TechClass

Mastering Version Control with Git

Advanced part (C)

1st edition

Lecturer: Farhad Eftekhari

Copyright Declaration

Course Links

Course www.techclass.co/courses/git

Slides www.techclass.co/courses/git/slides

Tutorial Videos www.techclass.co/courses/git/videos

Assignments www.techclass.co/courses/git/assignments

Files www.techclass.co/courses/git/files

Note: You need to be accepted in the course to have access to the course material.

Typographical conventions

Further study — At the end of the slides, there will be a link for further study.

Homework — There will be a question concerning the topic in the assignments.

Discussion — A class/pair discussion concerning the topic.

Do it yourself — Do the task yourself to have a hand-on experience.

Practical point — A practical point to consider.

Back in history — Comparison to the methods used to be before.

Be careful — An important issue to consider.

Fun to learn — The topic will not be included in the assignments and the quiz.

Difficulty Level Indicator

Difficulty Level Indicator will be placed on top of the slides.

Easy

Introduction to the material, basic concepts and their purposes.

Moderate

More practical material.

Hard

Advanced and complicated material (mostly these topics are for the students to be familiar with).

Slides Framework

Difficulty indicator

6/67

code

Title

Slide number

Tutorial video code
www.techclass.co/go/code

Content

Course name

Section name

Icons

Content

- Rewriting history
  - git commit –amend
  - git rebase
  - git rebase –i
  - git reflog
- Syncing
  - git remote
  - git fetch
  - git pull
  - git push
- Making a Pull Request
- Using Branches
  - git branch
  - git merge

Definitions

VCS

- Stands for Version Control System

SCM

- Stands for Source Code Management

RCS

- Stands for Revision Control System

CI

- Stands for Continuous Integration

IDE

- Stands for Integrated Development Environments

Rewriting history (1/2)

Git's main job is to make sure you never lose a committed change. But, it's also designed to give you total control over your development workflow. This includes letting you define exactly what your project history looks like; however, it also creates the potential to lose commits. Git provides its history-rewriting commands under the disclaimer that using them may result in lost content.

Rewriting history (2/2)

git commit –amend

git rebase

git rebase -i

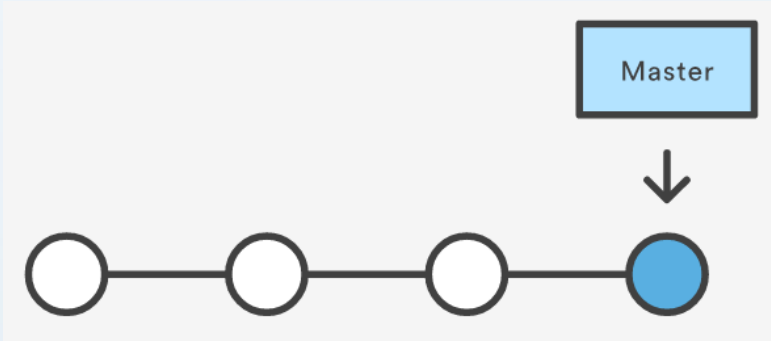
git reflog

git commit --amend (1/4)

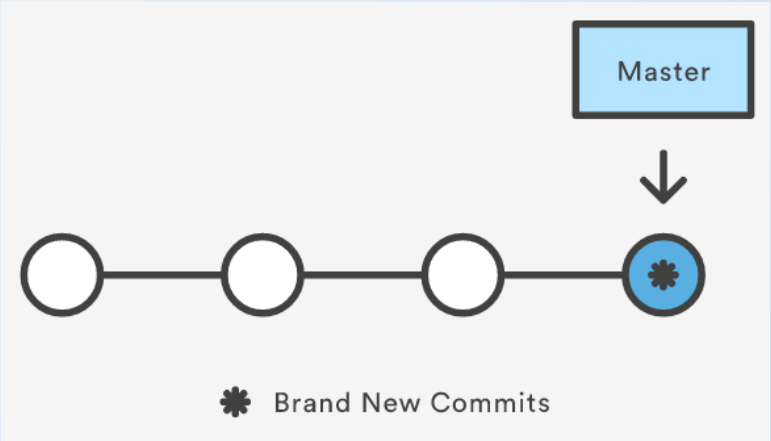
The git commit --amend command is a convenient way to fix up the most recent commit. It lets you combine staged changes with the previous commit instead of committing it as an entirely new snapshot. It can also be used to simply edit the previous commit message without changing its snapshot.

git commit --amend (2/4)

Initial History



Amended History



amending doesn't just alter the most recent commit—it replaces it entirely. To Git, it will look like a brand new commit, which is visualized with an asterisk (*) in the diagram above. It's important to keep this in mind when working with public repositories.

git commit --amend (3/4)

git commit --amend

Combine the staged changes with the previous commit and replace the previous commit with the resulting snapshot. Running this when there is nothing staged lets you edit the previous commit's message without altering its snapshot.

git commit --amend (4/4)

Don't Amend Public Commits

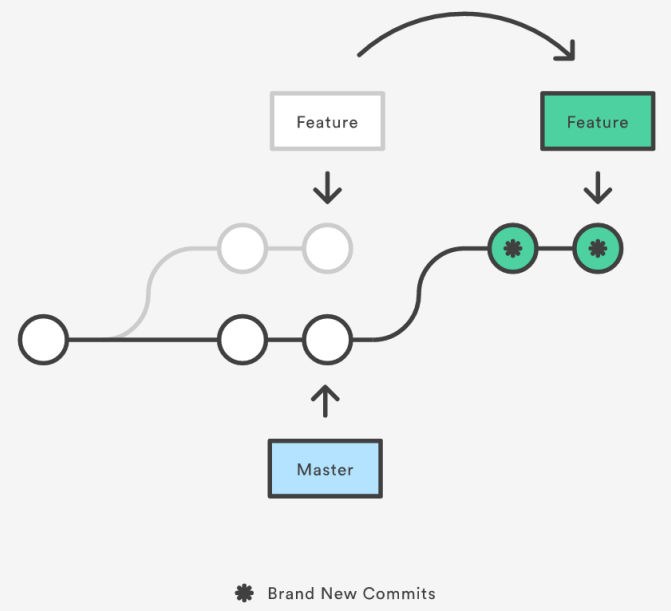
On the git reset page, we talked about how you should never reset commits that have been shared with other developers. The same goes for amending: never amend commits that have been pushed to a public repository.

Amended commits are actually entirely new commits, and the previous commit is removed from the project history. This has the same consequences as resetting a public snapshot. If you amend a commit that other developers have based their work on, it will look like the basis of their work vanished from the project history. This is a confusing situation for developers to be in and it's complicated to recover from.

git rebase (1/5)

Rebasing is the process of moving a branch to a new base commit. The general process can be visualized as the following:

From a content perspective, rebasing really is just moving a branch from one commit to another. But internally, Git accomplishes this by creating new commits and applying them to the specified base—it's literally rewriting your project history. It's very important to understand that, even though the branch looks the same, it's composed of entirely new commits.



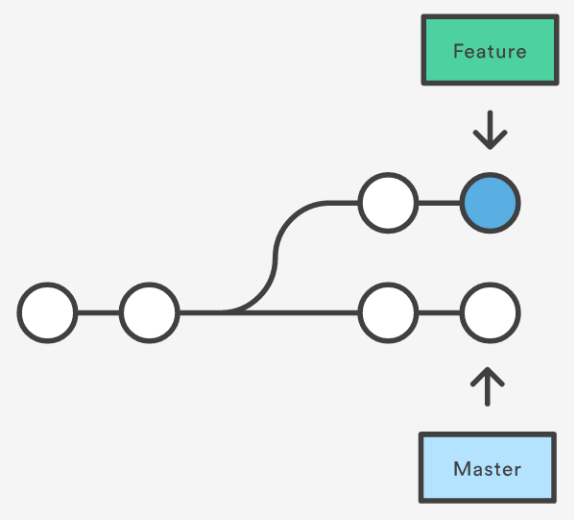
git rebase (2/5)

git rebase <base>

Rebase the current branch onto <base>, which can be any kind of commit reference (an ID, a branch name, a tag, or a relative reference to HEAD).

git rebase (3/5)

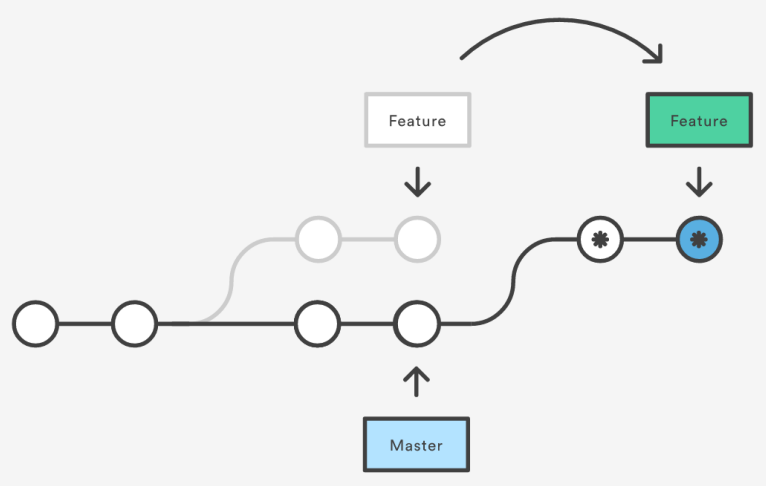
The primary reason for rebasing is to maintain a linear project history. For example, consider a situation where the master branch has progressed since you started working on a feature:



git rebase (4/5)

You have two options for integrating your feature into the master branch: merging directly or rebasing and then merging. The former option results in a 3-way merge and a merge commit, while the latter results in a fast-forward merge and a perfectly linear history. The following diagram demonstrates how rebasing onto master facilitates a fast-forward merge.

After Rebasing onto Master



After a Fast-Forward Merge

git rebase (5/5)

Don't Rebase Public History

As we've discussed with git commit --amend and git reset, you should never rebase commits that have been pushed to a public repository. The rebase would replace the old commits with new ones, and it would look like that part of your project history abruptly vanished.

git rebase -i (1/2)

Running git rebase with the -i flag begins an interactive rebasing session. Instead of blindly moving all of the commits to the new base, interactive rebasing gives you the opportunity to alter individual commits in the process. This lets you clean up history by removing, splitting, and altering an existing series of commits. It's like git commit --amend on steroids.

git rebase -i (2/2)

git rebase -i <base>

Rebase the current branch onto <base>, but use an interactive rebasing session. This opens an editor where you can enter commands (described below) for each commit to be rebased. These commands determine how individual commits will be transferred to the new base. You can also reorder the commit listing to change the order of the commits themselves.

git reflog (1/2)

Git keeps track of updates to the tip of branches using a mechanism called reflog. This allows you to go back to changesets even though they are not referenced by any branch or tag. After rewriting history, the reflog contains information about the old state of branches and allows you to go back to that state if necessary.

git reflog (2/2)

git reflog

Show the reflog for the local repository.

git reflog --relative-date

Show the reflog with relative date information (e.g. 2 weeks ago).

Synching

git remote

git fetch

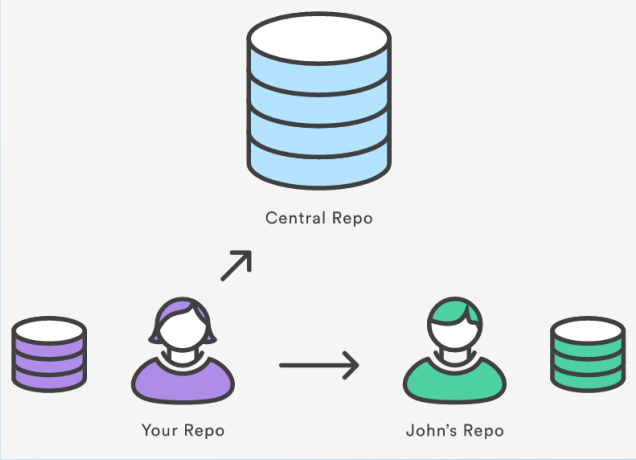
git pull

git push

git remote (1/6)

The git remote command lets you create, view, and delete connections to other repositories. Remote connections are more like bookmarks rather than direct links into other repositories. Instead of providing real-time access to another repository, they serve as convenient names that can be used to reference a not-so-convenient URL.

For example, the following diagram shows two remote connections from your repo into the central repo and another developer's repo. Instead of referencing them by their full URLs, you can pass the origin and john shortcuts to other Git commands.



git remote (2/6)

Git is designed to give each developer an entirely isolated development environment. This means that information is not automatically passed back and forth between repositories. Instead, developers need to manually pull upstream commits into their local repository or manually push their local commits back up to the central repository. The git remote command is really just an easier way to pass URLs to these "sharing" commands.

git remote (3/6)

The origin Remote

When you clone a repository with git clone, it automatically creates a remote connection called origin pointing back to the cloned repository. This is useful for developers creating a local copy of a central repository, since it provides an easy way to pull upstream changes or publish local commits. This behavior is also why most Git-based projects call their central repository origin.

git remote (4/6)

Repository URLs

Git supports many ways to reference a remote repository. Two of the easiest ways to access a remote repo are via the HTTP and the SSH protocols. HTTP is an easy way to allow anonymous, read-only access to a repository. For example:

http://host/path/to/repo.git

But, it's generally not possible to push commits to an HTTP address (you wouldn't want to allow anonymous pushes anyways). For read-write access, you should use SSH instead:

ssh://user@host/path/to/repo.git

git remote (5/6)

git remote

List the remote connections you have to other repositories.

git remote -v

Same as the above command, but include the URL of each connection.

git remote add <name> <url>

Create a new connection to a remote repository. After adding a remote, you'll be able to use <name> as a convenient shortcut for <url> in other Git commands.

git remote (6/6)

git remote rm <name>

Remove the connection to the remote repository called <name>.

git remote rename <old-name> <new-name>

Rename a remote connection from <old-name> to <new-name>.

git fetch (1/2)

The git fetch command imports commits from a remote repository into your local repo. The resulting commits are stored as remote branches instead of the normal local branches that we've been working with. This gives you a chance to review changes before integrating them into your copy of the project.

git fetch (2/2)

git fetch <remote>

Fetch all of the branches from the repository. This also downloads all of the required commits and files from the other repository.

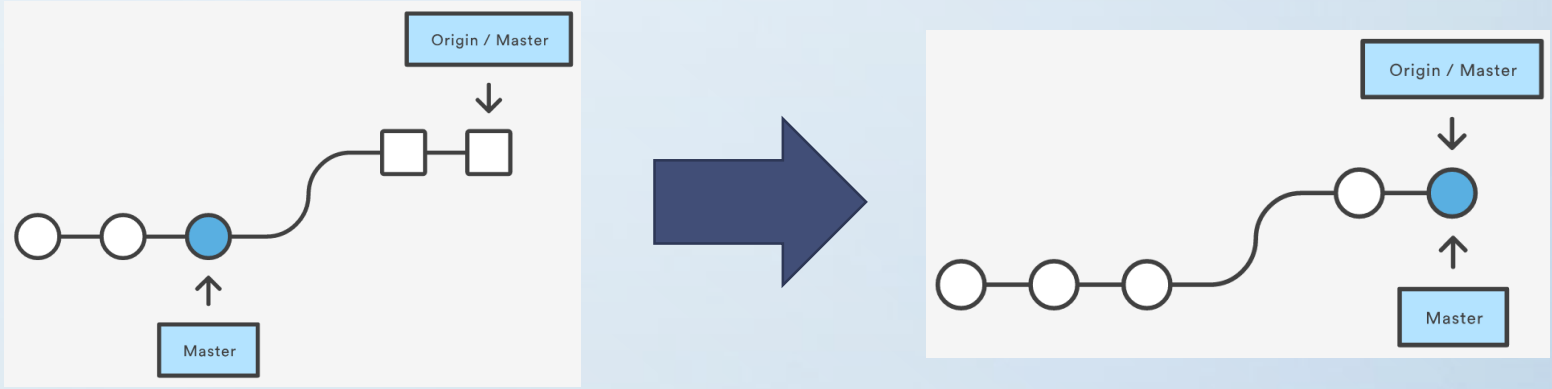
git fetch <remote> <branch>

Same as the above command, but only fetch the specified branch.

git pull (1/2)

Merging upstream changes into your local repository is a common task in Git-based collaboration workflows. We already know how to do this with git fetch followed by git merge, but git pull rolls this into a single command.

You can think of git pull as Git's version of svn update. It's an easy way to synchronize your local repository with upstream changes. The following diagram explains each step of the pulling process.



git pull (2/2)

git pull <remote>

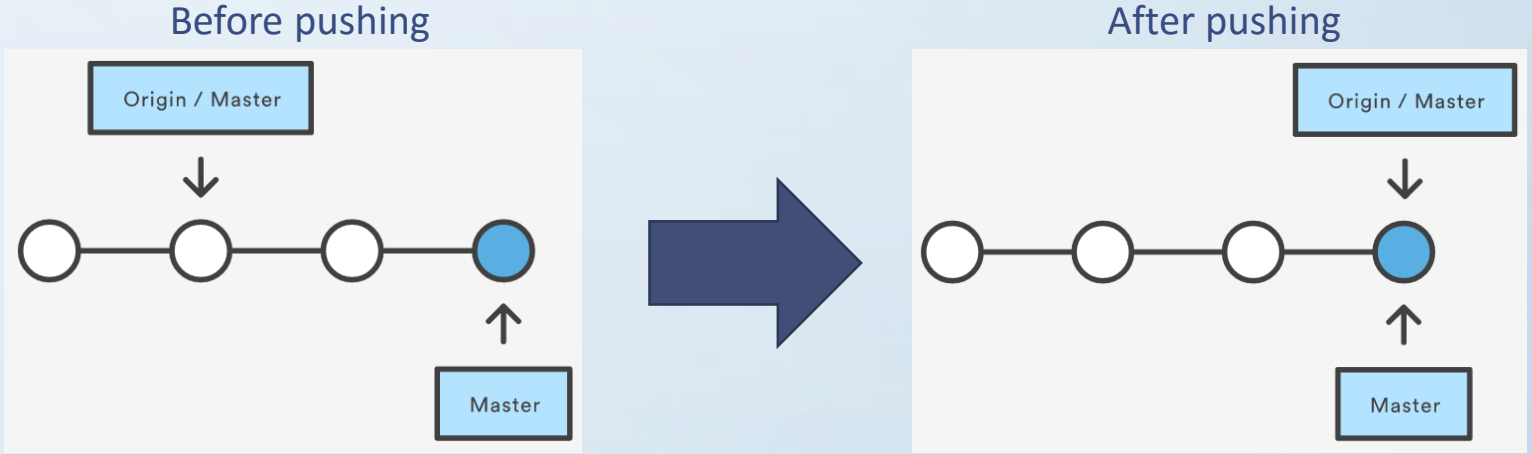
Fetch the specified remote's copy of the current branch and immediately merge it into the local copy. This is the same as git fetch <remote> followed by git merge origin/<current-branch>.

git pull --rebase <remote>

Same as the above command, but instead of using git merge to integrate the remote branch with the local one, use git rebase.

git push (1/4)

Pushing is how you transfer commits from your local repository to a remote repo. The most common use case for git push is to publish your local changes to a central repository. After you've accumulated several local commits and are ready to share them with the rest of the team, you (optionally) clean them up with an interactive rebase, then push them to the central repository.



git push (2/4)

Force Pushing

Git prevents you from overwriting the central repository's history by refusing push requests when they result in a non-fast-forward merge. So, if the remote history has diverged from your history, you need to pull the remote branch and merge it into your local one, then try pushing again. This is similar to how SVN makes you synchronize with the central repository via svn update before committing a changeset.

The --force flag overrides this behavior and makes the remote repository's branch match your local one, deleting any upstream changes that may have occurred since you last pulled. The only time you should ever need to force push is when you realize that the commits you just shared were not quite right and you fixed them with a git commit --amend or an interactive rebase. However, you must be absolutely certain that none of your teammates have pulled those commits before using the --force option.

git push (3/4)

git push <remote> <branch>

Push the specified branch to <remote>, along with all of the necessary commits and internal objects. This creates a local branch in the destination repository. To prevent you from overwriting commits, Git won't let you push when it results in a non-fast-forward merge in the destination repository.

git push <remote> --all

Push all of your local branches to the specified remote.

git push (4/4)

git push <remote> --force

Same as the above command, but force the push even if it results in a non-fast-forward merge. Do not use the --force flag unless you're absolutely sure you know what you're doing.

git push <remote> --tags

Tags are not automatically pushed when you push a branch or use the --all option. The --tags flag sends all of your local tags to the remote repository.

Creating a Pull Request (1/10)

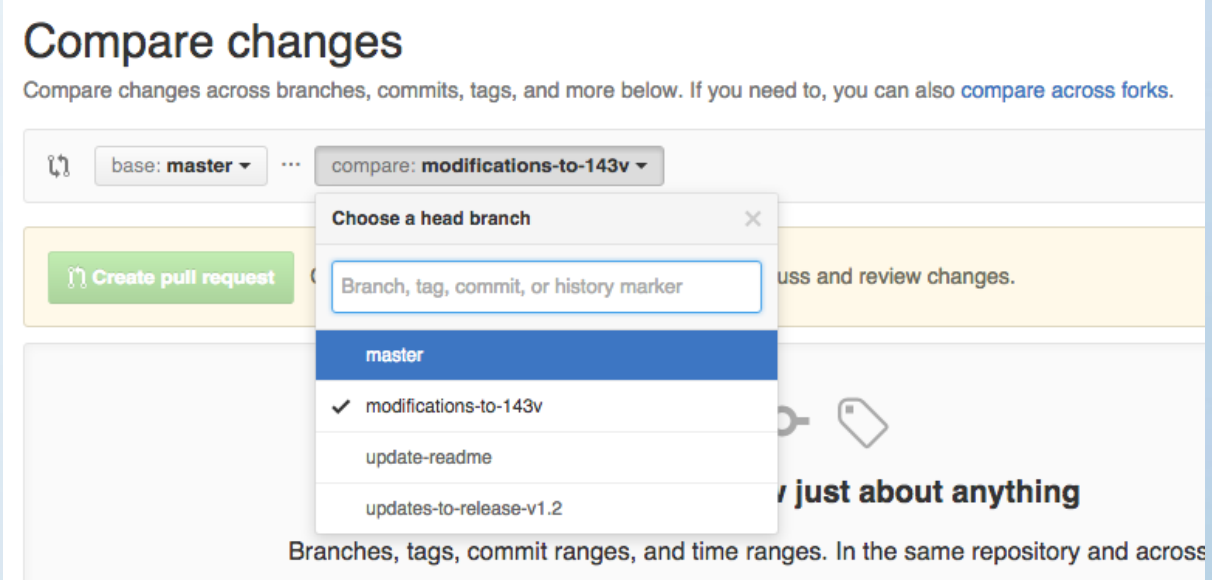
Create a pull request to propose and collaborate on changes to a repository. These changes are proposed in a branch, which ensures that the **master** branch only contains finished and approved work.

Pull requests can only be opened if there are differences between your branch and the upstream branch. You can specify which branch you'd like to merge your changes into when you create your pull request.

Creating a Pull Request (2/10)

By default, pull requests are based on the parent repository's default branch.

If the default parent repository isn't correct, you can change both the parent repository and the branch with the drop-down lists. You can also swap your head and base branches with the drop-down lists to establish diffs between reference points. References here must be branch names in your GitHub repository.



Creating a Pull Request (3/10)

When thinking about branches, remember that the base branch is where changes should be applied, the head branch contains what you would like to be applied.

When you change the base repository, you also change notifications for the pull request. Everyone that can push to the base repository will receive an email notification and see the new pull request in their dashboard the next time they sign in.

When you change any of the information in the branch range, the Commit and Files changed preview areas will update to show your new range.

Creating a Pull Request (4/10)

Steps:

1. On GitHub, navigate to the main page of the repository.
2. Branch dropdown menu in the "Branch" menu, choose the branch that contains your commits.
3. Pull Request button to the right of the Branch menu, click New pull request.
4. Drop-down menus for choosing the base and compare branches use the base branch dropdown menu to select the branch you'd like to merge your changes into, then use the compare branch drop-down menu to choose the topic branch you made your changes in.
5. Pull request title and description fields type a title and description for your pull request.
6. Create pull request button click Create pull request.

Creating a Pull Request - Steps (5/10)

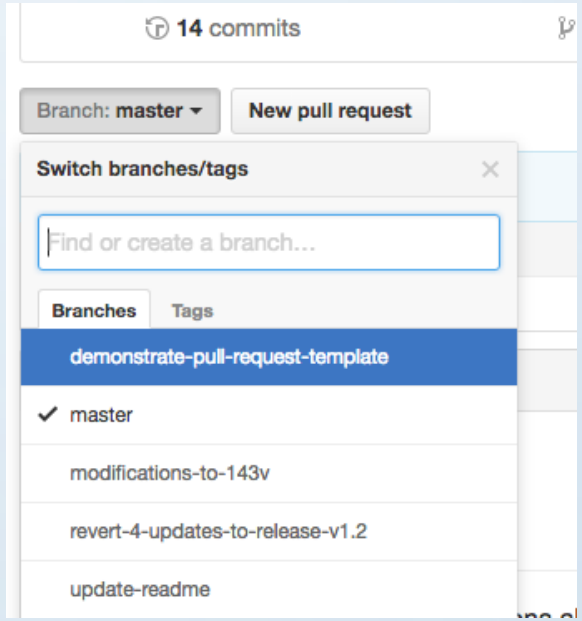
Step 1

- On GitHub, navigate to the main page of the repository.

Creating a Pull Request - Steps (6/10)

Step 2

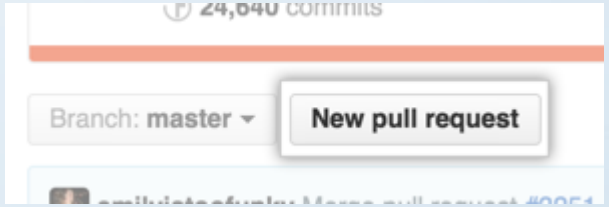
- Branch dropdown menu In the "Branch" menu, choose the branch that contains your commits.



Creating a Pull Request - Steps (7/10)

Step 3

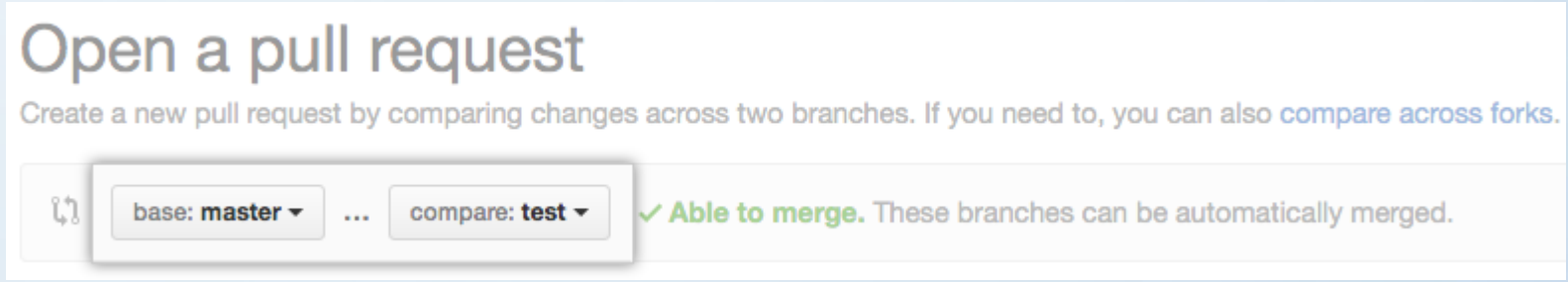
- Pull Request button to the right of the Branch menu, click New pull request.



Creating a Pull Request - Steps (8/10)

Step 4

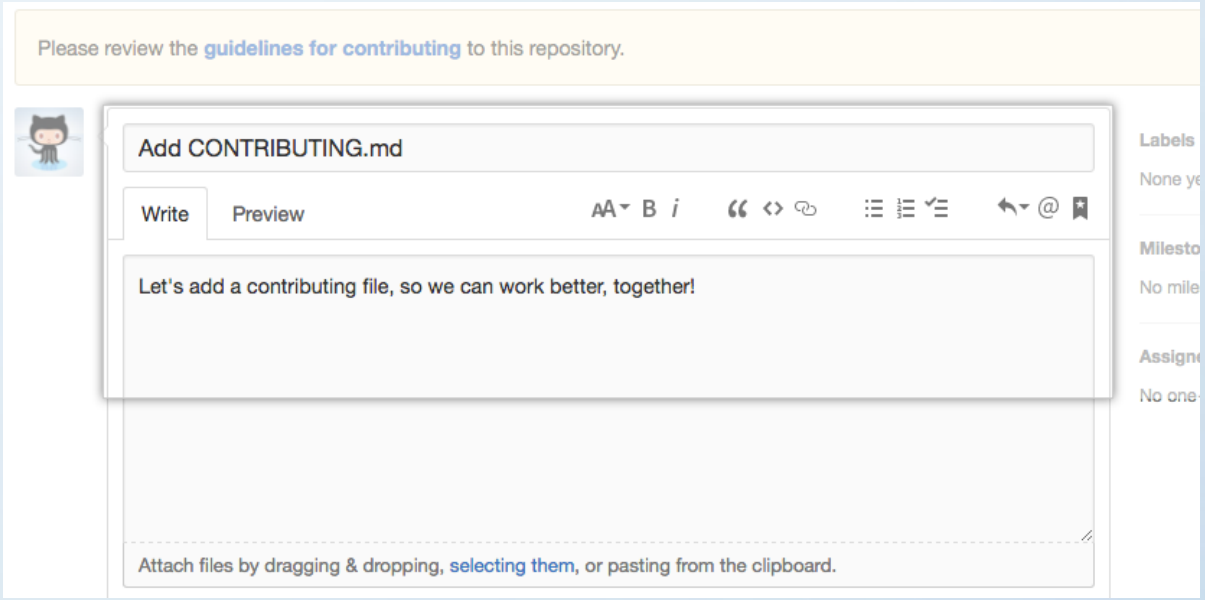
- Drop-down menus for choosing the base and compare branchesUse the base branch dropdown menu to select the branch you'd like to merge your changes into, then use the compare branch drop-down menu to choose the topic branch you made your changes in.



Creating a Pull Request - Steps (9/10)

Step 5

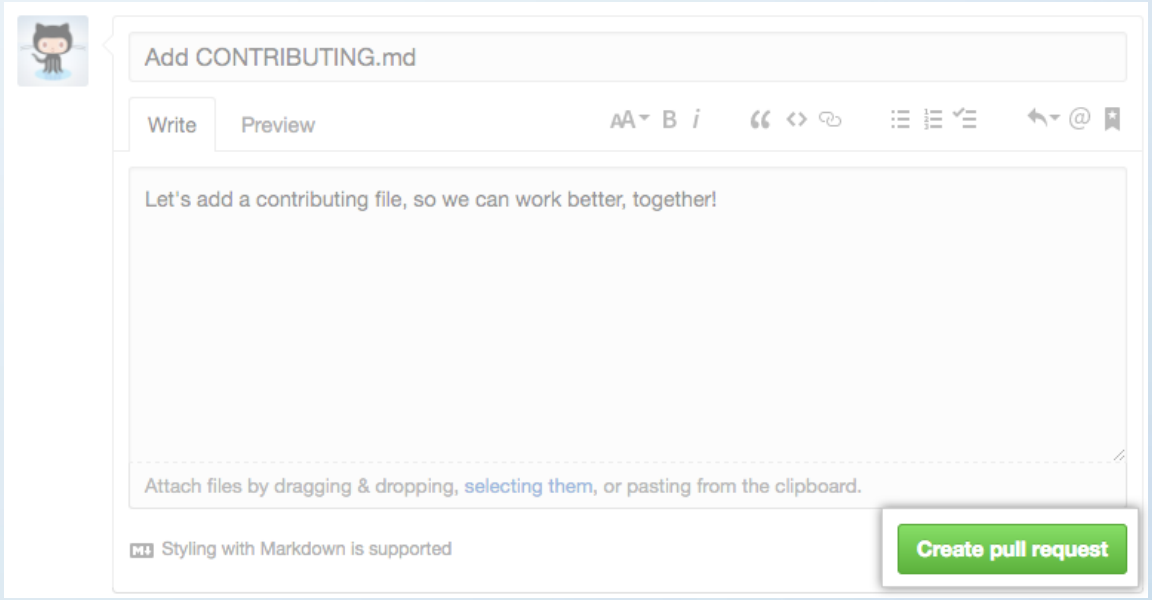
- Pull request title and description fieldsType a title and description for your pull request.



Creating a Pull Request - Steps (10/10)

Step 6

- Pull request title and description fields type a title and description for your pull request.



Using Branches

git branch

git merge

git branch (1/5)

A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process. You can think of them as a way to request a brand new working directory, staging area, and project history. New commits are recorded in the history for the current branch, which results in a fork in the history of the project.

The git branch command lets you create, list, rename, and delete branches. It doesn't let you switch between branches or put a forked history back together again. For this reason, git branch is tightly integrated with the git checkout and git merge commands.

git branch (2/5)

git branch

List all of the branches in your repository.

git branch <branch>

Create a new branch called <branch>. This does not check out the new branch.

git branch -d <branch>

Delete the specified branch. This is a “safe” operation in that Git prevents you from deleting the branch if it has unmerged changes.

git branch (3/5)

git branch -D <branch>

Force delete the specified branch, even if it has unmerged changes. This is the command to use if you want to permanently throw away all of the commits associated with a particular line of development.

git branch -m <branch>

Rename the current branch to <branch>.

git branch (4/5)

git branch

List all of the branches in your repository.

git branch <branch>

Create a new branch called <branch>. This does not check out the new branch.

git branch -d <branch>

Delete the specified branch. This is a “safe” operation in that Git prevents you from deleting the branch if it has unmerged changes.

git branch (5/5)

git branch -D <branch>

Force delete the specified branch, even if it has unmerged changes. This is the command to use if you want to permanently throw away all of the commits associated with a particular line of development.

git branch -m <branch>

Rename the current branch to <branch>.

git checkout (1/2)

The git checkout command lets you navigate between the branches created by git branch. Checking out a branch updates the files in the working directory to match the version stored in that branch, and it tells Git to record all new commits on that branch. Think of it as a way to select which line of development you're working on.

git checkout (2/2)

git checkout <existing-branch>

Check out the specified branch, which should have already been created with git branch. This makes <existing-branch> the current branch, and updates the working directory to match.

git checkout -b <new-branch>

Create and check out <new-branch>. The -b option is a convenience flag that tells Git to run git branch <new-branch> before running git checkout <new-branch>. git checkout -b <new-branch> <existing-branch>

git merge (1/8)

Merging is Git's way of putting a forked history back together again. The git merge command lets you take the independent lines of development created by git branch and integrate them into a single branch.

Once you've finished developing a feature in an isolated branch, it's important to be able to get it back into the main code base. Depending on the structure of your repository, Git has several distinct algorithms to accomplish this: a fast-forward merge or a 3-way merge.

git merge (2/8)

Fast-forward Merge

3-way Merge

git merge (3/8)

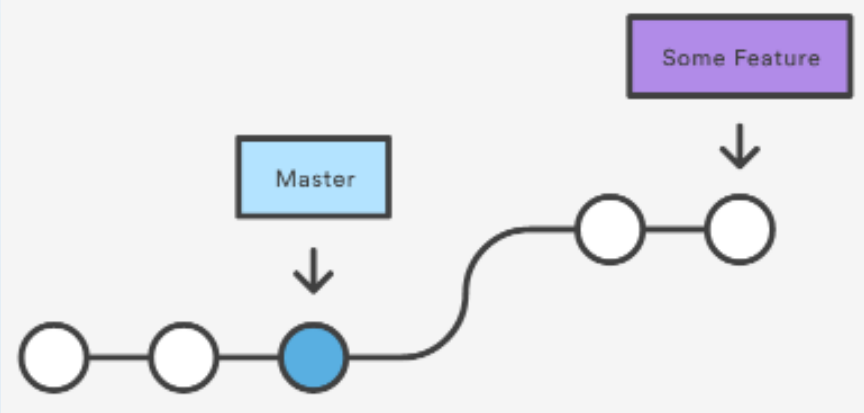
Fast-forward Merge

A fast-forward merge can occur when there is a linear path from the current branch tip to the target branch. Instead of "actually" merging 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. This effectively combines the histories, since all of the commits reachable from the target branch are now available through the current one.

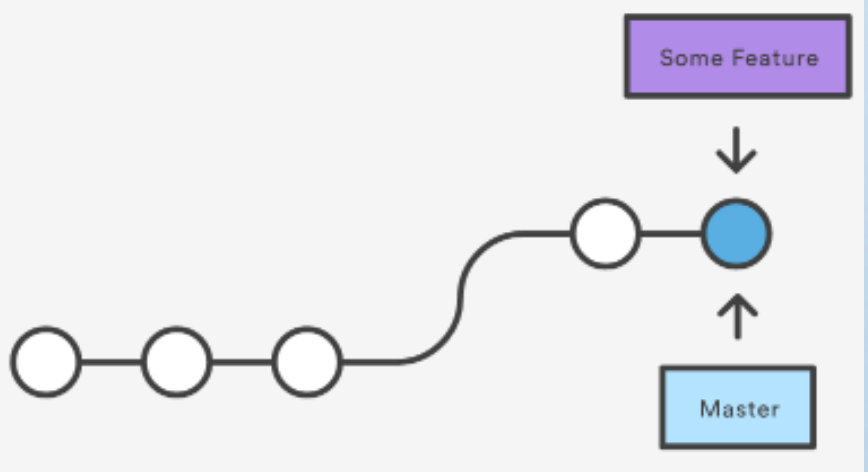
git merge (4/8)

Fast-forward Merge

Before merging



After merging



git merge (5/8)

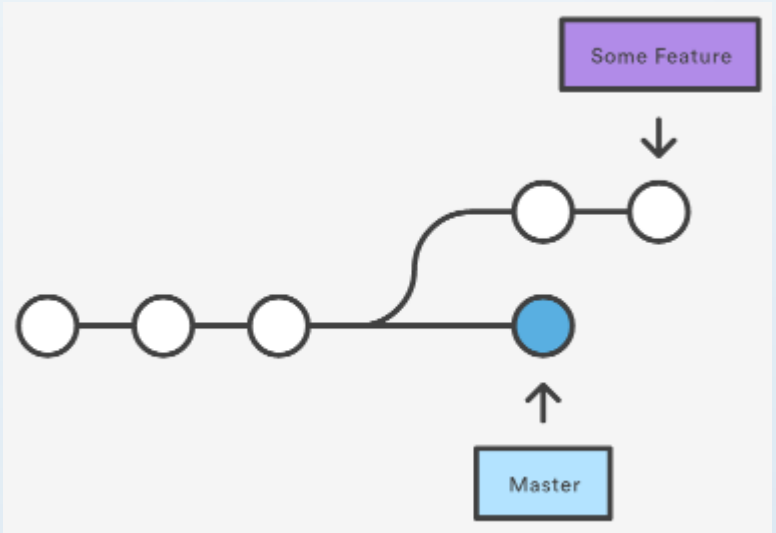
3-way Merge

A fast-forward merge is not possible if the branches have diverged. When there is not a linear path to the target branch, Git has no choice but to combine them via a 3-way merge. 3-way merges use a dedicated commit to tie together the two histories. The nomenclature comes from the fact that Git uses three commits to generate the merge commit: the two branch tips and their common ancestor.

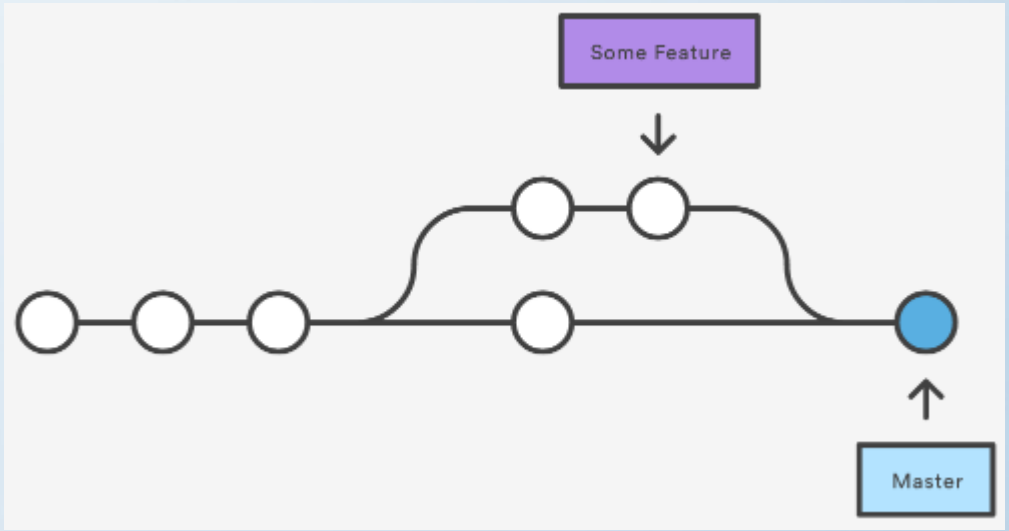
git merge (6/8)

3-way Merge

Before merging



After merging



git merge (7/8)

Resolving Conflicts

If the two branches you're trying to merge both changed the same part of the same file, Git won't be able to figure out which version to use. When such a situation occurs, it stops right before the merge commit so that you can resolve the conflicts manually.

The great part of Git's merging process is that it uses the familiar edit/stage/commit workflow to resolve merge conflicts. When you encounter a merge conflict, running the git status command shows you which files need to be resolved. Then, you can go in and fix up the merge to your liking. When you're ready to finish the merge, all you have to do is run git add on the conflicted file(s) to tell Git they're resolved. Then, you run a normal git commit to generate the merge commit. It's the exact same process as committing an ordinary snapshot, which means it's easy for normal developers to manage their own merges.

Note that merge conflicts will only occur in the event of a 3-way merge. It's not possible to have conflicting changes in a fast-forward merge.

git merge (8/8)

git merge <branch>

Merge the specified branch into the current branch. Git will determine the merge algorithm automatically (discussed below).

git merge --no-ff <branch>

Merge the specified branch into the current branch, but always generate a merge commit (even if it was a fast-forward merge). This is useful for documenting all merges that occur in your repository.

Ignoring Files

Untracked files typically fall into two categories. They‘re either files that have just been added to the project and haven’t been committed yet, or they're compiled binaries like .pyc, .obj, .exe, etc. While it's definitely beneficial to include the former in the git status output, the latter can make it hard to see what’s actually going on in your repository.

For this reason, Git lets you completely ignore files by placing paths in a special file called .gitignore. Any files that you'd like to ignore should be included on a separate line, and the * symbol can be used as a wildcard. For example, adding the following to a .gitignore file in your project root will prevent compiled Python modules from appearing in git status:

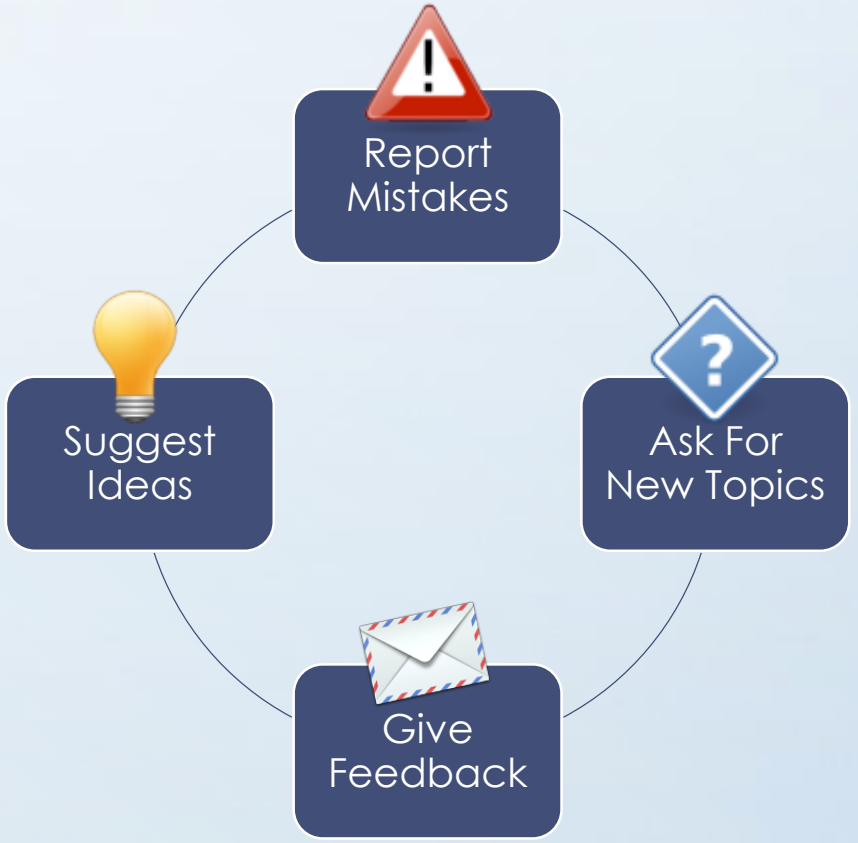
```
*.pyc
```

Further study

1. Git Branching – Rebasing (S15)
   - https://git-scm.com/book/en/v2/Git-Branching-Rebasing
2. Git rebase and the golden rule explained (S15)
   - https://medium.freecodecamp.com/git-rebase-and-the-golden-rule-explained-70715eccc372
3. How to Send a Pull Request using the Github Desktop Client (S39)
   - https://scotch.io/tutorials/how-to-send-a-pull-request-using-the-github-desktop-client#installing-the-github-desktop-client
4. Fast-Forward Git Merge (S59)
   - https://ariya.io/2013/09/fast-forward-git-merge
5. Three-Way Merging: A Look Under the Hood (S61)
   - http://www.drdobbs.com/tools/three-way-merging-a-look-under-the-hood/240164902

Contribute

Feel free to contribute to make the content of the course even more beneficial and practical for all the students!



TechClass

Thank you for your consideration!

I hope you have a wonderful class! ☺

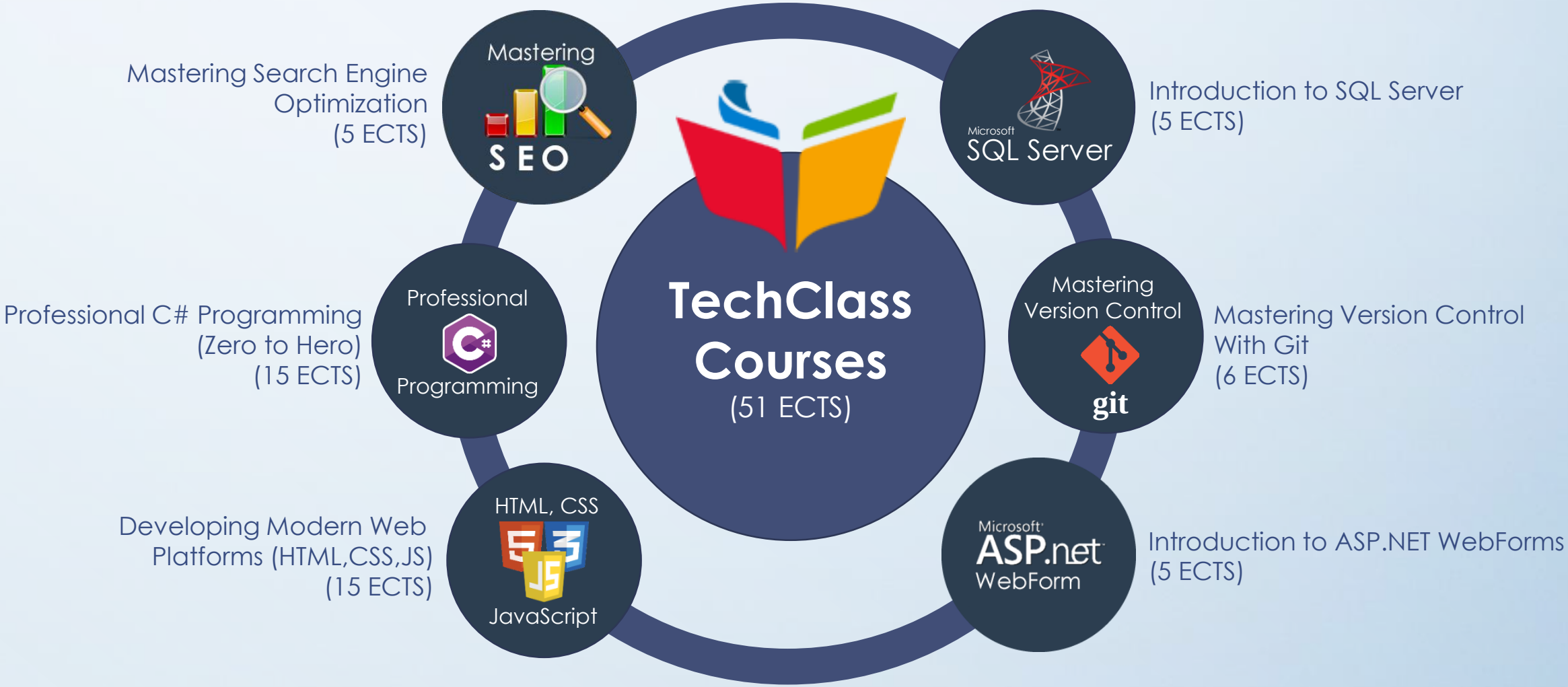
fb.com/techclass

@etechclass

@etechclass

bit.ly/etechclass

Helsinki Metropolia UAS
Bulevardi 31
00079 Heksinki, Finland
www.techclass.co


Mastering Search Engine Optimization (5 ECTS)
Mastering
SEO
Introduction to SQL Server (5 ECTS)
Microsoft
SQL Server
Professional C# Programming (Zero to Hero) (15 ECTS)
Professional
Programming
TechClass Courses
(51 ECTS)
Mastering Version Control With Git (6 ECTS)
Mastering
Version Control
git
Developing Modern Web Platforms (HTML,CSS,JS) (15 ECTS)
HTML, CSS
JavaScript
Introduction to ASP.NET WebForms (5 ECTS)
Microsoft
ASP.net
WebForm