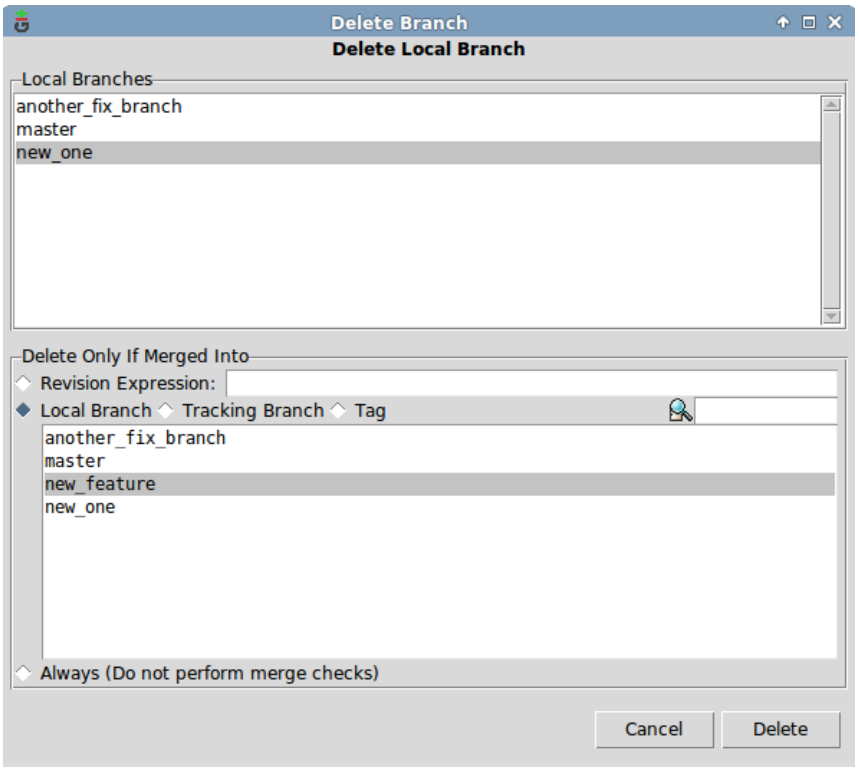



3. This exercise is a repeat of the previous two exercises. `gitk` is the easier of the two tools to create a branch off a tag. In Git GUI, in the "Create Branch" window, you must select the "Tag" button to present the list of available tags (next figure).

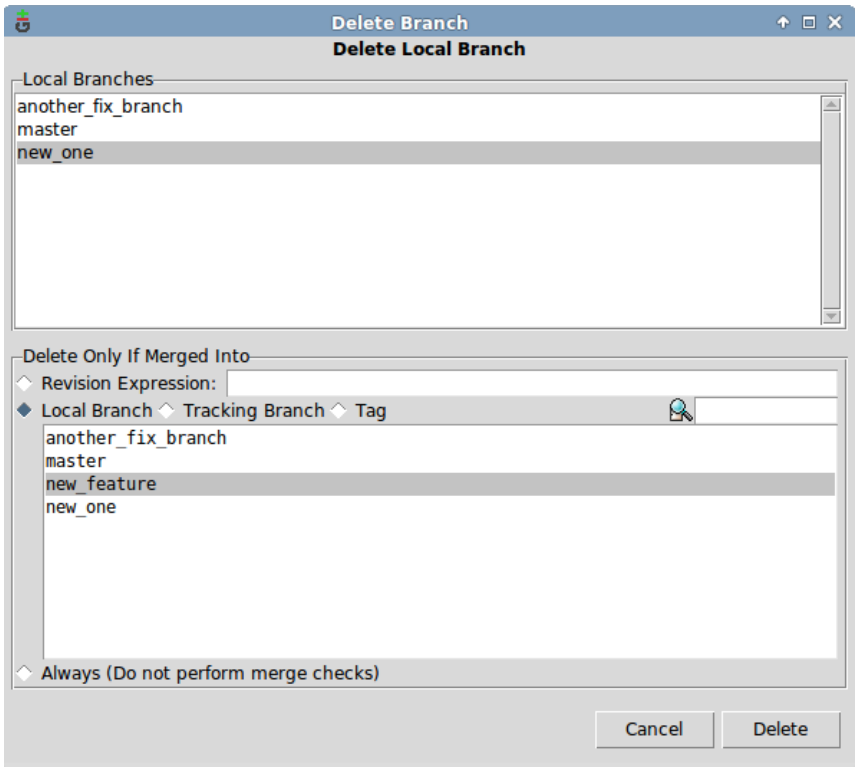



4. This exercise asks you to delete these branches using the two GUIs. Both are relatively straightforward.

   For Git GUI, you select Delete from the Branch menu (see the first figure of this section for the branch menu items). Then, in the Delete window (in the following picture), pick the branch to delete, then press "Delete."




For `gitk`, this time right-mouse button click on the branch name. You'll see a menu item to remove the branch (as shown in the next figure).




## Ch 9.5.2

1. You can rename a branch, via the `git branch -m old_name new_name` command.

2. We saw this technique earlier (in 8.5.3's second exercise). Use:

   ```
   git rev-parse :/"Renaming c and d"
   ```

3. As a reminder, the lengthy `git log` was:

   ```
   git log --graph --decorate --pretty=oneline --all --abbrev-commit
   ```

   Looking up each switch involves using the `git log --help` command. The point of this exercise is to have you read the `git log` documentation, specifically looking up particular switches.

   For the case of the `--decorate` switch, you'll see that it prints the ref names of any commit, but there are switches to decorate. For example, try `--decorate=full`.

4. To give this a proper try, go to the math directory, and try deleting the new_feature branch. (You may find it easier to regenerate the math directory using the `make_math_repo.sh` script.)

The new_feature branch has two commits in it: “Starting a second new file” and “Adding a new file to a new branch”.

Once you type `git branch -d new_feature`, you will be asked whether you really want to delete this branch, as there are commits that haven't been merged to any other branch. To really delete the branch, type `git branch -D new_feature`. (Notice that the second command uses the capital D, instead of a lowercase d.)

Now that the repository no longer has a new_feature branch, you might think that your commits for this branch are gone. However, they are still available, and the best way to see this is via `git reflog`.

Type `git reflog`, and you'll see a listing that looks like this:

```
3522453 HEAD@{0}: checkout: moving from fixing_readme to another_fix_branch
3522453 HEAD@{1}: checkout: moving from master to fixing_readme
7fb5600 HEAD@{2}: commit: A small update to readme.
b06aec0 HEAD@{3}: checkout: moving from new_feature to master
c5aab93 HEAD@{4}: commit: Starting a second new file
6851702 HEAD@{5}: commit: Adding a new file to a new branch
b06aec0 HEAD@{6}: checkout: moving from master to new_feature
b06aec0 HEAD@{7}: commit: Adding printf.
```

In the listing above, you can see our two commits at the label HEAD@{4} and HEAD@{5}. You can recreate a branch specifying this label, effectively recovering the two commits from the deleted branch!

Type:

`git checkout -b recovered_branch HEAD@{4}`

`git reflog` will be discussed some more in Chapter 16.

## Ch 9.5.3

1. This question was originally asked as part of 7.5.3!

   However, it is worth emphasizing the notion of ‘form’ again. For every Git command, the documentation lists a synopsis of ways to call that particular Git command. For the `git checkout` command, this is the synopsis.

   ```
   git checkout [-q] [-f] [-m] [<branch>]
   git checkout [-q] [-f] [-m] --detach [<branch>]
   git checkout [-q] [-f] [-m] [--detach] <commit>
   git checkout [-q] [-f] [-m] [[-b|-B|--orphan] <new_branch>] [<start_point>]
   git checkout [-f|--ours|--theirs|-m|--conflict=<style>] [<tree-ish>] [--] <paths>.
   git checkout [-p|--patch] [<tree-ish>] [--] [<paths>...]
   ```

Each line in the synopsis is a particular form of the `git checkout` command. As with 7.5.2, the answer is `git checkout [-p|--patch] [<tree-ish>] [--] [<pathspec>...]`. In the current question, we use the `--` argument, but as we saw in 7.5.2, this is optional.

2. One way to add the new line and commit the change is as follows:

```
% echo "c=1" >> math.sh
% git add math.sh
% git commit -m "Adding a third variable."
```

After the above steps, examine the repository via `gitk` (and the "All (local) branches" view configured). It should look like 9.23 in the book!

## Ch 9.5.4

1. All the branches originate from the commit with the message "Adding four empty files." This commit is tagged with "four_files_galore". To get this answer, you can use `gitk`, or use the `git lol` command and scroll all the way to the bottom.

2. The answer is random, on every run of the make_lots_of_branches.sh script. To get the right answer, type `git checkout branch_30`, then examine the output of `git log --oneline --decorate`. It should look as follows.

```
% git log --oneline --decorate --graph
* 7d3202f (HEAD -> branch_30) Commit for file_26611
* 5fe1ae0 Commit for file_20400
* 4c4316b Commit for file_25981
* 19d8bb1 Commit for file_16225
* 8bb7993 Commit for file_735
* 19a25f2 (tag: four_files_galore, master) Adding four empty files.
* 3cb4cd5 Adding b variable.
* 0967ff6 This is the second commit.
* 446143d This is the first commit.
```

In the above examine, it shows five commits. You can also see this via `gitk` (again with the "All (local) branches" configuration.

3. There are lots of ways to approach this.

   The brute force way is to use `gitk`, enable "All (local) branches", and then scroll through the entire list of commits.

   Another way to do this is to exploit the command line. You can use:

```
% git log --decorate --all |grep random_prize
commit 72dee077a500c7f7a678a24703cff04ef0ac3e13 (tag: random_prize_1, branch_22)
commit f07b28eda8c17868789b817371541bba8c18cc9f (tag: random_prize_2, branch_26)
commit 774a8cd97afe3e54427817bd25df1115cc2209b5 (tag: random_prize_3, branch_05)
```

The above command line takes the output of `git log`, and searches it via the grep command. (This is similar to the technique used in question 3 of 8.5.6.)

One more way to get the random branches is to use `git rev-parse`. Your session would look something like this:

```
% git rev-parse --tags=random_prize_*
d8d1ecc4b7e988b0daf176a3615665ef5101bc1b
aed1e7e201189636c84708f2b1420e07f93905e5
81527e46382b91f07b48f2c2871e0f7ab84aa5de
```

Then for every SHA1 ID, type `git log`, as follows:

```
% git log --decorate -1 d8d1ecc
commit 72dee077a500c7f7a678a24703cff04ef0ac3e13 (tag: random_prize_1, branch_22)
Author: Rick Umali <rickumali@gmail.com>
Date:   Tue Dec 1 21:35:44 2015 -0500

    Commit for file_6774
```

To check your answers, type `git checkout branch_40`, then examine the answers.txt file in that branch. (Notice that this file is only in this branch!)

4. The hint was the word 'contains'. `git branch` has a `--contains` switch that you can use to find any branch that contains a particular pattern (in this case, our tag name). The command line would be:

```
% git branch --contains random_tag_on_file
  branch_21
```

For the above, the random_tag_on_file is in the branch_21. (Yours will be different, potentially!)

5. Typing the command in this exercise will produce a lengthy listing of all the branches and tags *only*.

```
% git log --oneline --decorate --simplify-by-decoration --all
62240e1 (branch_32) Commit for file_3625
...
06c2269 (branch_39) Commit for file_28520
```

```
da81290 (HEAD -> branch_40) The answers are in this commit.
72dee07 (tag: random_prize_1, branch_22) Commit for file_6774
58c4052 (branch_23) Commit for file_341
dc71c38 (branch_24) Commit for file_6528
22f2527 (branch_25) Commit for file_7970
f07b28e (tag: random_prize_2, branch_26) Commit for file_8285
002b174 (branch_27) Commit for file_12577
a5c4356 (branch_28) Commit for file_20648
...
8c48a4c (branch_21) Commit for file_22333
195b7d0 (tag: random_tag_on_file) Randomly tagged file on Branch 21.
8fa867f (branch_06) Commit for file_23011
...
2da5dab (branch_04) Commit for file_22551
774a8cd (tag: random_prize_3, branch_05) Commit for file_21948
19a25f2 (tag: four_files_galore, master) Adding four empty files.
446143d This is the first commit.
```

In the output above, I used '...' to indicate rows that I left out for space. The key is that the output only contains commits that are either a branch or a tag (or both). This kind of history listing is known as a 'simple history'.

When you add the `--graph` switch, you'll see the same listing, with some ASCII branches indicating that all branches stem from the commit tagged 'four_files_galore'. (See and compare with the first question of this section.)

# Chapter 10

1. To access the `git merge` command's help file, type `git mege --help`.

2. It might be helpful, if your mergesample directory is a little disjointed, to recreate it using the `make_merge_repos.sh` script.

   This exercise asks you to merge 'master' into your branch. An example of this from scratch would look like this:

   ```
   % git branch
   * bugfix
     master

   % git checkout master
   Switched to branch 'master'

   % git checkout -b newbranch
   Switched to a new branch 'newbranch'
   ```

```
% git merge master
Already up-to-date.
```

The above session is the simplest situation. A newly created branch that merges in an unchanged master. Since nothing is changed in either branch, master is effectively merged with newbranch. To make things more realistic, add a file to newbranch:

```
% touch innb
% git add innb
% git commit -m "Adding innb file"
[newbranch fb0c9da] Adding innb file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 innb

% git merge master
Already up-to-date.
```

Does this response from `git merge master` make sense? Since master does not have any changes, it's already 'merged' with newbranch. newbranch is a direct-descendant of master. Review the diagrams in section 10.4.1 to refresh your memory of this concept.

Now, to really test your understanding, try to make a change to master.

```
% git checkout master
Switched to branch 'master'
% touch nil
% git add nil
% git commit -m "Adding nil file"
[master 4cb6caa] Adding nil file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 nil

% git checkout newbranch
Switched to branch 'newbranch'

% git merge master
Merge made by the 'recursive' strategy.
 nil | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 nil
```

Since master has a change, Git has to merge it into newbranch. You may find yourself in this situation when you have a branch off of master, but

then the master branch has a change. In order to bring in the changes from master, you could do this kind of merge, although we'll see other techniques to keep track of master in the chapter on `git rebase`.

3. "What happens if you type it now?" This isn't a great exercise question because I don't say what step to start from. Recreating the mergesample directory using the `make_merge_repos.sh` script will produce the desired state for exploring the third question: "What happens to the diff if you try a different order?"

```
% git diff master...bugfix
diff --git a/baz b/baz
index 56d6546..1c52108 100644
--- a/baz
+++ b/baz
@@ -1,3 +1,4 @@
 a=1
-b=0
+b=1
 let c=$a/$b
+echo $c

% git diff bugfix...master
diff --git a/bar b/bar
new file mode 100644
index 0000000..e69de29
diff --git a/foo b/foo
new file mode 100644
index 0000000..e69de29
```

   The "..." says what is the difference between master and branch relative to when they first became different. The output shows how to make master branch look like the bugfix branch. This is why the diff output looks so different when you reverse the two branches. To make bugfix look like master, you only have to add two new files: bar and foo.

   The question "Is there another word you can substitute for either master or bugfix?" points to the fact that any `gitrevisions` substitute for master or bugfix is allowed. The easiest example is HEAD can be substituted for either master or bugfix. From the `git diff` help page: "For a more complete list of ways to spell `commit`, see SPECIFYING REVISIONS section in `gitrevisions`."

4. After you add (and commit) a new file to the bugfix branch, doing `git diff master...bugfix` will show the output from listing 10.2, with the following entry specifying that a new file is also needed (in our example, the innb file):

```
diff --git a/innb b/innb
new file mode 100644
index 0000000..e69de29
```

The question asks about the `--name-status` output, and it would look like this:

```
% git diff --name-status master...bugfix
M       baz
A       innb
```

Here, it's apparent that innb is an addition to the branch.

5. To add a commit even though you're performing fast-forward merge, use the `--no-ff` switch. This is switch we'll explore further in Chapter 17.

# Chapter 11

1. This exercise assumes that you have two directories: math, which is the initial Git repository, and math.clone, which you created with `git clone math math.clone`.

   To complete this exercise, make another copy of the math directory using `cp -r math math.copy`, and then type `git log --oneline --all` and `git branch --all` in both the math.clone and the math.copy directories (remember: these directories are Git repositories). The `git branch` output will be different between the math.clone and math.copy directories.

   ```
   % cp -r math math.copy
   % cd math.copy
   % git branch --all
   * another_fix_branch
     master
     new_feature
   % cd ../math.clone
   % git branch --all
   * another_fix_branch
     remotes/origin/HEAD -> origin/another_fix_branch
     remotes/origin/another_fix_branch
     remotes/origin/master
     remotes/origin/new_feature
   ```

2. Similar to the above, you'll be making a clone of the math repository, but this time with the current branch of math as fixing_readme. (This branch was created back in Chapter 9. If you don't have this branch, reread section 9.2.2.) A session would look like this:

```
% cd math
% git checkout fixing_readme
Switched to branch 'fixing_readme'
% cd
% git clone math math.clone4
% cd math.clone4
% git branch
* fixing_readme
```

3. If you type `git branch --all`, you'll see all the available branches, including the remote tracking branches. The output might be this:

   ```
   % git branch --all
   * another_fix_branch
     remotes/origin/HEAD -> origin/another_fix_branch
     remotes/origin/another_fix_branch
     remotes/origin/master
     remotes/origin/new_feature
   ```

   In section 11.1.3, you saw that you could type 'new_feature', leaving out the 'remotes/origin' prefix:

   ```
   % git checkout new_feature
   Branch new_feature set up to track remote branch new_feature from origin.
   Switched to a new branch 'new_feature'
   ```

   The consequences of this automated 'set up' will be discussed in section 13.1.2.

   What if, by mistake, you specify the full name of the remote branch?

   ```
   % git checkout remotes/origin/new_feature
   Note: checking out 'remotes/origin/new_feature'.

   You are in 'detached HEAD' state. You can look around, make experimental
   changes and commit them, and you can discard any commits you make in this
   state without impacting any branches by performing another checkout.

   If you want to create a new branch to retain commits you create, you may
   do so (now or later) by using -b with the checkout command again. Example:

     git checkout -b <new-branch-name>

   HEAD is now at b99624d... Starting a second new file
   ```

   This variation does not do the automatic set up, and you are in a detached HEAD situation, as described in Chapter 8.

4. This exercise invites you to create a confusing situation: naming a local branch different from its remote-tracking branch. There may be times when this will come in handy (perhaps if you need two copies of a single branch, for example).

   Your session might look like this:

```
% git branch --all
fixing_readme
* new_feature
  remotes/origin/HEAD -> origin/fixing_readme
  remotes/origin/another_fix_branch
  remotes/origin/fixing_readme
  remotes/origin/master
  remotes/origin/new_feature

% git checkout -b my_other_fixing_readme remotes/origin/fixing_readme
Branch my_other_fixing_readme set up to track remote branch fixing_readme from ori
Switched to a new branch 'my_other_fixing_readme'
```

   Section 13.1.3 will describe the `git remote -v show origin`, which can help you keep track of which remote-tracking branch goes with which local branch.

5. As far as I know, there is no limit to the number of branches that use a particular starting point (starting commit).

6. The `--origin` switch in `git clone` allows you to specify a different name or label for the remote location. A session that demonstrates `--origin` would look as follows:

```
% git clone --origin distant_planet math math.clone5
Cloning into 'math.clone5'...
done.

% cd math.clone5

% git branch --all
* fixing_readme
  remotes/distant_planet/HEAD -> distant_planet/fixing_readme
  remotes/distant_planet/another_fix_branch
  remotes/distant_planet/fixing_readme
  remotes/distant_planet/master
  remotes/distant_planet/new_feature
```

# Chapter 12

## Ch 12.4.1

1. The output of `git log --oneline --decorate` is something like this.

```
% git log --oneline --decorate
4465c54 (HEAD, origin/master, origin/HEAD, master) A small update to readme.
6f6af16 Adding printf.
256d402 Adding two numbers.
23d3077 (origin/another_fix_branch) Renaming c and d.
80f5738 Removed a and b.
94abe13 Adding readme.txt
ef47d3f (tag: four_files_galore) Adding four empty files.
3847b0b Adding b variable.
2732d6a This is the second commit.
e7a974f This is the first commit.
```

2. 23d3077

3. Yes: four_files_galore.

4. The SHA1 IDs of math.github will be different from math.bob and math.clone. SHA1 IDs do not repeat when generated on different files and servers (see section 8.1.1). When SHA1 IDs are generated for commits, your specific computer and commit information is used in the generation of the SHA1 ID. Because of this, there is no way these SHA1 IDs would ever be the same.

## Ch 12.4.2

This exercise repeats sections 12.1.3 and 12.2. Your session shoud look similar to this:

```
% cd math.bob
% git remote
origin
% git remote add carol ../math.carol
% cd ../math.carol
% vim readme.txt
% git commit -a -m "Small update to readme.txt"
[master 4ec7d23] Small update to readme.txt
 1 file changed, 1 insertion(+)
% cd ../math.bob
% git ls-remote carol
```

```
4ec7d2333b71eb7ed1f4c3e50e082d8a278414f0        HEAD
4ec7d2333b71eb7ed1f4c3e50e082d8a278414f0        refs/heads/master
ffe2dda78e503a618d87782e960e5b3e8371db89        refs/remotes/origin/HEAD
249644fc932f6e005a4eb2d705159b8eb8f21c75        refs/remotes/origin/another_fix_branch
ffe2dda78e503a618d87782e960e5b3e8371db89        refs/remotes/origin/master
03d04ea643f9344f814853cc670cc221882cd995        refs/remotes/origin/new_feature
2edbb3543e903400f39d9e19e516d321b3855e24        refs/tags/four_files_galore
411dc82706661c0b65664c0dd97fbd7c4ea5f9f3        refs/tags/four_files_galore^{}
```

The key is that the commit you make in math.carol (4ec7d23) is shown in the `git ls-remote carol` command.

## Ch 12.4.3

This exercise asks you to try out the other `git remote` commands. In the math.bob directory, your experiments with the delete and update commands might look like this:

```
% cd math.bob
% git remote remove carol
% git remote
origin
% git remote add tester ../math.nonexist
% git remote -v show
origin  /home/rick/gitbook/math.git (fetch)
origin  /home/rick/gitbook/math.git (push)
tester  ../math.nonexist (fetch)
tester  ../math.nonexist (push)
% git remote set-url tester ../math.carol
% git remote -v show
origin  /home/rick/gitbook/math.git (fetch)
origin  /home/rick/gitbook/math.git (push)
tester  ../math.carol (fetch)
tester  ../math.carol (push)
```

The set-url subcommand of `git remote` seems to allow you to specify multiple URLs for a remote, by employing the –add switch. For example:

```
% git remote set-url --add tester ../math.another
% git remote -v show
origin  /home/rick/gitbook/math.git (fetch)
origin  /home/rick/gitbook/math.git (push)
tester  ../math.carol (fetch)
tester  ../math.carol (push)
tester  ../math.another (push)
```

This seems to be a way to use one remote name to push to multiple locations! I haven't tested this, but I could see where it might be useful.

## Ch 12.4.4

What you should see if you try to clone to a directory that already exists using Git GUI is:

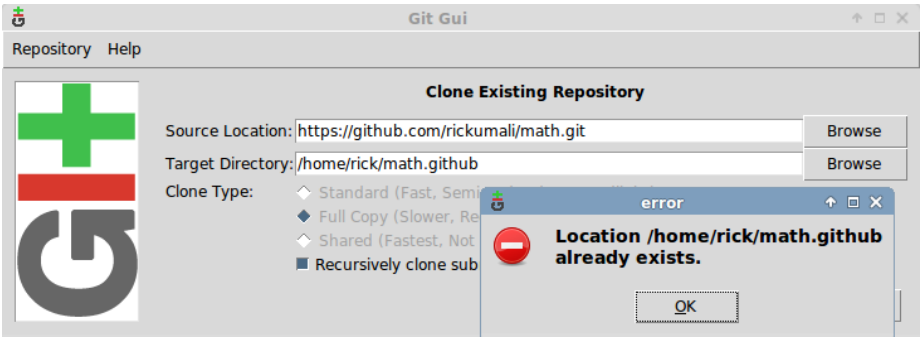



Figure 3: Git GUI won't overwrite an existing directory

Once you make the clone in Git Gui, you can compare the SHA1 IDs with `git log` between the new clone and your earlier math.github clone.

## Ch 12.4.5

Yes, the SHA1 IDs are the same. Your session could look like this:

```
% git clone https://gitlab.com/rickumali/math math.gitlab
Cloning into 'math.gitlab'...
remote: Counting objects: 35, done.
remote: Compressing objects: 100% (23/23), done.
remote: Total 35 (delta 6), reused 0 (delta 0)
Unpacking objects: 100% (35/35), done.
Checking connectivity... done.
% cd math.gitlab
% git remote -v
origin  https://gitlab.com/rickumali/math (fetch)
origin  https://gitlab.com/rickumali/math (push)
% git log --oneline
4465c54 A small update to readme.
6f6af16 Adding printf.
256d402 Adding two numbers.
```

```
23d3077 Renaming c and d.
80f5738 Removed a and b.
94abe13 Adding readme.txt
ef47d3f Adding four empty files.
3847b0b Adding b variable.
2732d6a This is the second commit.
e7a974f This is the first commit.
% cd ../math.github
% git remote -v
origin  https://github.com/rickumali/math.git (fetch)
origin  https://github.com/rickumali/math.git (push)
% git log --oneline
4465c54 A small update to readme.
6f6af16 Adding printf.
256d402 Adding two numbers.
23d3077 Renaming c and d.
80f5738 Removed a and b.
94abe13 Adding readme.txt
ef47d3f Adding four empty files.
3847b0b Adding b variable.
2732d6a This is the second commit.
e7a974f This is the first commit.
```

# Chapter 13

TBD.

# Chapter 14

TBD.

# Chapter 15

TBD.

# Chapter 16

TBD.

## Chapter 17

TBD.

## Chapter 18

TBD.

## Chapter 19

TBD.

## Chapter 20

TBD.