TechClass

Mastering Version Control with Git

Intermediate part (B)

1st edition

Lecturer: Farhad Eftekhari

Copyright Declaration

Course Links

Course www.techclass.co/courses/seo

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

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

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

Files www.techclass.co/courses/seo/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 the topics are for the students to be familiar with, and not completly be able to use them).

Slides Framework

Difficulty indicator

6/70

code

Title

Slide number

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

Content

Course name

Section name

Icons

Content

- Setting up a repository
  - git init
  - git clone
  - git config
- Saving changes
  - git add
  - git commit
  - git stash
- Inspecting a repository
  - git status
  - git log
- Viewing old commits
  - git checkout
- Undoing Changes
  - git checkout
  - git revert
  - git reset
  - git clean

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

Setting up a repository

git init

git clone

git config

git init (1/3)

The git init command creates a new Git repository. It can be used to convert an existing, unversioned project to a Git repository or initialize a new empty repository. Most of the other Git commands are not available outside of an initialized repository, so this is usually the first command you'll run in a new project.

Executing git init creates a .git subdirectory in the project root, which contains all of the necessary metadata for the repo.

git init (2/3)

git init

Transform the current directory into a Git repository. This adds a .git folder to the current directory and makes it possible to start recording revisions of the project.

git init <directory>

Create an empty Git repository in the specified directory. Running this command will create a new folder called <directory containing nothing but the .git subdirectory.

git init (3/3)

git init --bare <directory>

Initialize an empty Git repository, but omit the working directory. Shared repositories should always be created with the --bare flag. Conventionally, repositories initialized with the --bare flag end in .git. For example, the bare version of a repository called my-project should be stored in a directory called my-project.git.

git init vs. git init –bare (1/2)

What is the difference between a repository created using the git init command and the git init --bare command?

Repositories created with the git init command are called working directories. In the top level folder of the repository you will find two things:

1. A .git subfolder with all the git related revision history of your repo
2. A working tree, or checked out copies of your project files.

git init vs. git init –bare (2/2)

Repositories created with git init --bare are called bare repos. They are structured a bit differently from working directories. First off, they contain no working or checked out copy of your source files. And second, bare repos store git revision history of your repo in the root folder of your repository instead of in a .git subfolder. Note… bare repositories are customarily given a .git extension.

Why use one or the other?

Well, a working repository created with git init is for… working. It is where you will actually edit, add and delete files and git commit to save your changes.

A bare repository created with git init --bare is for… sharing.

(Source: Jon Saints)

git clone (1/2)

The git clone command copies an existing Git repository. This is sort of like svn checkout, except the "working copy" is a full-fledged Git repository—it has its own history, manages its own files, and is a completely isolated environment from the original repository.

As a convenience, cloning automatically creates a remote connection called origin pointing back to the original repository. This makes it very easy to interact with a central repository.

git clone (2/2)

git clone <repo>

Clone the repository located at <repo> onto the local machine. The original repository can be located on the local filesystem or on a remote machine accessible via HTTP or SSH.

git clone <repo> <directory>

Clone the repository located at <repo> into the folder called <directory> on the local machine.

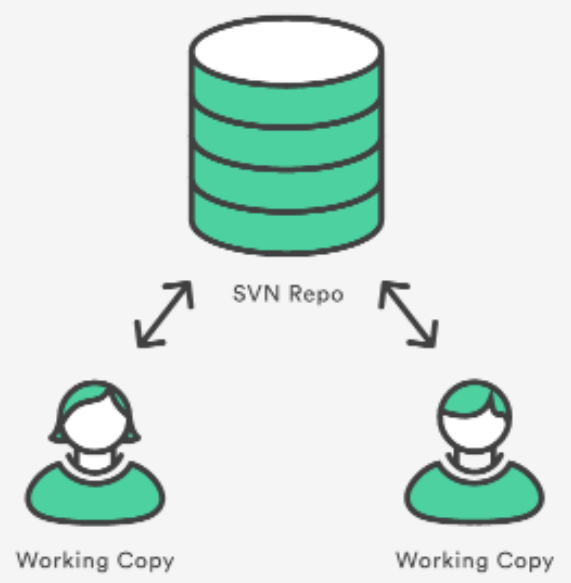
Repo-To-Repo Collaboration (1/2)

It's important to understand that Git's idea of a "working copy" is very different from the working copy you get by checking out code from an SVN repository. Unlike SVN, Git makes no distinction between the working copy and the central repository—they are all full-fledged Git repositories.

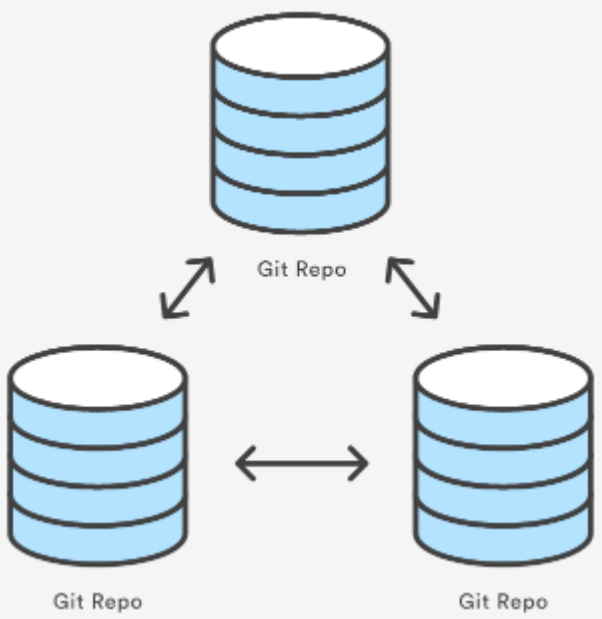
This makes collaborating with Git fundamentally different than with SVN. Whereas SVN depends on the relationship between the central repository and the working copy, Git's collaboration model is based on repository-to-repository interaction. Instead of checking a working copy into SVN's central repository, you push or pull commits from one repository to another.

Repo-To-Repo Collaboration (2/2)

Central-Repo-to-Working-Copy Collaboration



Repo-To-Repo Collaboration



git config (1/4)

The git config command lets you configure your Git installation (or an individual repository) from the command line. This command can define everything from user info to preferences to the behavior of a repository. Several common configuration options are listed below.

git config user.name <name>

Define the author name to be used for all commits in the current repository.

git config --global user.name <name>

Define the author name to be used for all commits by the current user.

git config (2/4)

git config --global user.email <email>

Define the author email to be used for all commits by the current user.

git config --global alias.<alias-name> <git-command>

Create a shortcut for a Git command.

git config --system core.editor <editor>

Define the text editor used by commands like git commit for all users on the current machine. The <editor> argument should be the command that launches the desired editor.

git config --global –edit

Open the global configuration file in a text editor for manual editing.

git config (3/4)

Git stores configuration options in three separate files, which lets you scope options to individual repositories, users, or the entire system:

1. <repo>/.git/config – Repository-specific settings.
2. ~/.gitconfig – User-specific settings. This is where options set with the --global flag are stored.
3. $(prefix)/etc/gitconfig – System-wide settings.

git config - Sample (4/4)

```
[user]
name = John Smith
email = john@example.com
[alias]
st = status
co = checkout
br = branch
up = rebase
ci = commit
[core]
editor = vim
```

You can manually edit these values to the exact same effect as git config.

Saving changes

git add

git commit

git stash

git add (1/3)

The git add command adds a change in the working directory to the staging area. It tells Git that you want to include updates to a particular file in the next commit. However, git add doesn't really affect the repository in any significant way—changes are not actually recorded until you run git commit.

git add (2/3)

git add <file>

Stage all changes in <file> for the next commit.

git add <directory>

Stage all changes in <directory> for the next commit.

git add .

Stage all changes in the repository for the next commit.

git add –p

Begin an interactive staging session that lets you choose portions of a file to add to the next commit. This will present you with a chunk of changes and prompt you for a command. Use y to stage the chunk, n to ignore the chunk, s to split it into smaller chunks, e to manually edit the chunk, and q to exit.

git add (3/3)

The git add command should not be confused with svn add, which adds a file to the repository. Instead, git add works on the more abstract level of changes. This means that git add needs to be called every time you alter a file, whereas svn add only needs to be called once for each file.

git commit (1/3)

The git commit command commits the staged snapshot to the project history. Committed snapshots can be thought of as “safe” versions of a project—Git will never change them unless you explicitly ask it to. Along with git add, this is one of the most important Git commands.

git commit (2/3)

git commit

Commit the staged snapshot. This will launch a text editor prompting you for a commit message. After you've entered a message, save the file and close the editor to create the actual commit.

git commit -m "<message>"

Commit the staged snapshot, but instead of launching a text editor, use <message> as the commit message.

git commit –a

Commit a snapshot of all changes in the working directory. This only includes modifications to tracked files (those that have been added with git add at some point in their history).

git commit (3/3)

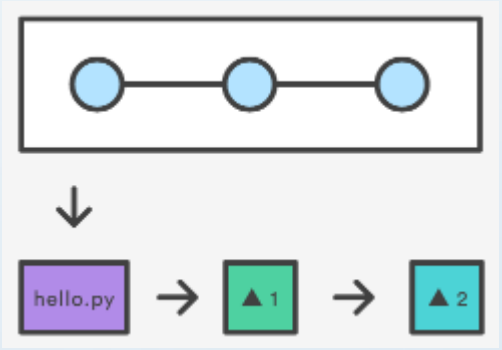
Snapshots are always committed to the local repository. This is fundamentally different from SVN, wherein the working copy is committed to the central repository. In contrast, Git doesn't force you to interact with the central repository until you're ready.

Snapshots, Not Differences (1/2)

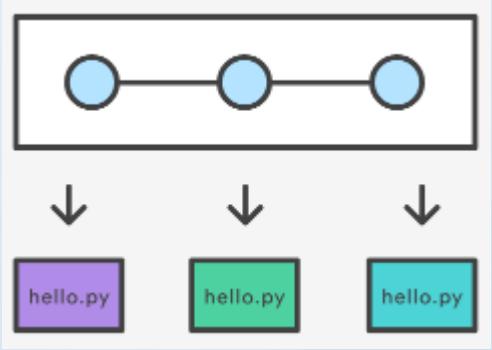
Aside from the practical distinctions between SVN and Git, their underlying implementation also follow entirely divergent design philosophies. Whereas SVN tracks differences of a file, Git's version control model is based on snapshots. For example, an SVN commit consists of a diff compared to the original file added to the repository. Git, on the other hand, records the entire contents of each file in every commit.

Snapshots, Not Differences (2/2)

This makes many Git operations much faster than SVN, since a particular version of a file doesn't have to be "assembled" from its diffs—the complete revision of each file is immediately available from Git's internal database.



Recording File Diffs (SVN)



Recording Snapshots (Git)

git stash

git stash temporarily shelves (or stashes) changes you've made to your working copy so you can work on something else, and then come back and re-apply them later on. Stashing is handy if you need to quickly switch context and work on something else, but you're mid-way through a code change and aren't quite ready to commit.

The git stash command takes your uncommitted changes (both staged and unstaged), saves them away for later use, and then reverts them from your working copy.

git stash - Sample

```
$ git status
On branch master
Changes to be committed:
    new file:   style.css
Changes not staged for commit:
    modified:   index.html
$ git stash
Saved working directory and index state WIP on master: 5002d47 our new homepage
HEAD is now at 5002d47 our new homepage
$ git status
On branch master
nothing to commit, working tree clean
```

At this point you're free to make changes, create new commits, switch branches, and perform any other Git operations; then come back and re-apply your stash when you're ready.

Note that the stash is local to your Git repository; stashes are not transferred to the server when you push.

git stash - Re-applying your stashed changes

You can reapply previously stashed changes with git stash pop:

```
$ git status
On branch master
nothing to commit, working tree clean
$ git stash pop
On branch master
Changes to be committed:
    new file:   style.css
Changes not staged for commit:
    modified:   index.html
Dropped refs/stash@{0} (32b3aa1d185dfe6d57b3c3cc3b32cbf3e380cc6a)
```

Popping your stash removes the changes from your stash and reapplies them to your working copy.

git stash - Re-applying your stashed changes

You can reapply the changes to your working copy and keep them in your stash with git stash apply:

```
$ git stash apply

On branch master

Changes to be committed:

    new file:   style.css

Changes not staged for commit:

    modified:   index.html
```

git stash - Managing multiple stashes

You aren't limited to a single stash. You can run git stash several times to create multiple stashes, and then use git stash list to view them. By default, stashes are identified simply as a "WIP" – work in progress – on top of the branch and commit that you created the stash from.

```
$ git stash list

stash@{0}: WIP on master: 5002d47 our new homepage

stash@{1}: WIP on master: 5002d47 our new homepage

stash@{2}: WIP on master: 5002d47 our new homepage
```

You can choose which stash to re-apply by passing its identifier as the last argument, for example:

```
$ git stash pop stash@{2}
```

git stash - Creating a branch from your stash

If the changes on your branch diverge from the changes in your stash, you may run into conflicts when popping or applying your stash. Instead, you can use git stash branch to create a new branch to apply your stashed changes to:

```
$ git stash branch add-style stash@{1}
Switched to a new branch 'add-stylesheet'
On branch add-stylesheet
Changes to be committed:
    new file:   style.css
Changes not staged for commit:
    modified:   index.html
Dropped refs/stash@{1} (32b3aa1d185dfe6d57b3c3cc3b32cbf3e380cc6a)
```

This checks out a new branch based on the commit that you created your stash from, and then pops your stashed changes onto it.

git stash - Cleaning up your stash

If you decide you no longer need a particular stash, you can delete it with git stash drop:

```
$ git stash drop stash@{1}
```

```
Dropped stash@{1}
(17e2697fd8251df6163117cb3d58c1f62a5e7cdb)
```

Or you can delete all of your stashes with:

```
$ git stash clear
```


Inspecting a repository

git status

git log

git status (1/3)

The git status command displays the state of the working directory and the staging area. It lets you see which changes have been staged, which haven't, and which files aren't being tracked by Git. Status output does not show you any information regarding the committed project history. For this, you need to use git log.

git status

List which files are staged, unstaged, and untracked.

git status (2/3)

The git status command is a relatively straightforward command. It simply shows you what's been going on with git add and git commit. Status messages also include relevant instructions for staging/unstaging files. Sample output showing the three main categories of a git status call is included below:

```
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
#modified: hello.py
#
# 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: main.py
#
# Untracked files:
# (use "git add <file>..." to include in what will be committed)
#
#hello.pyc
```

The first status output will show the file as unstaged. The git add action will be reflected in the second git status, and the final status output will tell you that there is nothing to commit.

git status (3/3)

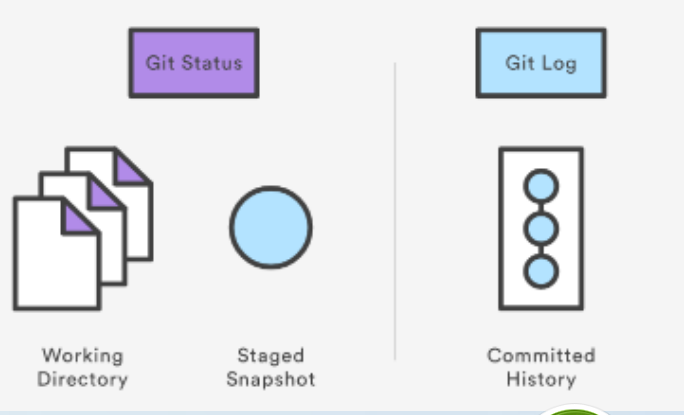
It‘s good practice to check the state of your repository before committing changes so that you don’t accidentally commit something you don't mean to. This example displays the repository status before and after staging and committing a snapshot:

```
# Edit hello.py

git status

# hello.py is listed under "Changes not staged for commit"

git add hello.py

git status

# hello.py is listed under "Changes to be committed"

git commit

git status

# nothing to commit (working directory clean)
```

git log (1/6)

The git log command displays committed snapshots. It lets you list the project history, filter it, and search for specific changes. While git status lets you inspect the working directory and the staging area, git log only operates on the committed history.

Log output can be customized in several ways, from simply filtering commits to displaying them in a completely user-defined format.



git log (2/6)

git log

Display the entire commit history using the default formatting. If the output takes up more than one screen, you can use Space to scroll and q to exit.

git log -n <limit>

Limit the number of commits by <limit>. For example, git log -n 3 will display only 3 commits.

git log --oneline

Condense each commit to a single line. This is useful for getting a high-level overview of the project history.

git log (3/6)

git log --stat

Along with the ordinary git log information, include which files were altered and the relative number of lines that were added or deleted from each of them.

git log -p

Display the patch representing each commit. This shows the full diff of each commit, which is the most detailed view you can have of your project history.

git log --author="<pattern>"

Search for commits by a particular author. The <pattern> argument can be a plain string or a regular expression.

git log (4/6)

git log --grep="<pattern>"

Search for commits with a commit message that matches <pattern>, which can be a plain string or a regular expression.

git log <since>..<until>

Show only commits that occur between <since> and <until>. Both arguments can be either a commit ID, a branch name, HEAD, or any other kind of revision reference.

git log <file>

Only display commits that include the specified file. This is an easy way to see the history of a particular file.

git log (5/6)

git log --graph --decorate --oneline

A few useful options to consider. The —graph flag that will draw a text based graph of the commits on the left hand side of the commit messages. —decorate adds the names of branches or tags of the commits that are shown. —oneline shows the commit information on a single line making it easier to browse through commits at-a-glance.

git log - Sample (6/6)

several options can be combined into a single command:

```
git log --author="John Smith" -p hello.py
```

This will display a full diff of all the changes John Smith has made to the file hello.py.

Viewing old commits

```
git
checkout
```

git checkout (1/5)

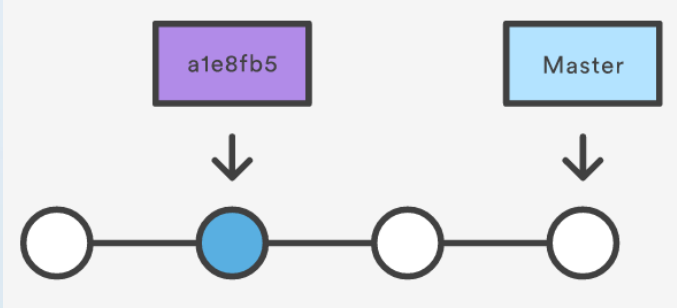
The git checkout command serves three distinct functions: checking out files, checking out commits, and checking out branches. In this slide, we're only concerned with the first two configurations.

Checking out a commit makes the entire working directory match that commit. This can be used to view an old state of your project without altering your current state in any way. Checking out a file lets you see an old version of that particular file, leaving the rest of your working directory untouched.

git checkout (2/5)

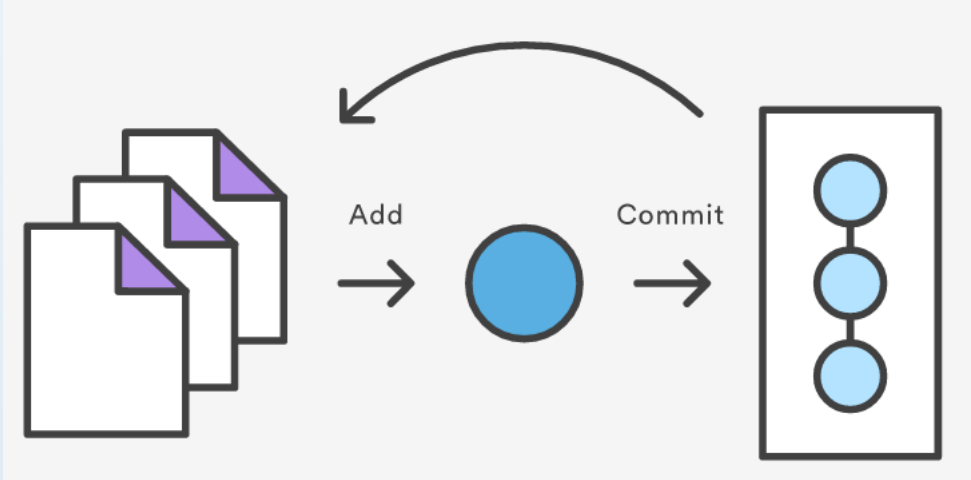
The whole idea behind any version control system is to store "safe" copies of a project so that you never have to worry about irreparably breaking your code base. Once you've built up a project history, git checkout is an easy way to "load" any of these saved snapshots onto your development machine.

Checking out an old commit is a read-only operation. It's impossible to harm your repository while viewing an old revision. The "current" state of your project remains untouched in the master branch



git checkout (3/5)

On the other hand, checking out an old file does affect the current state of your repository. You can re-commit the old version in a new snapshot as you would any other file. So, in effect, this usage of git checkout serves as a way to revert back to an old version of an individual file.



git checkout (4/5)

git checkout master

Return to the master branch. Branches are covered in depth in the next module, but for now, you can just think of this as a way to get back to the "current" state of the project.

git checkout <commit> <file>

Check out a previous version of a file. This turns the <file> that resides in the working directory into an exact copy of the one from <commit> and adds it to the staging area.

git checkout <commit>

Update all files in the working directory to match the specified commit. You can use either a commit hash or a tag as the <commit> argument. This will put you in a detached HEAD state.

git checkout – In practice (5/5)

Let's say your project history looks something like the following:

b7119f2 Continue doing crazy things
872fa7e Try something crazy
a1e8fb5 Make some important changes to hello.py
435b61d Create hello.py
9773e52 Initial import

You can use git checkout to view the "Make some import changes to hello.py" commit as follows:

git checkout a1e8fb5

Undoing changes

git checkout

git revert

git reset

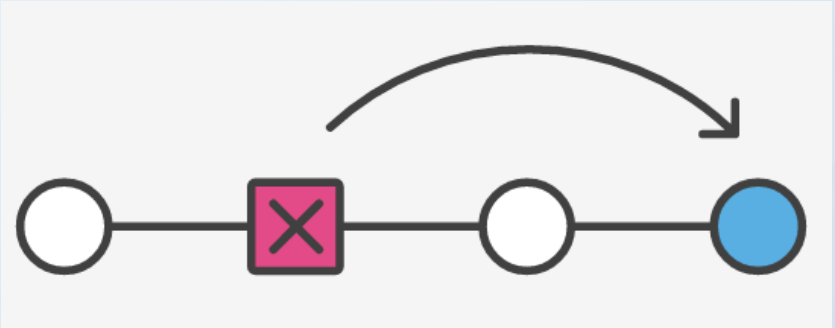
git clean

git checkout

Same material we went through in the previous section regarding git checkout.

git revert (1/2)

The git revert command undoes a committed snapshot. But, instead of removing the commit from the project history, it figures out how to undo the changes introduced by the commit and appends a new commit with the resulting content. This prevents Git from losing history, which is important for the integrity of your revision history and for reliable collaboration.



git revert (2/2)

git revert <commit>

Generate a new commit that undoes all of the changes introduced in <commit>, then apply it to the current branch.

git reset (1/4)

If git revert is a “safe” way to undo changes, you can think of git reset as the dangerous method. When you undo with git reset (and the commits are no longer referenced by any ref or the reflog), there is no way to retrieve the original copy—it is a permanent undo. Care must be taken when using this tool, as it’s one of the only Git commands that has the potential to lose your work.

Like git checkout, git reset is a versatile command with many configurations. It can be used to remove committed snapshots, although it’s more often used to undo changes in the staging area and the working directory. In either case, it should only be used to undo local changes—you should never reset snapshots that have been shared with other developers.

git reset (2/4)

git reset <file>

Remove the specified file from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes.

git reset

Reset the staging area to match the most recent commit, but leave the working directory unchanged. This unstages all files without overwriting any changes, giving you the opportunity to re-build the staged snapshot from scratch.

git reset (3/4)

git reset --hard

Reset the staging area and the working directory to match the most recent commit. In addition to unstaging changes, the --hard flag tells Git to overwrite all changes in the working directory, too. Put another way: this obliterates all uncommitted changes, so make sure you really want to throw away your local developments before using it.

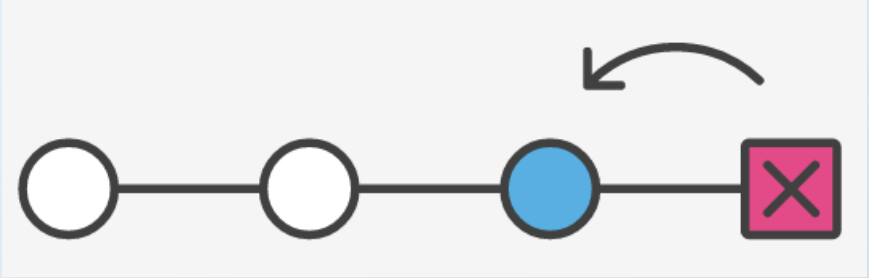
git reset <commit>

Move the current branch tip backward to <commit>, reset the staging area to match, but leave the working directory alone. All changes made since <commit> will reside in the working directory, which lets you re-commit the project history using cleaner, more atomic snapshots.

git reset (4/4)

git reset --hard <commit>

Move the current branch tip backward to <commit> and reset both the staging area and the working directory to match. This obliterates not only the uncommitted changes, but all commits after <commit>, as well.

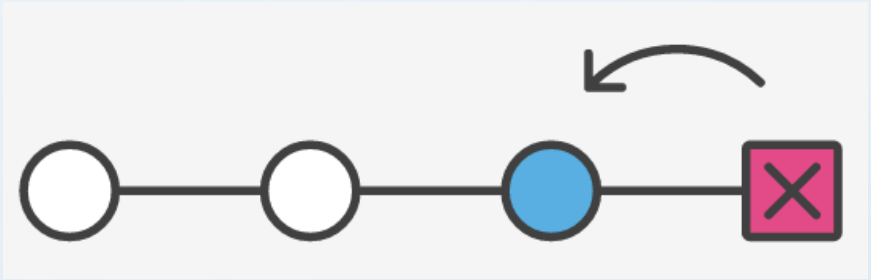


Reverting vs. Resetting (1/2)

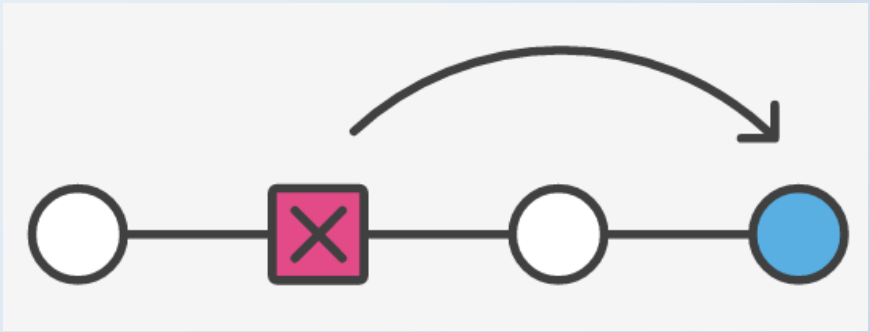
Reverting has two important advantages over resetting. First, it doesn't change the project history, which makes it a "safe" operation for commits that have already been published to a shared repository.

Second, git revert is able to target an individual commit at an arbitrary point in the history, whereas git reset can only work backwards from the current commit. For example, if you wanted to undo an old commit with git reset, you would have to remove all of the commits that occurred after the target commit, remove it, then re-commit all of the subsequent commits.

Reverting vs. Resetting (2/2)



Resetting



Reverting

git clean (1/3)

The git clean command removes untracked files from your working directory. This is really more of a convenience command, since it's trivial to see which files are untracked with git status and remove them manually. Like an ordinary rm command, git clean is not undoable, so make sure you really want to delete the untracked files before you run it.

The git clean command is often executed in conjunction with git reset --hard. Remember that resetting only affects tracked files, so a separate command is required for cleaning up untracked ones. Combined, these two commands let you return the working directory to the exact state of a particular commit.

git clean (2/3)

git clean -n

Perform a “dry run” of git clean. This will show you which files are going to be removed without actually doing it.

git clean -f

Remove untracked files from the current directory. The -f (force) flag is required unless the clean.requireForce configuration option is set to false (it's true by default). This will not remove untracked folders or files specified by .gitignore.

git clean (3/3)

git clean -f <path>

Remove untracked files, but limit the operation to the specified path.

git clean -df

Remove untracked files and untracked directories from the current directory.

git clean -xf

Remove untracked files from the current directory as well as any files that Git usually ignores.

Undoing changes

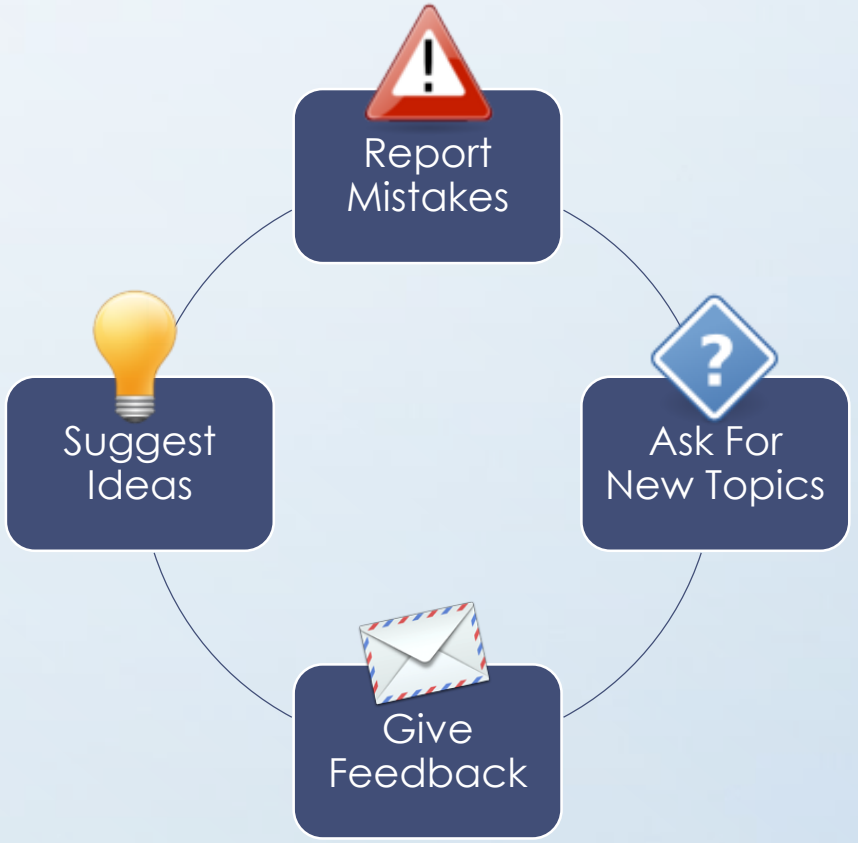
Command	Scope	Common use cases
git reset	Commit-level	Discard commits in a private branch or throw away uncommited changes
git reset	File-level	Unstage a file
git checkout	Commit-level	Switch between branches or inspect old snapshots
git checkout	File-level	Discard changes in the working directory
git revert	Commit-level	Undo commits in a public branch
git revert	File-level	Not Available

Further study

1. Difference Between a git reset and git revert (S63)
   - http://www.hostingadvice.com/how-to/git-undo-commit/

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! ☺

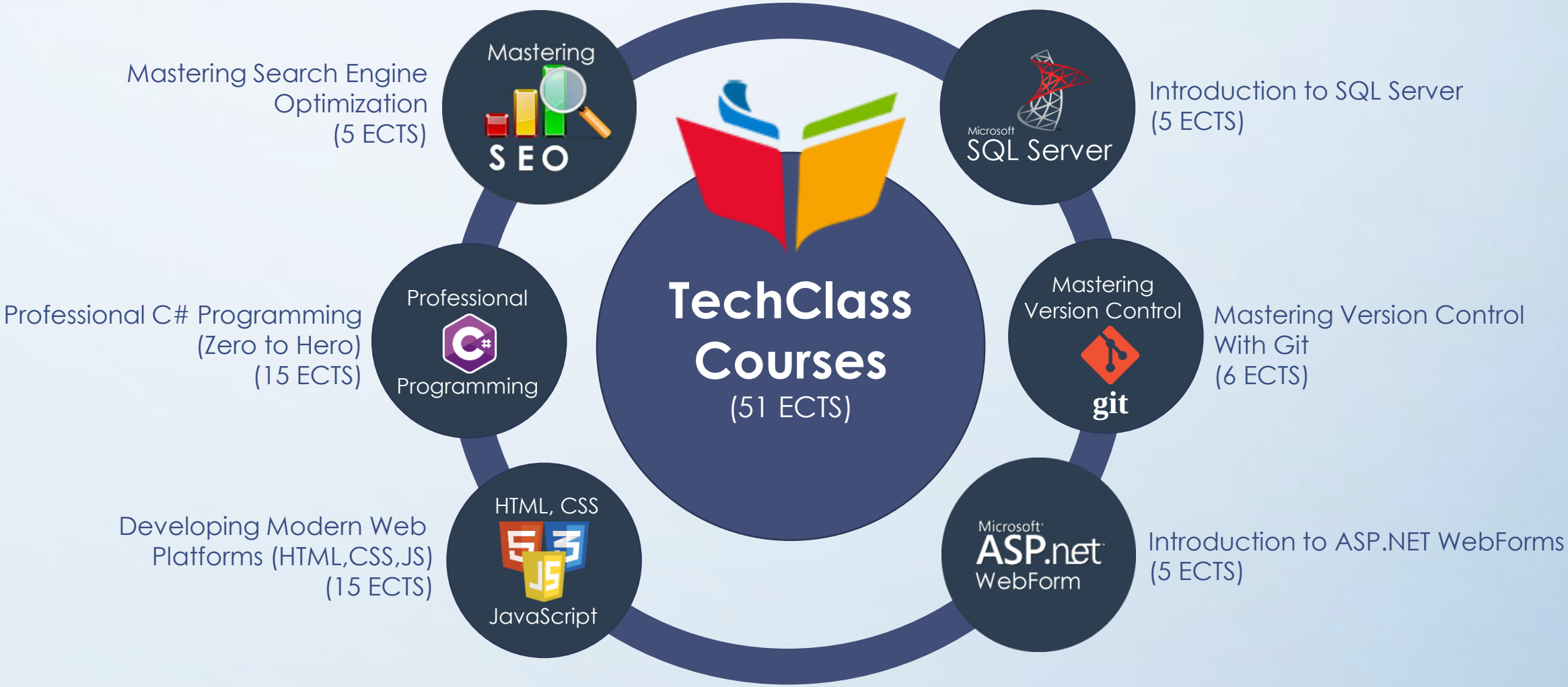
fb.com/techclass

@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