Learn to Code

Git Branching

Workbook 6

Version 6.0 Y

Table of Contents

Module 1

Branching and Merging with Git

Section 1–1

Working with Branches

Branching

- **Branching allows us to pursue a new line of development**
  - We can keep it separate and distinct from the ***main*** branch
    - NOTE: GitHub used to call the main branch the "master" branch and you will still see that mentioned in many Google references

- **By doing this, we can:**
  - We can keep code in development separate from code in production
  - Add a new feature while keeping the main branch stable
  - Test out an idea that we aren't sure we want to keep

- **In the diagram you just saw, the Develop branch is used during the coding phase**
  - New features branch off from there

- **When a new feature is completed, it can be merged back in the Develop branch**

- **When it is time to release the product, you can merge it into a Release branch**
  - By keeping code here, you know EXACTLY what code went live
- **The "main" copy of working code is merged back into the main branch**
- **When the next set of changes need to be made to the program, you can branch from Release**

Practical Example of Branching

- **You are going to create a new feature in a project that will take some time to complete**
  - Make a new branch and start coding the new feature
- **Suddenly, an emergency change comes in that has to be pushed live**
- **Switch back to main (or whatever branch has the issue)**
  - Create another branch for the emergency changes
  - Make the changes
  - Commit the changes
  - Merged the emergency change branch back to main
- **Switch back to your feature branch and continue working on the task**

Section 1–2

Merging and Branching Commands

Creating a Branch: `git branch <name>`

- **The `git branch` command creates a branch with the specified name**

Creating a branch

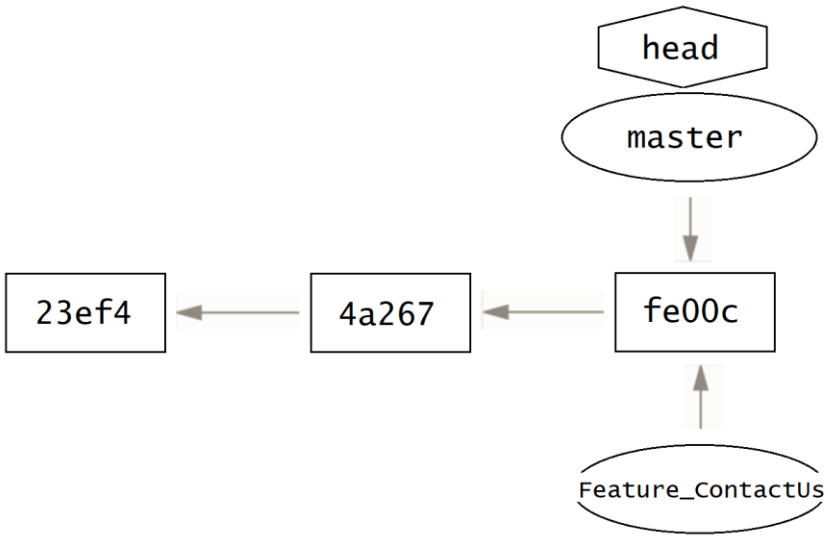
```
$ git branch Feature_ContactUs
  ... still in previous branch ...
```

- **However, it doesn't checkout (switch) to the branch**
  - You must checkout the branch in a separate command

What Happens When You Create a Branch?

- **Under the hood, Git maintains a reference called "head" to the branch you are working on**

- **Creating a new branch creates a new reference to the branch you are working on**
  - However, it doesn't switch the head reference to the new branch



Checking Out a Branch: `git checkout`

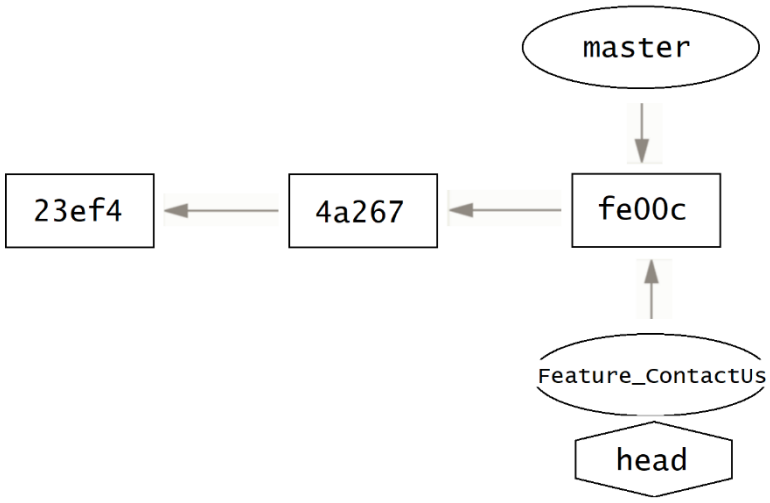
- **Checking out a branch with the `git checkout` command switches the head reference to the checked out branch**
  - The code in the Working Directory is replaced with the code in the branch

Checking out a branch

```
$ git checkout Feature_ContactUs

Switched to branch 'Feature_ContactUs'
```

- **The command can also be used to checkout a specific commit, tag, or file**
  - A tag is essentially a named commit



Creating a Branch and Checking it Out in One Command: `git checkout -b <name>`

- The `git checkout` command with the `-b` flag creates a branch with the specified name and checks it out at the same time

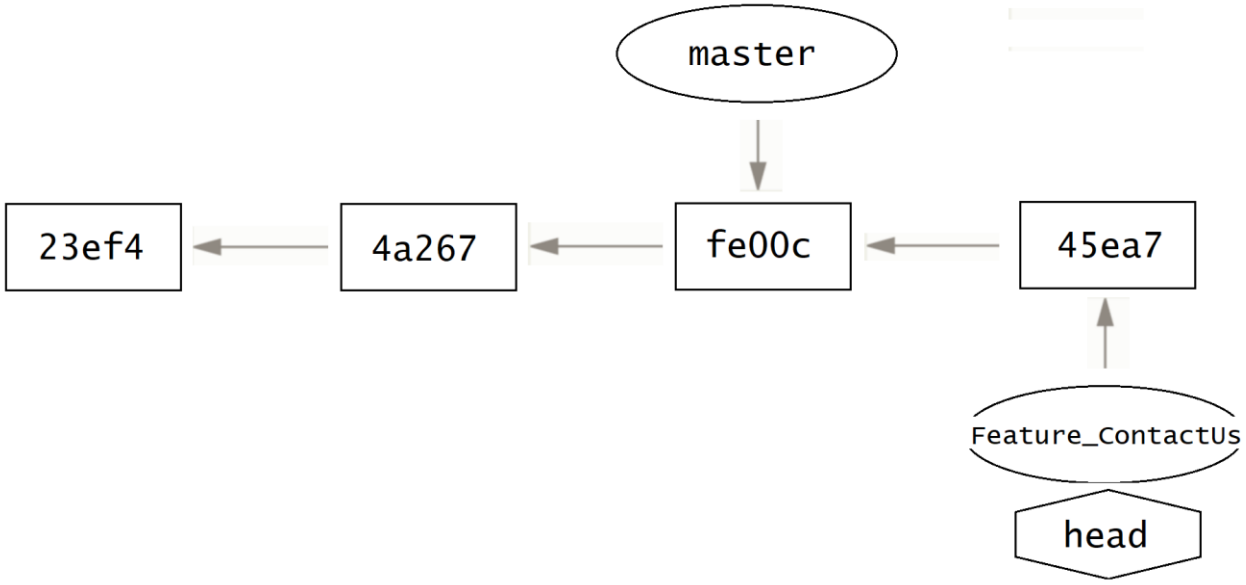
Creating and checking out a branch - One Step!

```
$ git checkout -b Feature_ContactUs

Created and switched to branch 'Feature_ContactUs'
```

Commits on a New Branch

- **When you have a commit on a new branch, your graph resembles:**



Switch -- a new command

- **Git 2.23 added the `git switch` command as a substitute for checking out a branch**
  - The working tree and staging area are updated to match the branch
  - It is essentially checkout with a new name!

Switching to a branch

```
$ git switch Feature_ContactUs
Switched to branch 'Feature_ContactUs'
```

- **The `-c` flag lets you create a branch and then switch to it**

Creating and switching to a branch - One Step!

```
$ git switch -c Feature_ContactUs
Created and switched to branch 'Feature_ContactUs'
```

- **Under the hood, it is the same as:**

```
$ git branch Feature_ContactUs
$ git checkout Feature_ContactUs
```

Viewing all Branches: `git branch`

- The `git branch` command shows you what branches exist in your repository and what branch you are on
  - Your current branch is marked with a `*`

Viewing the Branches

```
$ git branch
  Feature_ContactUs
  Feature_ProductsPage
  Feature_CheckoutPage
  Feature_ContactManufacturer
  * main
```

Delete a Branch: `git branch -d`

- If you create an experimental branch and decide you don't want to keep it, you can delete the branch using the `git branch -d` command

Deletes a local branch from your local repository

```
$ git branch -d Feature_ContactManufacturer

Deleted branch Feature_ContactManufacturer (was a6adcc2).
```

Section 1–3

Merging Branches

Merge Branches: `git merge <name>`

- The `git merge` command merges the named branch into the branch you are currently on

- All the changes that exist in the branch you are merging from now exist in your branch
  - Assuming there are no merge conflicts!

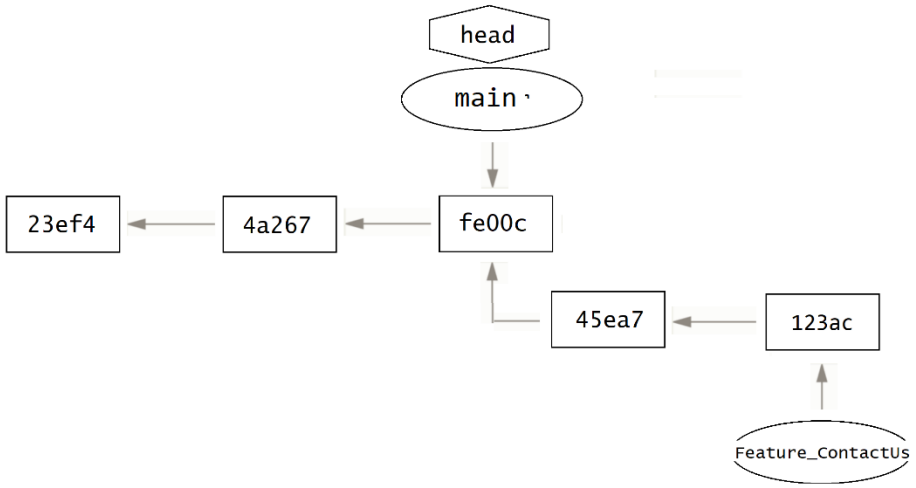
Merging a branch into main

```
$ git switch main
$ git merge Feature_ContactUs
Updating a6adcc2..a71b4f2
Fast-forward
 contact.html | 9 +++++++++
 1 file changed, 9 insertions(+)
```

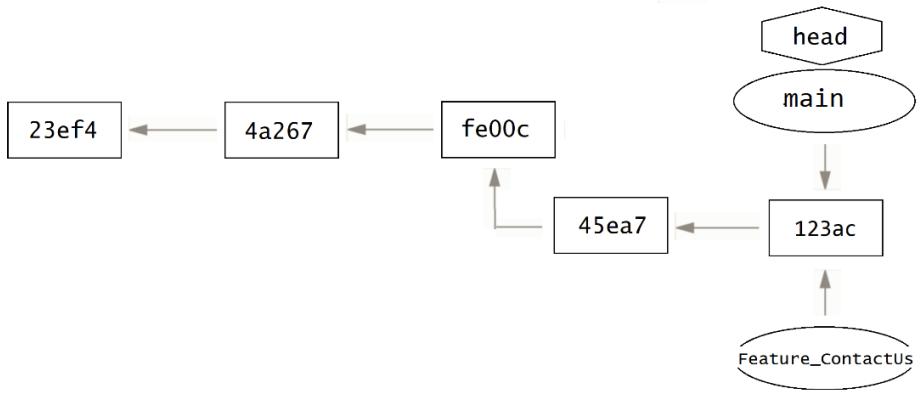
Fast-Forward Merge

- **A fast-forward merge occurs when the path from the current branch tip to the target branch is linear**
  - To merge the branches, all Git has to do to integrate the histories is move (i.e., "fast forward") the current branch tip up to the target branch tip

BEFORE MERGE



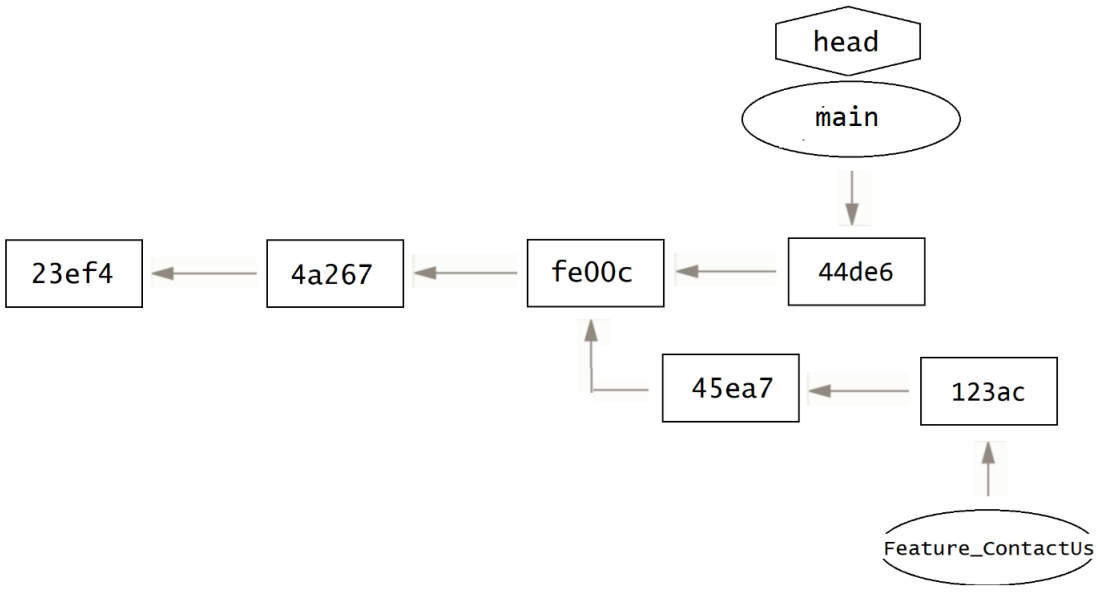
AFTER MERGE



Merge Conflicts

- **A merge conflict can happen when you try to merge two branches that have made edits to the same line in a file**
  - It can also happen when a file has been deleted in one branch but edited in the other

- **Conflicts usually happens when many developers are working on the same project**
  - In the diagram below, someone has done a commit on the main branch that contains changes that the Feature_ContactUs branch doesn't know about



- **In many cases, Git will figure out how to integrate new changes (it is incredibly intelligent!)**
  - But sometimes you have to resolve the conflict yourself

- **When Git can't resolve the conflict, it actually edits the conflicted files and injects markers to help you see/resolve the conflicts**
  - Use the `cat` command to see the file or examine it in an IDE like IntelliJ or Visual Studio Code

Using git status to show conflicts

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

Resolving Merge Conflicts

- **When you issue the merge command the Git discovers conflicts, it marks the problem area *in the file* by enclosing it in "<<<<<<< HEAD" and ">>>>>>> [*other_branch_name*]"**

Example of file contents showing merge conflict markers

```
1  <<<<<<< HEAD
2  This line was committed while working in the "login-box" branch.
3  =======
4  This line, in contrast, was committed while working in the "contact-form" branch.
5  >>>>>>> refs/heads/contact-form
```

- **Examine contents of the conflicted file(s)**
  - The contents after the <<<< angle brackets represent the current working branch (HEAD)
  - A line with "=======" separates the two difference
  - The name of the branch where the conflicts came from is listed after the >>>> angle brackets

- **To fix the conflicts, open the file in an editor and figure out what changes you need to make**
  - This might mean typing new code, editing code, or even removing code!
  - You may also have delete files or restore files!

- **Resolving merge conflicts can take a few minutes or they can take days if there are extensive conflicts!**

- **Fix the conflicts and commit the new changes**

```
$ git add .
$ git commit -m "Merged branch Feature_ContactUs"
```

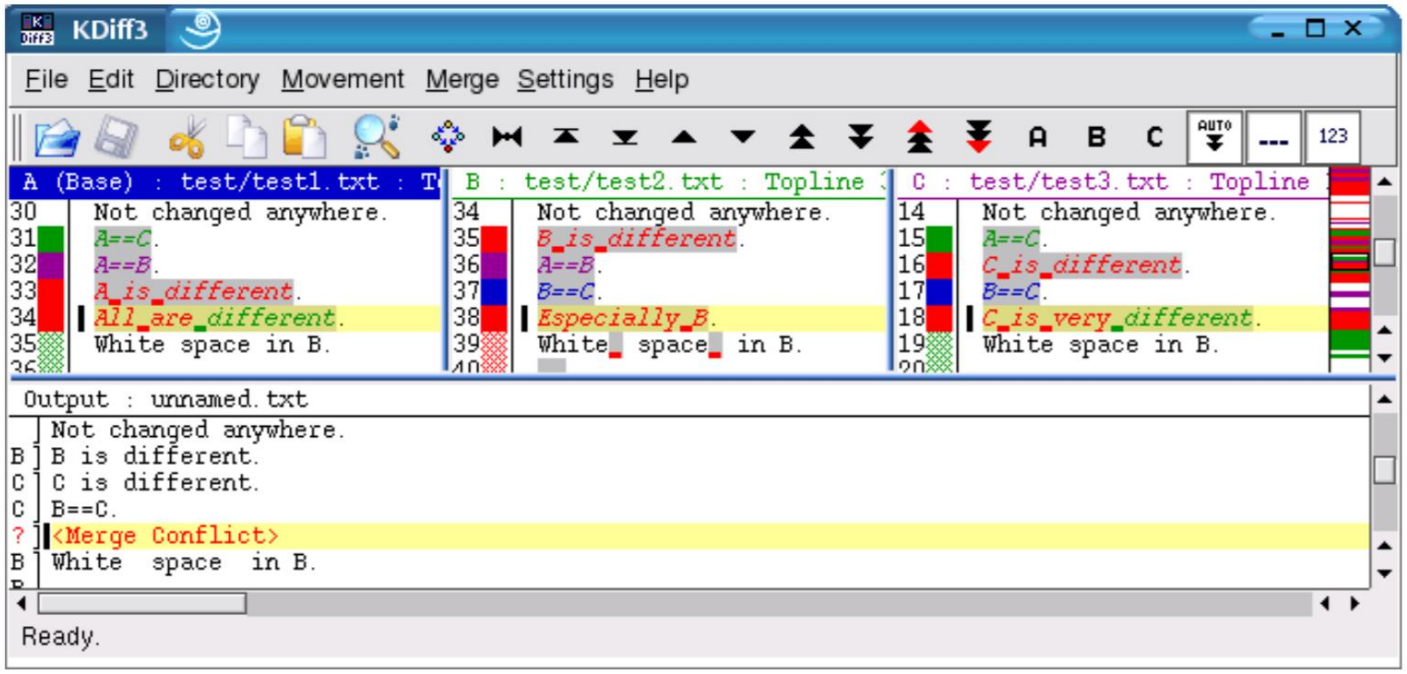
- **If the conflicts are non-trivial, you may want to abort the merge to give you a chance to fix all of the errors**

```
$ git merge --abort
```

Merge Tools

- **You can configure Git merge tools with a sophisticated UI that makes the process easier**
  - One list of tools can be found at: https://www.git-tower.com/blog/diff-tools-windows/
  - Examples of how to configure them can be found at: https://stackoverflow.com/questions/137102/whats-the-best-visual-merge-tool-for-git

- **Example (KDiff3):**



Git Tools

- **GitBash is the command line tool that is installed with Git**

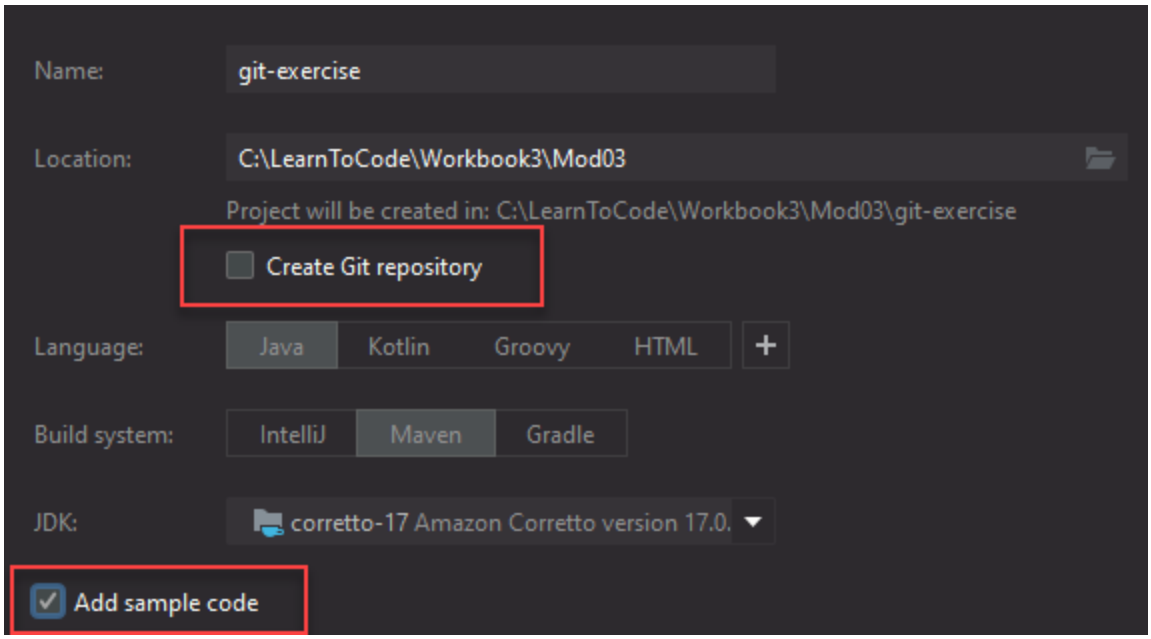
- **Many software tools have been developed to make working with Git easier**
  - Most IDEs have built in support for Git
    * IntelliJ
      https://www.jetbrains.com/help/idea/using-git-integration.html
    * Visual Studio Code
      https://code.visualstudio.com/docs/sourcecontrol/intro-to-git
  - There are also free and paid tools that are visual tools dedicated to working with Git
    * GitHub Destop
    * GitKraken
    * SourceTree
    * Tortoise Git
    * Git Tower
    * https://blog.devart.com/best-git-gui-clients-for-windows.html

Exercise

In this exercise, you will practice branching and merging inside your repository. We will also be a little less specific with directions as you get more comfortable with using Git.

Complete your work in the `workbook-6` folder.

Step 1: Use IntelliJ to create a new Java Project named `git-exercise`. Use the following configuration.



Run your application to verify that the default Hello World code works.

Step 2: Create a branch for a feature

We will modify the code to customize the greeting. You will create a branch to complete this feature. In reality, this is probably too small of a task to have its own branch - but we are just practicing.

Run `git branch CustomizeGreeting` to create a branch called CustomizeGreeting.

Run `git switch CustomizeGreeting` to checkout the new branch.

Run `git status` and notice that you are now on the CustomizeGreeting branch

```
On branch CustomizeGreeting

nothing to commit, working directory clean
```

Step 3: Edit `Main.java`

Open up `Main.java` and edit the code to prompt the user for their name and then display `"Hello <your name>!"`.

Run `git status` to show that we are on our branch and that the `Main.java` file has been modified.

```
On branch CustomizeGreeting

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:   src/main/java/com/pluralsight/Main.java

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

Step 3: Commit the changes

Run `git add -A` to stage this file for commit

Run `git commit -m "added custom user greeting"`

Run `git log` and now you can see that a new commit has been added

Step 4: Explore the differences in the main branch and your feature branch

Checkout the main branch with `git switch main`

Now, look at the contents of `Main.java`. You will notice the header you added is not there! This is because that change only lives on the CustomizeGreeting branch.

Checkout the CustomizeGreeting branch, `git switch CustomizeGreeting`, and you should see your code changes again.

Step 5: Merge the feature branch into main

Now checkout main, `git switch main`.

Merge the CustomizeGreeting branch into the main branch by running the command: `git merge CustomizeGreeting -m "merged CustomizeGreeting branch"`.

NOTE: If you forget the `-m` and your message when committing, Git may popup a text editor for you to provide a message for the merge commit. If this happens, (1) hit `i` on your keyboard and enter the message "merged AddHeader branch", (2) hit `esc` twice, (3) hold `shift` and hit `z` twice.

Now open the `Main.java` file while on the main branch and notice the custom greeting has been added.

You can check out the commit history and you will see how your project has changed over time.

Some organization then delete the feature branch. Other organizations leave it for history purposes. To delete a branch, use:

```
git branch -d CustomizeGreeting
```