introduction to

tutorial @ auburn university • october 2015

Before You Start...

1. Create a GitHub Account

1. Browse to https://github.com and click Sign up.
2. Fill out the form (username, e-mail address, password), and click "Create an account."
3. Select the Free plan ($0/month, 0 private repositories), and click "Finish sign up."
4. You will be sent an e-mail with a link to verify your e-mail address. You will need to verify your address before you can continue using GitHub.

2. Install Git

Windows:

- Download an installer from http://git-scm.com/download/win

Mac OS X:

- If you have installed the Xcode Command Line Tools, you should already have Git. Open a Terminal, and type `git` (you should get a lengthy help message if it is installed).
- Otherwise, you can download an installer from http://git-scm.com/download/mac

Linux:

- On Ubuntu or Debian: `sudo apt-get install git`
- On Fedora or CentOS – as root: `yum install git`

3. Configure Git

When you use Git in a team, it will be important to keep track of who changed which files. It's good practice to configure Git with your name and e-mail address when you first install it.

1. Open a Terminal (Mac/Linux) or Git Command Prompt (Windows).
2. Type: `git config --global user.name "`*Your name*`"`
3. Type: `git config --global user.email "`*Your e-mail address*`"`

This will create a "git lg" command that displays the Git log in a compact, easy-to-read format.[1]

4. Type this (all on one line): `git config --global alias.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr)%C(bold blue)<%an>%Creset' --abbrev-commit"`

[1] From http://fredkschott.com/post/2014/02/git-log-is-so-2005/

Hands-On 2.1: Creating and Cloning a Git Repository

Create a New Repository on GitHub

You must be logged into GitHub. From the GitHub main page...

1. Click the + symbol in the upper-right corner, and select "New repository" from the dropdown.
2. In the Repository name field, enter `aututorial1`
3. Check the checkbox labeled "Initialize this repository with a README."
4. Click "Create repository."
5. You will be taken to the GitHub main page for your repository. Notice the URL: https://github.com/*your_user_name*/aututorial1

Clone the Repository to Your Local Machine

1. On the right side of the GitHub main page for your repository, there is an "HTTPS clone URL." The URL will have the form: https://github.com/*your_user_name*/aututorial1.git
   Click the clipboard icon on the right side of the URL to copy it to the clipboard.

2. Open a Terminal (Mac/Linux) or Git Command Prompt (Windows).

3. At the command prompt, clone the repository from that URL by typing:
   `git clone https://github.com/`*your_user_name*`/aututorial1.git`
   The output of the git clone command should be something like this:
   ```
   Cloning into 'aututorial1'...
   remote: Counting objects: 3, done.
   remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
   Unpacking objects: 100% (3/3), done.
   Checking connectivity... done.
   ```
   This will create a directory called aututorial1 containing a single file, README.md.

4. Change to the aututorial1 directory. Type: `cd aututorial1`

Summary

- An easy way to create a new Git repository is to create it at GitHub.com and initialize it with a README (or a .gitignore).
- After creating a repository on GitHub, copy its URL from GitHub (the HTTPS URL is easiest), and use `git clone` ***URL*** to copy it to your local machine.

Hands-On 2.2: Populating the Git Repository

You should be in a command prompt window, and aututorial1 should be the current directory.

1. Type `git status`. Since you have not changed anything, the output should be:

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

2. In the aututorial1 directory, create a new file called lines.txt will the following contents:

```
This is line 1
This is line 2
This is line 3
This is line 4
This is line 5
This is line 6
This is line 7
This is line 8
This is line 9
```

3. Type `git status`. The output should be:

```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        lines.txt

nothing added to commit but untracked files present (use "git add" to track)
```

   "Untracked" means **that the file exists, but it is not part of the Git repository.**

4. Type `git add lines.txt`

   This informs Git that this file (lines.txt) should be part of the next commit. In Git terminology, this change is staged for commit.

5. Type `git status`. The output should be:

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

        new file:   lines.txt
```

6. Type `git commit`. An editor will open; enter a one-line description of your change (e.g., "Added sample file"), save the file, and exit. The output should be:

```
 1 file changed, 9 insertions(+)
 create mode 100644 lines.txt
```

7. Type `git status`. The output should be:

```
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
nothing to commit, working directory clean
```

   "Ahead of 'origin/master' by 1 commit" means that you have made a commit on your local computer, but those changes have not been saved to the GitHub servers.

8. In your Web browser, browse to `https://github.com/`*your_user_name*`/aututorial1` and notice that lines.txt is not visible. The commit where you added lines.txt is on your local machine, but it is not yet on the server.

9. Type `git log`. The output should show two commits in reverse chronological order.

10. If you configured the `git lg` command (see page 1), type `git lg`. Your output will be different, but it should look something like this:

```
* b563cf1 - (HEAD -> master) Added sample file (2 minutes ago)<AU Tutorial>
* 74998ff - (origin/master, origin/HEAD) Initial commit (3 weeks ago)<aututorial
```

    - Notice that **HEAD** is next to the latest commit. This is the most recent commit on your local machine.
    - Notice that **origin/HEAD** is on the initial commit. This is the most recent commit that the server is aware of. In other words, the server does not "know" that you added lines.txt; you still need to copy that commit from your local machine to the server.

11. Now, to save the new commit to the server, type `git push origin master`. You will be prompted to enter your GitHub username and password. The output should be something like this:

```
Counting objects: 4, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 334 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/your_user_name/aututorial1.git
   74998ff..b563cf1  master -> master
```

12. In your Web browser, browse to `https://github.com/`*your_user_name*`/aututorial1` and notice that lines.txt is now visible, since your commit has been pushed to the server.

Summary

- Use `git status` to determine what files are being tracked and whether your local files differ from the latest version under version control. Untracked files are not under version control (essentially, Git won't touch them).
- Use `git add` *filename* to begin tracking a file that was previously untracked.
- Use `git commit` to save your changes in version control on your local machine.
- Use `git push origin master` to copy local commits to the GitHub server.
- Use `git log` (or if you configured it, `git lg`) to view a list of commits.

Hands-On 2.3: (Re)moving Files

You should be in a command prompt window, and aututorial1 should be the current directory.

Rename lines.txt to stuff.txt

1. Type `git status`. Since you have not changed anything, the output should be:
   ```
   On branch master
   Your branch is up-to-date with 'origin/master'.
   nothing to commit, working directory clean
   ```

2. You should have two files in the current directory: README.md and lines.txt. (To list the files in the current directory, use the `ls` command on Mac/Linux or `dir` on Windows.)

3. Type `git mv lines.txt stuff.txt`. This will rename lines.txt to stuff.txt. If you list the files in the current directory, you should have two files: README.md and stuff.txt.

4. Type `git status`. The output should be:
   ```
   # On branch master
   # Changes to be committed:
   #   (use "git reset HEAD <file>..." to unstage)
   #
   #       renamed:    lines.txt -> stuff.txt
   #
   ```

   In other words, this renaming is staged for commit.

5. Commit this change using `git commit`. Again, you will be prompted to enter a commit comment (something like "Renamed lines.txt to stuff.txt" is appropriate).

   Remember that `git commit` only commits a change *locally*—the change is not yet visible on the server.

6. Look at your repository on GitHub, and notice that the file is still called lines.txt, even though it's been renamed on your local machine.

7. Push this commit to the server using `git push origin master`. Then, look at your repository on GitHub, and notice that the file has been renamed to stuff.txt.

Delete README.md

8. Type `git rm README.md`. This will delete the file README.md. If you list the files in the current directory, you should have only one file: stuff.txt.

9. Type `git status`. The output should be:

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

      deleted:    README.md
```

10. Commit this change using `git commit`, as before.

11. Push this commit to the server using `git push origin master`. Look at your repository on GitHub, and verify that the README is gone.

Ignore files named *.bak

12. Make a copy of stuff.txt, and name it stuff.txt.bak
13. Type `git status`. Notice that stuff.txt.bak is untracked.
14. Create a new file called **.**gitignore (don't forget the leading period!) that contains a single line:

```
*.bak
```

15. Type `git status`. The output should be:

```
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)

      .gitignore

nothing added to commit but untracked files present (use "git add" to track)
```

    Since the pattern "*.bak" was added to .gitignore, Git will ignore all files with a .bak filename extension—they will not be listed as untracked, even if they are present.

16. Add the .gitignore file to the repository using `git add .gitignore`.
17. Commit it using `git commit`.
18. Push this commit to the server using `git push origin master`.
19. Type `git status`. The output should be:

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

20. Look at your repository on GitHub, and verify that stuff.txt.bak is *not* present on the server. However, if you list the files in the current directory, it is still present on your local machine. This file is being ignored by Git.

What Files to Ignore?

You should commit *source code* to a repository, but not backup files or output files. For example:

- Some text editors create backup files, like something.txt.bak or something.txt~. These should be ignored.
- Only source code should go into a repository, not generated files.
    - In Java, each .java file is compiled to a .class file. These .class files should be ignored. Likewise, if you compile your code into a JAR, the .jar file should be ignored.
    - In C/C++, executables (*.exe files on Windows, a.out on Linux, etc.) should be ignored, as well as object files (*.obj on Windows, *.o on Linux), core dumps, etc.
    - In Python, .pyc files should be ignored.
- This site contains .gitignore files for most common programming languages: https://github.com/github/gitignore It also contains a .gitignore for Visual Studio projects (Visual Studio projects can be shared by several people on a team, but they contain some files that are unique to each computer and therefore should be ignored by Git).

Later, On Your Own: If you delete or rename a file without using "git mv" or "git rm"...

- Try deleting a file *without* using git rm. In other words, delete it like any other file. Then, `git status` will report that the file has been deleted, but this deletion is *not* staged for commit. That's OK; you can still use git rm to stage this change, even though the file has already been deleted from your local directory.
- Try renaming a file *without* using git mv. In other words, rename it like any other file on the file system. Use git status to see what happens: Git thinks the original file was deleted, and a new file was added. You can use git rm and git add to stage these changes.

Summary

- Use `git mv` *old_filename* *new_filename* to rename a file (or to move a file to a different directory.)
- Use `git rm` *filename* to delete a file.
- To ignore files with certain names, add a text file named `.gitignore` to your repository. Each line contains a filename pattern. For example, the pattern `*.bak` will ignore files with a .bak filename extension. See https://github.com/github/gitignore for common .gitignore files.

Hands-On 2.4: Undoing Modifications

You should be in a command prompt window, and aututorial1 should be the current directory. The file stuff.txt should be in the current directory, and it should contain nine lines of text.

Reset: "Unstaging" a Change

1. Modify stuff.txt. Delete a line or two, and add a couple of new lines.
2. Add a new file called new.txt.
3. Type `git add stuff.txt new.txt` to stage the changes to both files.
4. Type `git status`, and verify that both files will be included in the next commit.
5. Now, suppose you don't want to include stuff.txt. Type `git reset stuff.txt` and verify that the output is:
   ```
   Unstaged changes after reset:
   M       stuff.txt
   ```
6. Type `git status`, and verify that only new.txt will be included in the next commit.

Hard Reset: Eliminate All Local Changes

7. Create a new file called more.txt.
8. Type `git status`, and verify that
   (1) new.txt is a new file staged for commit,
   (2) stuff.txt is modified but not staged for commit, and
   (3) more.txt is untracked.
9. Now, suppose you want to undo *all* of your changes to tracked files. Type
   `git reset --hard`
   The output should be something like:
   `HEAD is now at` *description of the last commit*
10. Type `git status`. The output should be:
    ```
    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)

            more.txt

    nothing added to commit but untracked files present (use "git add" to track)
    ```
11. View the contents of stuff.txt. Notice that all of your changes are gone.
12. Get a directory listing. Notice that new.txt is gone, but more.txt is still present.
13. Delete more.txt.

What just happened? A hard reset resets the contents of every tracked file to its contents from the last commit. Since stuff.txt was tracked, its contents were reset. The file new.txt was deleted because you staged its addition (so it was tracked), but it was not present in the last commit. Since more.txt was untracked, it was not changed.

Checkout: Undoing Changes to a Particular File

14. Modify stuff.txt. Delete a line or two, and add a couple of new lines.
15. Type `git checkout -- stuff.txt`
16. Look at the contents of stuff.txt. Notice that your changes are gone: It has been replaced with the version from the most recent commit.
17. Now, modify stuff.txt again. Delete a line or two, and add a couple of new lines.
18. Use git add to stage your changes.
19. Type `git status`, and verify that the modification will be included in the next commit.
20. Edit stuff.txt, and add a new line to the bottom of the file:
    ```
    This line was added after staging
    ```
21. Type `git status` and verify that the output is:
    ```
    On branch master
    Changes to be committed:
      (use "git reset HEAD <file>..." to unstage)

            modified:   stuff.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:   stuff.txt
    ```

    Huh? How is stuff.txt staged and not staged at the same time?!

    This means that you staged a change to stuff.txt, but the file has been modified since you staged it. If you committed it now, the staged version would be committed, without the more recent modifications.

22. Type `git checkout -- stuff.txt`
23. Look at the contents of stuff.txt. Notice that it has reverted to the last version you staged: Your new last line is gone, but the other changes are still present.
24. Type `git checkout HEAD -- stuff.txt`
25. Look at the contents of stuff.txt. Notice that it has reverted to the last committed version. Now, all of your changes are gone, including changes you previously staged.
26. Type `git status`, and verify that there are no outstanding changes.

Summary

- Use `git reset` *filename* to "unstage" a change (e.g., to undo the effects of git add/rm). If a file was modified, it will still be modified, but the modification will no longer be staged for commit.
- Use `git reset --hard` to get rid all of changes to tracked files, changing all tracked files back to the version from the most recent commit. Untracked files will be left alone.
- Use `git checkout --` *filename* to reset a file to the last staged version.
- Use `git checkout HEAD --` *filename* to reset a file to the last committed version.

Hands-On 2.5: Git Diff

You should be in a command prompt window, and aututorial1 should be the current directory. The file stuff.txt should be in the current directory, and it should contain nine lines of text.

Diff Against HEAD: What lines have changed since the last commit?

1. Type `git status`. Ensure that you don't have any local changes; the output should be:
   ```
   On branch master
   Your branch is up-to-date with 'origin/master'.
   nothing to commit, working directory clean
   ```
2. Edit stuff.txt. Delete line 3, and add a line between lines 6 and 7, so it looks like this:
   ```
   This is line 1
   This is line 2
   This is line 4
   This is line 5
   This is line 6
   This is a new line between lines 6 and 7
   This is line 7
   This is line 8
   This is line 9
   ```
3. Type `git diff HEAD`. The output should be:
   ```
   diff --git a/stuff.txt b/stuff.txt
   index 8675b0b..d924c44 100644
   --- a/stuff.txt
   +++ b/stuff.txt
   @@ -1,9 +1,9 @@
    This is line 1
    This is line 2
   -This is line 3
    This is line 4
    This is line 5
    This is line 6
   +This is a new line between lines 6 and 7
    This is line 7
    This is line 8
    This is line 9
   ```

   Notice the format of the diff:
   - The filename is shown first (stuff.txt).
   - The @@ –1, 9 indicates that 9 lines have changed, starting at line 1.
   - Deleted lines have – at the left.
   - Added lines have + at the left.
   - The diff includes 3 unchanged lines before and after the lines that have changed.

Diff with New Files

4. Create a new file, called new.txt, and add a line or two of text to it. Do not stage it yet.
5. Type `git diff HEAD`; notice that new.txt is not included in the diff (since it's untracked).
6. Use git add to add new.txt to the repository.
7. Type `git diff HEAD`; notice that new.txt is included in the diff (since it's tracked now).

Diff for a Single File

8. Type `git diff HEAD -- stuff.txt`. This displays a diff against HEAD for only stuff.txt (new.txt is not included in the diff).

Diff against a Particular Version

9. Type `git log` and look at the very last entry in the log (which will be labeled "Initial commit"). The full log entry should look something like this (yours will be slightly different):

```
commit 74998ff596afe7aefeb0a5531ab5e231a27e9a82
Author: aututorial <com-github-aututorial@jeff.over.bz>
Date:   Wed Sep 16 19:10:09 2015 -0500

    Initial commit
```

Every commit has a unique identifier (above, 74998ff5…). This is its SHA-1 hash (pronounced "shaw one," like "saw one" but with an "sh"). You will use this when you need to refer to a particular commit. Usually, you do not need to type the entire hash; the first 5 or 6 characters are enough to identify it.

10. Type `git diff` *sha1* `-- stuff.txt` where *sha1* is the first 5 or 6 characters of the SHA-1 hash for your initial commit (from the previous step). This will show how the current version of stuff.txt differs from the version in your initial commit. Since it was not present in the initial commit, the diff should indicate that the entire file is new (all lines have been added).

Cleanup

11. Use `git reset --hard` to undo all of the changes since your last commit.

Summary

- Use `git diff HEAD` to see what has changed since the most recent commit.
- Use `git diff HEAD --` *filename* to see changes to a specific file.
- Use `git diff` *sha1* `--` *filename* to see differences from a particular commit. You can use git log (or git lg) to find the SHA-1 hash for each commit.

Hands-On 3.1: Creating a Local Branch

You should be in a command prompt window, and aututorial1 should be the current directory. The file stuff.txt should be in the current directory, and it should contain nine lines of text.

Create a Local Branch & Make a Commit

1. Type `git status`. The output should indicate that you are on the master branch (which is the default if you haven't created any new branches) and don't have any local changes; the output should be:
   ```
   On branch master
   Your branch is up-to-date with 'origin/master'.
   nothing to commit, working directory clean
   ```

2. Type `git branch foo`. This will create a branch called foo, starting from the most recent commit on the master branch.

3. Type `git status`. Notice that you are still on the master branch—you have not switched to the foo branch, even though you have created it. The output should be:
   ```
   On branch master
   nothing to commit, working directory clean
   ```

4. Another way to determine which branch you are on is to type `git branch`. It will show a list of branches with an asterisk next to the branch you currently have checked out.
   ```
     foo
   * master
   ```

5. Type `git checkout foo`. This will switch to the foo branch. The output should be:
   ```
   Switched to branch 'foo'
   ```

6. Type `git branch` and verify that you are now working on the foo branch:
   ```
   * foo
     master
   ```

7. Edit stuff.txt. Change line 5 so it reads:
   ```
   This is line 5, which has changed on the foo branch
   ```

8. Commit your changes using git add and git commit.

9. Type `git diff master -- stuff.txt`. This will show the differences between the master branch and the current branch (foo). The output should be:

```
diff --git a/stuff.txt b/stuff.txt
index 8675b0b..5e0c27a 100644
--- a/stuff.txt
+++ b/stuff.txt
@@ -1,9 +1,9 @@
 This is line 2
 This is line 3
 This is line 4
-This is line 5
+This is line 5, which has changed on the foo branch
 This is line 6
 This is line 7
 This is line 8
```

Try to Switch Branches with Uncommitted Changes

10. Type `git checkout master` to check out the master branch. The output should be:

```
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'
```

11. Open stuff.txt, and notice that line 5 has gone back to the original version, not the changed version from the foo branch.

12. Edit stuff.txt, and add a new line to the bottom of the file:

```
Line 10 was added in the master branch
```

13. Create a new file called newfile.txt. (Put anything you want in it.)

14. Type `git status` and verify that the output is:

```
#On branch master
Your branch is up-to-date with 'origin/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:   stuff.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        newfile.txt

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

15. Try to switch to the foo branch without committing this changes. Type `git checkout foo` and verify that the output is:

```
#error: Your local changes to the following files would be overwritten by checkout:
        stuff.txt
Please, commit your changes or stash them before you can switch branches.
Aborting
```

Commit Changes to stuff.txt But Leave newfile.txt Untracked

16. Type `git add stuff.txt` and `git commit` to commit the changes to stuff.txt in the master branch. (Do not commit newfile.txt; leave it untracked.)
17. Use `git checkout foo` to switch to the foo branch.
18. Type `git status` and verify that newfile.txt is untracked, even after switching branches.

Commit newfile.txt on the foo Branch

19. Commit newfile.txt to the foo branch using git add and git commit.
20. Check out the master branch.
21. Get a directory listing, and verify the newfile.txt is gone—it only exists on the foo branch now.

Summary

- Use `git branch` to get a list of local branches, including which branch is checked out.
- The output of `git status` also displays the current branch.
- Use `git branch` *branch_name* to create a branch without checking it out.
- Use `git checkout` *branch_name* to check out a branch.
- Use `git diff` *branch_name* `--` *filename* to see how a specific file differs from the version on another branch.
- You cannot switch branches if you have uncommitted changes. However, untracked files are fine—they will not be changed or deleted when you switch branches.

Hands-On 3.2: Merging foo into master

You should be in a command prompt window, and aututorial1 should be the current directory. You should be on the master branch.

1. Type `git status` and verify that you are working on the master branch with no uncommitted changes.

2. To merge the foo branch into the current branch (master), type `git merge foo`. A text editor will open, asking you to enter a commit message (the default, "Merge branch 'foo'", is fine.) The output should be:

```
#Auto-merging stuff.txt
Merge made by the 'recursive' strategy.
 newfile.txt | 0
 stuff.txt   | 2 +-
 2 files changed, 1 insertion(+), 1 deletion(-)
 create mode 100644 newfile.txt
```

3. To delete the foo branch, type `git branch -d foo`. The output should be something like:

```
Deleted branch foo (was 900d36d).
```

4. If you configured the `git lg` command, type `git lg` to see your commit history:

```
*   f75b6ba - (HEAD, master) Merge branch 'foo' (25 minutes ago)<AU Tutorial>
|\
| * 900d36d - Added newfile.txt (68 minutes ago)<AU Tutorial>
| * 6471590 - Changed stuff.txt on foo (84 minutes ago)<AU Tutorial>
* | c4347e1 - Modified stuff.txt in master (71 minutes ago)<AU Tutorial>
|/
* 635b4b7 - (origin/master, origin/HEAD) Added .gitignore (7 days ago)<AU Tutorial>
* 4f9b886 - Deleted README.md (7 days ago)<AU Tutorial>
* a0fc71b - Renamed lines.txt to stuff.txt (7 days ago)<AU Tutorial>
* b563cf1 - Added sample file (7 days ago)<AU Tutorial>
* 74998ff - Initial commit (4 weeks ago)<aututorial>
```

5. Recall that newfile.txt existed in the foo branch but not in the master branch. Get a directory listing, and verify that newfile.txt exists now. This file was created when you did the merge, just like any other change.

Summary

- To merge changes from a particular branch into the master branch:
  - Check out the master branch
  - Type `git merge` ***branch_name***
- Use `git branch -d` ***branch_name*** to delete a branch.

Hands-On 3.3: Resolving Conflicts

You should be in a command prompt window, and aututorial1 should be the current directory. You should be on the master branch.

1. Type `git status` and verify that you are working on the master branch with no uncommitted changes.
2. Create a branch called `bar`. (Do not check it out yet.)
3. Make sure you're still on the master branch, and edit stuff.txt.
    - Change line 8 so it reads:
      `This is line 8 on master. It will conflict.`
    - Add a line at the top of the file
      `This line was added at the top of the file in master.`
4. Commit this change to the master branch.
5. Check out the bar branch.
6. Make sure you're on the bar branch, and edit stuff.txt.
    - Change line 8 so it reads:
      `This is line 8 on bar. It will conflict.`
    - Add a line at the bottom of the file
      `This line was added at the bottom of the file in bar.`
7. Commit this change to the bar branch.
8. Check out the master branch.
9. Type `git merge bar` to merge the bar branch into the current branch (master). The output should be:

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

10. Open stuff.txt in a text editor. It should look something like this.

```
This line was added at the top of the file in master.
This is line 1
This is line 2
This is line 3
This is line 4
This is line 5, which has changed on the foo branch
This is line 6
This is line 7
<<<<<<< HEAD
This is line 8 on master. It will conflict.
=======
This is line 8 on bar. It will conflict.
>>>>>>> bar
This is line 9
Line 10 was added in the master branch
This line was added at the bottom of the file in bar.
```

Git was able to merge the changes from the top and bottom of the file. However, there is a problem with line 8, since it changed in both the master branch and the bar branch.

The text between <<<<<<< and ======= is from the current branch. The text between ======= and >>>>>>> is from the bar branch.

11. Delete the <<<<<<< line and the four lines after it, including the >>>>>>> line, and replace those lines with:
    `This is line 8 after I have resolved the conflict.`
12. Type `git add stuff.txt`
13. Type `git commit` (and notice that the default commit message is "Merge branch 'bar'" since you are finishing a merge commit).
14. If you configured the `git lg` command, type `git lg` to see your commit history.
15. Delete the bar branch.

Summary

- It is OK if a file changes on both branches. Git will try to auto-merge the changes.
- If Git cannot auto-merge changes…
    - You will get an error message telling you to manually resolve conflicts.
    - Git will insert <<<<<<< ======= >>>>>>> markers around the lines it was not able to merge. Edit the file, replacing that section with whatever the "correct" version is.
    - Use `git add` and `git commit` to finish the merge.

Hands-On 4.1: Submitting a Pull Request

You should navigate back to whatever directory you wish to clone a new repository into. It should NOT be inside of an already existing git repository.

1. In your Web browser, open https://github.com/aututorial/aututorial2
2. On the top right of the page, click the button labeled "Fork"
3. Type `git clone https://github.com/`*your_user_name*`/aututorial2.git`
4. Change directories to the newly cloned directory (it should be called aututorial2)
5. Type `git status` to be certain you are on master.
6. Create and switch to a branch called `bugfixes`.
7. Open PerfectShuffle.java in your favorite text editor.
8. Fix PerfectShuffle. The new file should look like this:

```
import java.util.*;

public class PerfectShuffle {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);

        for (;;) {
            int num = in.nextInt();
            if (num == 0) {
                return;
            }
            String[] input = new String[num];
            for (int i = 0; i < num; i++) {
                input[i] = in.next();
            }

            String[] strings = new String[num];
            for (int i = 0; i < num / 2 + num % 2; i++) {
                strings[i * 2] = input[i];
            }

            for (int i = 0; i < num / 2; i++) {
                strings[i * 2 + 1] = input[i + num / 2 + num % 2];
            }

            for (String s : strings) {
                System.out.println(s);
            }
        }
    }
}
```

9. Commit this change to the bugfixes branch.
10. Go back to your browser and browse to your fork of aututorial2 (the URL should be github.com/*your_user_name*/aututorial2).
11. Click the green button that says "Compare and Pull Request."
12. On the next page, click "Create Pull Request."

Summary

- To make changes on an open-source repository on GitHub:
    - Fork the repository (to create a copy under your GitHub account).
    - Make changes in your fork.
    - Then submit a pull request.
- Submitting a pull request allows the code to be reviewed by others and allows for them to decide how to handle the changes that you have proposed.