**How Git is Different**:

Distributed vs. Centralized

* There are generally two models in version control systems: centralized and distributed.
* Centralized VCS (Subversion) typically require a network connection to a centralized server.
  + You make a change to your project and then commit that change, which is sent to the centralized server to track.
  + Other developers can then immediately access your changes.
* Distributed version control systems (Git) break the process of committing code and sharing it with others into two parts.
  + You can commit your code to your local private repository without having to talk to a centralized server, removing the need to be connected to a network to make a change.

Private vs. Public Repositories

* Each developer who is sharing code with other developers has at least two repositories: a private and a public repository.
* The private repository is the one that exists on your computer and is the one you make commits to.
* Public repositories are the repository that you use to share your changes with other developers.
  + Multiple developers might have access to push changes to the – same public repository or each developer may have their own public repositories.
  + You can push and fetch from – multiple repositories. This allows you to pull in changes from another developer who’s working on the same project.

Commit IDs Instead of Revision Numbers

* Centralized VCS have the benefit of having one system that deals with revision numbers.
  + As everyone is committing and sharing their code in one repository, that repository can control what numbers it assigns to a particular commit.
  + That model doesn’t work in a decentralized system.
* Git uses commit IDs that are SHA-1 hashes instead.
* The hash is based on
  + The code, what came before it, who made the commit, when they made it, and a few other pieces of metadata.
* The chances are incredibly small of there being two different commits with the same commit ID.

Forking Is Good

* A fork is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.
* Most commonly, forks are used to either propose changes to someone else's project or to use someone else's project as a starting point for your own idea.

Propose changes to someone else's project

* A notable example of using forks to propose changes is for bug fixes. Rather than logging an issue for a bug you've found, you can:
  + Fork the repository.
  + Make the fix.
  + Submit a pull request to the project owner.
  + If the project owner likes your work, they might pull your fix into the original repository!

The Git Workflow

* The high-level overview of Git’s general workflow is shown in Figure



My Standard Workflow

* My standard day working with Git goes something like this:

1. I fetch all the changes from the other developers on my team – to make sure I’m working with the latest code, and then I start working on the user stories I have for the day.
2. As I make changes, I create a handful of commits—a separate commit for each change that I make.
3. Occasionally, I would need several separate changes that all need to be committed. I’ll break out – Git’s patch mode, stage, and finally commit each set of changes separately.
4. Once I have the feature complete, I give a quick review of the commits I’ve created – to make sure all the changes are necessary. At this point, I look for commits that can be combined and make sure they are in the most logical order.
5. Finally, once I have those commits ready, I share those commits by pushing them back upstream to my public repository (push is the term for sending commits to another repository), so the rest of the team can view them and integrate them with their repositories.

Small Teams with a Shared Repository

* Many small teams use Git like a traditional version control system. They have one main repository that all the developers can send changes to, and each developer has their own private repository to track their changes in.
* You make your changes locally; then when you’re ready to share them with other developers, you push them back to the repository you all share.
* If someone else has shared their changes since the last time you updated from the shared repository, you will get an error.
  + You must first get the changes from the shared repository and integrate them into your repository through a process called merging.
  + Once the changes are merged, you can push your changes to share with the rest of the team.

Repository Layouts

* The distributed nature of Git gives you a lot of flexibility in how you manage your repositories.
* Every person on your team has their own private repository
  + The repository that only that person can update.
* However, there are two distinct ways to handle public repositories:
  + Method 1 – Fully distributed model.
  + Method 2 – Shared repository model.
* Fully Distributed Model – Here, each developer has their own public repository that the developer uses to publish their changes to.
  + All the other developers on the team then pull changes from everyone else’s repositories to keep it up-to-date.
  + In practice, most teams have a lead developer who is responsible for making sure all the changes are integrated.
* Shared repository model – Here, all developers can push to a shared repository.
  + It resembles the Standard Centralized Model.
  + It requires the least amount of mental overhead when it comes to thinking about where a change is shared.



|  |
| --- |
| **Getting Started with Git** |

**Task 1 – Installing Git**

* You can install Git in several ways:
  + By using one of the GUI installers
  + By using a package management system
  + By using Linux tradition that spawned Git – by compiling it from source.
* All major operating systems provide alternatives to compiling Git yourself. For example, Ubuntu provides Git via its apt-get tool.

Install Git on Ubuntu

* + prompt> sudo apt-get install git-core

To install the user manual, do this:

* + prompt> sudo apt-get install git-doc
* OS X users have the option of using MacPorts4 or the new Homebrew5 to handle the installation.
* The following tools on Ubuntu are helpful for tracking down all of the dependencies.
  + prompt> sudo apt-get build-dep git-core git-doc git-svn

**Task 2 – Configuring Git**

* Git requires some configuration to work.
* You must tell Git your name and your email address since there is no central repository to keep track of that information.
* Git uses both to calculate the commit ID that identifies each commit.
* The following commands use --global to specify that – they are configuration values for every repository you interact with on this machine.

Configure Git to know who you are.

* + prompt> git config --global user.name "Your Name"
  + prompt> git config --global user.email "user@domain.com"
  + prompt>
* The configuration file is stored in ~/.gitconfig.
* You can edit the file directly in addition to using the git config command.

Set the Git user for a specific repository

* You can set every setting on a global or per-repository basis.
* By leaving --global out of the command, the settings will be stored in the repository’s .git/config file.
  + prompt> cd /path/to/repository
  + prompt> git config user.name "Your Name"
  + prompt> git config user.email "user@domain.com"
  + prompt>

Turn colors on wherever possible in the Git UI

* You can set color.ui to auto if you like to have your command-line interfaces colorized.
* The auto setting tells Git – to Use color whenever it is generating output to be displayed but – to render plain text whenever the output is being piped to another process.
  + prompt> git config --global color.ui auto
  + prompt>

Configure Git’s editor

* Git uses core.editor to specify a particular editor.
* Git launches an editor whenever you need to create a commit message, edit patches, and do a few other tasks.
  + prompt> git config --global core.editor /path/to/editor
  + prompt>
* Git doesn’t require you to set the core.editor value, though.
* It tries to figure out what editor to use by checking the following values, in order:
  + GIT\_EDITOR environment variable;
  + core.editor configuration value;
  + VISUAL environment variable;
  + EDITOR environment variable;
  + plain vi.
* The value is the command-line script to launch your editor.

**Task 3 – Creating a new Repository**

* Repositories in Git are stored on your ‘local file system’ right alongside the code they track.
* You create a repository by typing git init in the directory that you want to start tracking files in.
* You use two repositories in Git to collaborate with others: a private one and a public one.
* Your private repository is where you do all your work. It’s the repository with the ‘working tree’.
* This two-tier system gives you – the ability to track local experimental changes while – only sharing changes via your public repository that are ready for others to work with.
* git init creates a ‘.git’ directory in your current directory and initializes the Git repository inside that directory.
* You have to initialize the repository only once.

Create a repository (Structure)

* + prompt> mkdir some-repository
  + prompt> cd some-repository
  + prompt> git init

To create a repository called ‘widgets’ in the /work directory, use this:

* + prompt> mkdir /work/widgets
  + prompt> cd /work/widgets
  + prompt> git init
  + Initialized empty Git repository in /work/widgets/.git/
* Once you’ve initialized a repository, you still need – to add and commit the files – using git add (Staging Changes to Commit) and git commit (Committing Changes), respectively.
  + But both first require an initialized repository.

Create a repository in an existing directory and add all files from that directory.

* + prompt> cd /path/to/some/directory
  + prompt> git init
  + prompt> git add
  + prompt> git commit -m "some commit message"

For example, to create a repository inside an existing directory called /work/existing-widget, use this:

* + prompt> cd /work/existing-widget
  + prompt> git init
  + Initialized empty Git repository in /work/existing-widget/.git/
  + prompt> git add
  + prompt> git commit -m "initial commit"
  + [master (root-commit) 6e477fa] initial commit 101 files changed, 4083 insertions(+), 0 deletions(-) create mode 100644 AUTHORS ... and 100 more files ...
* Once you’ve initialized a repository, you have a working tree that you can interact with.
  + The working tree is your ‘view’ into what’s stored in your repository.
  + It typically represents the latest copy of what’s stored in your repository.

**Task 4 – Creating a Local Copy of an Existing Repository**

* You need to create a clone of a remote repository to start making changes to it.
* The git clone command initializes a new repository on your computer and fetches the entire history. (all the changes that have been tracked during the life of that repository)
* After it’s complete, you can start making changes to the files in your local working tree and tracking commits locally.

Clone a repository

* + prompt> git clone some-repository

... example ...

* + prompt> git clone git://github.com/tswicegood/bobby-tables.git
  + Cloning into bobby-tables...
  + remote: Counting objects: 39, done.
  + remote: Compressing objects: 100% (25/25), done.
  + remote: Total 39 (delta 16), reused 26 (delta 9)
  + Receiving objects: 100% (39/39), 39.23 KiB, done.
  + Resolving deltas: 100% (16/16), done.
* Sometimes you don’t need the entire history of the repository.
* You don’t always need the last ten years of changes—the last year’s might suffice.
* You can use the ‘--depth parameter’ – to limit how many revisions you fetch. This is called a shallow repository.

Create a shallow clone with the last fifty commits

* + prompt> git clone --depth 50 some-repository
* You use ‘git clone’ – to fetch changes when a repository already exists.
  + but you don’t have to clone a repository – to work with a remote repository.
* Remote repositories are repositories that you can talk to (over a network) push and pull changes from.
* You can initialize a new repository, create a New Repository, and then add a remote repository later with the ‘*git remote’* command.

|  |
| --- |
| **Working with Git** |

* It’s time to start learning how to interact with Git.
* The workflow in Git is different from other version control systems:
  + Each time you make a change you want to track, you need to commit it.
* The workflow goes like this.
  + Create your repository—either create a new repository or clone an existing one.
  + Make some changes.
  + Test that they do what you want
  + Commit those changes
  + Make some more changes, and so on.
  + You share those changes when they’re ready.
* One thing to keep in mind when working with a distributed version control system(DVCS) like Git is that committing a change and sharing that change are two different processes.
* This separation provides you with a lot of freedom. You can experiment locally, try a whole bunch of things, and then share the best solution.
* Lots of small, discrete changes that touch very specific portions of the code are better than a few monolithic changes.
* Making changes and testing are up to you and how you interact with the code in your project.
* Seeing what changes need to be committed is where we pick up.

**Task 5 - Seeing What Has Changed**

‘git status’ tool

* Your local repository will track the changes.
* The following things must be seen before you start committing everything:
  + We need to see – What changes exist between your working tree and your repository?
  + We need to see – What changes are staged and ready to commit?
* git status is the tool for seeing what has changed.
* Staging is a step before the commit process in git.
  + A commit in git is performed in two steps: staging and actual commit.

What the status of a new repository looks like.

* If you just created a repository using git init, this is what your repository looks like:
  + prompt> git status
  + # On branch master
  + #
  + # Initial commit
  + #
  + nothing to commit (create/copy files and use "git add" to track)
* ‘git status’ has several different outputs, depending on what’s in your working tree.
  + The three types of outputs are – staged changes, changes to known files, and untracked files.
  + Untracked files – Git outputs the files and paths that it doesn’t know anything about—the files that you haven’t told Git about yet.
  + Changes to known files – files that Git knows about but that have changed.
  + Staged changes – what all files you would commit if you ran git commit right now.

What git status looks like in a repository with changes.

* In order to see the various outputs of git status, it requires a repository with some changes in its working tree.
* The following is the output of git status on my local Castanaut repository:

Line 1 prompt> git status

- # On branch master

- # Changes to be committed:

- # (use "git reset HEAD <file>..." to unstage)

5 #

- # modified: castanaut.gemspec

- #

- # Changed but not updated:

- # (use "git add <file>..." to update what will be committed)

10 # (use "git checkout -- <file>..." to discard changes in ...

- #

- # modified: README.txt

- #

- # Untracked files:

15 # (use "git add <file>..." to include in what will be ...

- #

- # pkg/

What git status looks like with no changes.

* + prompt> git status
  + # On branch master
  + nothing to commit (working directory clean)

**Task 6 – Staging changes to commit**

* Git uses a two-step process to get changes into the repository.
  + The first step is staging changes through ‘git add’.
  + Staging a change adds it to the index, or staging area.
  + This sits between the working tree (your view of the repository) and the actual repository.
* Through the staging area, ***you can control what is staged*** from the most coarse-grained (adding everything within the repository) down to editing the changes, line by line.

Stage an entire file to commit.

* you can select individual files or paths to add by calling git add and passing the filename or path as the parameter. Git adds everything under a path if you provide that.
  + prompt> git add path/to/file
  + ... or ...
  + prompt> git add path/
  + ... or everything under the current directory ...
  + prompt> git add
  + prompt>

Add all files in the current repository.

* Another quick way to add all files is the -A parameter. This adds all the files inside the repository that are not explicitly ignored
  + prompt> git add -A|--all
  + prompt>

Add all tracked files that have been changed.

* You can add files that have changed using the -u parameter.
* It doesn’t add any new files, though, only files that have already been tracked and have modifications in them.
  + prompt> git add -u|--update
  + prompt>

Choose which changes to commit.

* You can control which parts of a file you commit using the -p parameter.
* Running this, you’re presented with each section of the file that has changed, and you’re given the opportunity to add or skip it.
* You can stage the change by pressing y or skip a change with n.
* s lets you break the change into smaller pieces.
  + prompt> git add -p|--patch
  + ... or a specific file ...
  + prompt> git add -p path/to/file
  + prompt>

Open the current diff in the editor

* you can directly edit the changes that are being staged by using the -e parameter. This opens the diff in your configured editor.
  + prompt> git add -e
  + ... or a specific file ...
  + prompt> git add -e path/to/file
  + prompt>

**Task 7 – Committing Changes**

* Git tracks changes to your repository through commits, which you make with the git commit command.

Stage and commit changes

* Prior to most commits, you need to stage the files you want to commit using the git add.
* Each commit requires a commit message. You can use -m and a string in quotation marks as your message or use Git’s editor to write a message.
  + prompt> git add <some file>
  + prompt> git commit -m "Some message"
  + [master a276f08] Some message
  + 1 files changed, 2 insertions(+), 0 deletions(-)

Commit all modified files

* By avoiding git add you can commit every change in your working tree with the -a parameter.
* It commits everything you have staged and all the changes to your working tree.
  + prompt> git commit -m "Some message" -a
  + [master 5d251db] Some message
  + 1 files changed, 1 insertions(+), 0 deletions(-)

**Task 8 – Ignoring Files**

* Software projects generate a lot of unwanted code. Some of it you don’t need to commit.
* We don’t need or want these files cluttering up our repository or showing up in git status. That’s where the .gitignore and friends comes in.
* Each line of the .gitignore is scanned, and any matches it finds are ignored by Git.
* Your .gitignore file is inside your repository, so you can track it like any other file.
  + You can put it at the top level of your repository, and the rules cascade through all sub-directories.
  + You can also use subdirectory-specific .gitignore, and those rules will only apply to files and directories inside that subdirectory.

Ignore a specific file and/or path called cache.

* + Add the following to .gitignore:
  + Cache

Ignore all .swp files from Vim

* + Add the following to .gitignore:
  + \*.swp

Set up a global excludes file.

* Your excludesfile can exist anywhere you want on your computer. The following example puts it in your home directory in the .gitignore file:
  + prompt> git config --global core.excludesfile \
  + ~/.gitignore

**Task 9 – Undoing Uncommitted Changes**

* Git’s two-step process for tracking a commit means – you can have files that are staged for commit that you’re not ready to commit.
* You use git reset HEAD or git rm--cached depending on the circumstance.

Unstage a modified file that’s been staged.

* Scenario 1: You staged a change to file and want to unstage it.
* use git reset HEAD.
* This is the most common use.
* You’re telling Git, “Change the index (the staging area) to the latest version of this file.”
* Scenario 2: You have a new file that’s been staged that you don’t want to commit now.
* Use git rm--cached.
* Normally, git rm is used to remove files from your repository but adding the --cached option tells Git to leave your working tree alone.
* For example, to undo changes to cache.py, use this:
  + prompt> git reset HEAD -- cache.py
  + Unstaged changes after reset:
  + M cache.py
  + It tells Git that all arguments are done and that the rest are files or paths.

Undo all uncommitted changes to a file.

* git checkout is used for making changes that you want to undo completely.
* We have to be careful as git checkout removes all untracked changes from a file or directory and cannot be undone.
* You can’t get those changes back if they were never tracked by Git.
  + prompt> git checkout -- cache.py

**Task 10 - Moving Files in Git**

* Performing tasks such as – reorganizing files, changing file formats, and so on, requires that files and sometimes entire directories get moved.
  + git mv handles this for you.

Move a file or directory

* You provide it with two options: the name of the original file and the new name.
* This works on files, directories, or symlinks—anything Git can track.
* For example, to move README.md to README.rst, use this:
  + prompt> git mv README.md README.rst
  + prompt> git commit -m "Changed README from Markdown to ReSTructured text"
  + [master f810d86] Changed README from Markdown to ReSTructured text
  + 1 files changed, 0 insertions(+), 0 deletions(-)
  + rename README.md => README.rst (100%)

Move a file or directory into another directory.

* You can move files, directories, and symlinks into other directories as well.
* Provide git mv a directory as the second option, and you’re set.
* Git stages the change for you after you call git mv.
* You must call git commit after git mv to make the move permanent.
  + prompt> git mv README.rst docs/
  + prompt> git commit -m "Moved README into docs/ directory"
  + [master 99a0de8] Moved README into docs/ directory
  + 1 files changed, 0 insertions(+), 0 deletions(-)
  + rename README.rst => docs/README.rst (100%)
* git mv won’t overwrite an existing file; it displays an error instead.
* You can override this behaviour by providing --force (or -f).
* Be careful because this makes Git overwrite the existing file.
* That’s dangerous if the existing file you’re overwriting isn’t tracked by Git.
* You have no way of getting that file back.

**Task 11 – Deleting Files in Git**

* Files and directories sometimes outlive their usefulness.
* You can remove them from your working tree and tell Git to quit tracking them using the git rm command.

Delete a file from Git.

* To delete a file called outdated.py, use this:
  + prompt> git rm -- outdated.py
  + rm 'outdated.py'
  + prompt> git commit -m "remove outdated.py"
  + [master 42010bf] remove outdated.py
  + 1 files changed, 0 insertions(+), 17 deletions(-)
  + delete mode 100644 outdated.py
* This doesn’t remove the file from your repository’s history; it removes it only from your working tree going forward.
* You can always go back in the history of the repository and see the files or directories that have been removed.
* You call git rm and provide it with a filename to tell Git to remove it.

Delete a directory from Git.

* You must provide the -r option if you are deleting a directory and all the files under it.
* It tells Git to recursively delete all the files starting at the provided directory.
* Like most other actions in Git, git rm requires git commit to finalize its action.
  + git rm stages the removal, and git commit finalizes it.
* To delete a directory called old/, use this:
  + prompt> git rm -r -- old/
  + rm 'old/outdated.py'
  + prompt> git commit -m "remove the old/ directory"
  + [master ddbd005] remove the old/ directory
  + 1 files changed, 0 insertions(+), 17 deletions(-)
  + delete mode 100644 old/outdated.py

Get a directory back after deleting it but before committing it

* You can undo a git rm before you make a commit through a two-step process.
  1. You have to reset the index using git reset HEAD.
  2. Check out the file from the repository to restore it using git checkout -- path/to/file.
* This example uses the previous example where old/ is deleted using git rm, but before the staged deletes are committed.
* There are two steps. First, reset the index:
  + prompt> git reset HEAD -- old/
  + Unstaged changes after reset:
  + M old/outdated.py
* Second, check out the files from the repository:
  + prompt> git checkout -- old/

Force a file to be removed

* git rm attempts to keep you from accidentally deleting a file that has changes that have not been committed. You can override this behaviour with -f.
* Forcing Git to delete the file – will remove the file and all traces of the changes that haven’t been committed yet.
  + prompt> git rm -f -- outdated.py
  + rm 'outdated.py'

**Task 12 – Sharing Changes**

* Git is different from most traditional version control systems; committing a change and sharing that change are two distinct tasks.
* Committing changes is covered in detail in Task 7.
  + The task gives you a quick cheat sheet for the various tasks you need to perform to collaborate with others.

1. Set up the remote repository

* + You clone a repository. Or...
  + You add a remote to an existing repository.

2a. Fetch changes from a remote repository

* Once you have a local clone, or have set up a remote after initializing your repository (Task 19), you need to fetch changes from the remote repository – to keep your local branches (Part III) in sync using git fetch.
  + prompt> git fetch <remote name>
  + ... then merge the changes into your work ...
* After fetching changes, you must merge those changes using any of the methods covered in Part III.

2b. Pull changes from a remote repository

* You can also fetch changes and merge them at the same time using git pull.
  + It fetches the changes and then merges them into the current branch.
* You can specify the --rebase parameter to have Git rebase your local branch (Task 16) on top of the remote changes.
  + prompt> git pull <remote name>
  + ... pull from a repository you cloned ...
  + prompt> git pull origin
  + ... pull, but rebase your local changes on top
  + ... of the remote change instead of merging them
  + prompt> git pull --rebase origin <remote branch name>

3. Push changes to a remote repository

* Sending changes back to a remote repository to share is done via the git push command.
* It is the inverse of git pull;
* It sends your changes to the remote repository and merges those changes into the remote branch via a fast-forward merge.
* Fast-forward merge is a merge where both branches share a common ancestor and only the branch being merged in has changes in it.
  + prompt> git push <remote name> <branch name>

|  |
| --- |
| **Organizing Your Repository with Branches and Tags** |

* Branches allow you to segregate different lines of development.
* One of the most compelling features of Git is its ability to easily handle creating branches and gracefully merge them back together.
* Branches track changes to multiple versions of a project.
  + For example, you might be finishing up version 1.0 and already starting on new features for version 1.1.
  + Using branches, you can keep the code from version 1.0 isolated so new features from 1.1 don’t accidentally slip into the version that is getting ready to release.
* You give branches names, making it easier track them based on their name, rather than some commit ID.
  + Master is the name of the default branch that Git uses.
  + You can create as many branches as you want
* Branches in Git are relatively simplistic—they’re simply a text file inside the repository that marks the latest commit in the branch.
* Treating branches as pointers makes operations with branches painless and fast.

Several different ways to approach using branches

* One approach is the topic branch. You can use this style of branch to work on a specific feature, fix a bug, or deal with any other “topic”.
  + Once it’s complete, then you merge the finished changes back into your master branch.
* Another common type of branch is the release branch. You create release branches as you approach a release in your project.
  + They’re useful when you or other members on your team are working on multiple versions of your project.
  + You can create a branch called release\_v1.0 for the 1.0 version – to isolate that release from features that aren’t supposed to ship until version 1.1.
  + Work on version 1.1 continues like normal in the master branch.
  + Any changes from the final work on version 1.0 get merged back into master.

Use of Tags

* Tags are used to mark milestones in your project, such as releases.
* Tags are similar to branches, except they are read-only.
* Once you create a tag, you can’t change it.
* Branches and tags are not shared by default.
* Like commits, we can decide which branches and tags to share with other developers and when.
* Branches in Git are pointers to a specific commit in your repository’s history.
* Since each commit knows about its parent (or parents), Git can reconstruct what’s in a branch – by looking at the latest commit in that branch and walking the history backward to find all the ancestors.
  + This simplified approach to branches makes them quick to create, rename, merge, and even delete.
* You occasionally have to merge changes between branches to keep from duplicating the same work in different branches.
* Git (makes this easy by tracking which commits have been merged between branches) for you.
  + Many modern VCS do not do this at all or do it poorly

Different merge strategies (Fast-forward merge)

* Git can employ several different merge strategies to merge branches together.
* The first is the fast-forward merge.
* Fast-forwards are performed when two conditions are met:

1. When the branch you are merging in is being merged back into the branch it was created from.
2. When the original branch hasn’t had any new commits since the new branch was created. For an example, check out Figure.



* Fast-forward merges do not actually create anything new in the repository. They “fast-forward” the branch pointer to the new location.

Different merge strategies (Recursive merge)

* Recursive merges are used by default when both the branches have commits that are not in the other branch.
* Git creates a merge commit that has two parent commits — the latest commit in each branch.



Rebasing

* Another way to get information from one branch into another is through rebasing the branch.
* Rebasing is a powerful tool in Git that’s often misunderstood, which is understandable. It’s best explained with an example workflow.
* Consider the following example.
* You start your morning by pulling in all the changes from the company’s shared repository and then start working. During the morning, your co-worker pushes some commits upstream. When you try to push the commits you’ve made, you get an error. You now have two options. You can fetch the changes and do the following:

1. Merge them into your local branch, creating a merge commit.

2. Rebase your local branch on top of the remote branch.

* Rebasing takes the commits you made this morning and then replays them, one by one, starting on the other branch. You can do this to keep the appearance of a continuous stream of development instead of having a bunch of merge commits scattered throughout your repository’s history.

**Task 13 – Creating and Switching Branches**

* Git’s branches enable you to separate experimentation from production-ready code.
* Git’s convention is to treat the master branch as its main line of code.
* You can rename it to anything you want, but it’s a good idea to keep with the convention.

Create a new branch from current place in the repository

* You can create a new branch using the git branch command and providing it with at least – one additional parameter (the name of the branch you want to create).
* This uses your current location in the repository as the place from where to create the branch.
  + prompt> git branch <new branch name>
  + ... example ...
  + prompt> git branch new
  + prompt>

Create a new branch from another branch, tag, or commit

* You can also create branches starting at points in the history of the repository.
* Provide git branch with the name of the new branch you want to create followed by the commit ID or branch or tag name – to create a branch at that point.
  + prompt> git branch <new branch name> <starting point>
  + ... example ...
  + prompt> git branch newer 99a0de8
  + prompt>

Check out a different branch, tag, and so on

* git branch just creates the branch; you have to switch to it.
* You can use the git checkout command to check out the new branch.
  + prompt> git checkout <branch>
  + ... example ...
  + prompt> git checkout newer
  + Switched to branch 'newer'

Create a branch and check it out at the same time

* Creating a new branch and checking it out immediately is common in Git.
  + You can do both actions with one command: git checkout -b.
* Like git branch, git checkout requires at least one parameter (the branch name) and takes an optional second parameter (specifying the point to create it from).
  + prompt> git checkout -b <new branch> [<starting point>]
  + ... example ...
  + prompt> git checkout -b newest 64648c9
  + Switched to branch 'newest'

Create a branch with or without tracking

* Tracking branches store additional metadata information about the relationship between two branches.
* The most common tracking branch is a local branch that tracks a remote branch.
* This additional information is used by other commands (such as git push and git status) to provide additional functionality.
* Using a remote branch as your <starting point> implies that --track is on.
* Use --no-track to turn it off.
  + prompt> git branch --track <new branch> [<starting point>]
  + prompt> git branch --no-track <new branch> [<starting point>]
  + prompt>

**Task 14 – Viewing Branches**

View all local branches

* You need to be able to see what branches your repository has in it, so you can switch between them.
* You can use git branch to get that information.
* You can view local, remote, or all branches depending on which parameters you pass to git branch.
* Calling git branch by itself shows your local branches.
  + prompt> git branch
  + master
  + new
  + newer
  + \* newest

View all remote branches or all branches

* You can add either the -r parameter or the -a parameter to view only the remote branches or all the branches, respectively.
* Your current branch always has an asterisk before it in the output from git branch.
  + prompt> git branch -r
  + origin/master
  + prompt> git branch -a
  + master
  + new
  + newer
  + \* newest
  + remotes/origin/master

View all that are or are not merged into the current branch

* It’s useful to see what branches have or have not been merged into the current branch.
* You can see that by using the --merged and --no-merged parameters.

View all branches that contain a particular commit

* It’s useful to be able to find out which branches contain a particular commit.
* For example, you can track which branches contain a commit that has a known bug in it by using the --contains parameter.
  + prompt> git branch --contains <commit id>

**Task 15 – Merging Commits Between Branches**

* You must merge changes from another branch into your current branch in order to be able to use them. The simplest way to do this is through git merge.
* git merge takes two options:
  + The name of the other branch you want to merge and
  + The optional local branch you want to merge into.
* You can leave off the current branch when you’re merging changes into your current branch.
  + prompt> git checkout master
  + Switched to branch 'master'
  + prompt> git merge development
  + Updating af0fe21..290b0d2
  + Fast-forward
  + old/README.rst | 8 ++------
  + 1 files changed, 2 insertions(+), 6 deletions(-)

Merge changes, but don’t commit

* By default, git merge commits the merged changes if they can be successfully merged.
* You can short-circuit this with the --no-commit option.
* This is useful when you want to review, and possibly edit, the changes from the merge before making a commit.
  + prompt> git merge --no-commit development
  + Automatic merge went well; stopped before committing...

Force the creation of a merge commit

* Fast-forward merges are often useful to merge commits, but sometimes you want to log that a merge happened.
* This is common in projects when a big feature that was developed in a separate branch is merged in; it provides a single commit you can revert if it needs to be removed in the future.
* You can do this with the --no-ff option.
* It forces Git to create a merge commit, showing that the two branches were merged.
  + prompt> git merge --no-ff development
  + Merge made by recursive.
  + old/README.rst | 8 ++------
  + 1 files changed, 2 insertions(+), 6 deletions(-)

Add a one-line log message from each merged commit to the merge message

* When merge commits are created you can add the --log option.
* Traditional merge log messages contain – Merge branch ‘development’.
  + prompt> git merge --log development

Specify a custom log message for a merge commit, if created

* There are two ways you change this.
  + You can add the --log, which adds (the subject line from each commit) to the merge commit message.
    - prompt> git merge --log --no-ff development
  + You can use -m and a message, which lets you specify the entire message just like git commit.
    - prompt> git merge -m "my message" development

**Task 16 – Rewriting History by Rebasing**

* Using git rebase, you can rewrite the history of a repository in a variety of ways.
* It is one of the most powerful commands in Git, which makes it one of the most dangerous.

Rebase your current branch against another

* Rebase takes a series of commits (normally a branch) and replays them on top of another commit (normally the last commit in another branch).
* The parent commit changes so all the commit IDs are recalculated.
  + This can cause problems for other developers who have your code because the IDs don’t match up.
* For example, rebase your current branch against master:
  + prompt> git rebase master
  + First, rewinding head to replay your work on top of it...
  + Applying: simple commit
* There’s a simple rule of thumb with git rebase: use it as much as you want on local commits.
* git rebase takes a branch ((the most common use), a tag, or a commit ID) to rebase on top of.
* For example, if af0fe21 is the commit ID for master, use this:
  + prompt> git rebase af0fe21
  + ... same as above ...
* You can also pass the --rebase option to git pull, causing it to perform a rebase instead of merging the upstream changes into your local branch.
* git rebase requires a clean working tree—that is, a working tree with no modified files.
* If you have changes that you’re not ready to commit, you can stash them (Temporarily Hiding Changes) until you’re done.
* A conflict might arise during the replaying of commits.
  + A conflict happens when two commits modify the same line of code.
  + git rebase stops when this happens and asks you to fix the conflict and then continue.
  + You tell Git you’re ready with git rebase --continue.
  + You can also skip a commit that’s causing a conflict by calling git rebase --skip.
  + However, that could lead to further conflicts.
  + You can abort the rebase too with git rebase --abort.

Undo a rebase after it completes

* There’s always a safety net if you need to undo a rebase after it’s completed.
* Git points ORIG\_HEAD at the commit before major changes like git rebase are run.
* You can use git reset to reset your repository back to that original state.
  + prompt> git reset --hard ORIG\_HEAD
  + HEAD is now at e9f9fdc update the README

**Task 17 - Deleting Branches**

* Branches can outlive their usefulness.
* Once you no longer need one, you can delete it from your repository.
* Remember, branches in Git are pointers to a commit.
* Deleting a branch doesn’t delete any commits, just deleting the named pointer that refers to that commit.
  + Since tags and branches both point to a single commit, you can tag your release and then delete the release branch.
* You can always create a new branch from the tag later if you need to make a change and then retag the new version, and the history will look like the branch had always been there.

Delete a branch that has been merged into the current branch

* You can delete a branch with git branch -d.
* You must provide the branch name you want to delete.
* To delete a branch called experiment, do this:
  + prompt> git branch -d experiment
  + Deleted branch experiment (was e9f9fdc).

Delete a branch that hasn’t been merged into the current branch

* Git warns you if the branch you are trying to delete has not been merged into the current branch.
* You can override this behaviour by using -D (capital D).
* This tells Git that you want to delete the branch even if it hasn’t been merged in.
  + prompt> git branch -D experiment
  + Deleted branch experiment (was e9f9fdc).

**Task 18 – Tagging Milestones**

* You need to make milestones in your projects, each slightly different, such as for one, its weekly iterations, and for another, its version numbers.
* You can use git tag to handle this.

List all tags

* git tag creates a read-only marker within the repository.
* You can treat tags, but you can’t check them out and start committing to them.
* However, you can create a new branch from a tag.
* Creating a tag requires one parameter: the name of the tag.
  + prompt> git tag
  + v0.1
  + v0.2
* Tag the latest commit as version 1.0 in the current branch.
  + prompt> git tag v1.0
  + prompt>
* One key difference between tags and branches is that there is no difference between a remote tag and a local tag. A tag is a tag.

|  |
| --- |
| **Working with a Team** |

* Using Git, you can collaborate with other developers on your team.
* You share code with other developers using remote repositories.
* Remote repositories are given names to make them easier to remember.
  + For example, the conventional name for your default remote repository is origin.
* There are two different ways to share code through a remote repository in Git: with a shared repository model or a distributed repository model.

A shared repository model

* A shared repository means that all members of your team can push and pull from the same repository.
* Team members keep their changes locally until they’re ready; then they push those changes back upstream when they’re complete for the rest of the team to use.
* There’s very little overhead in determining where the code is and what state everything is in.

A distributed repository model

* Each member of your team has their own private repositories on their computers plus a public repository that they push their code to.
* Each team member needs to pull changes from the other team members to make sure they have the latest code.
* Most teams using a distributed model designate someone as the release manager.
* It becomes their job to make sure that everything is merged together, and that member’s public repository becomes the repository of record.
* Here we introduce two new concepts you use to retrieve and send changes to remote repositories:
  + Fetching, or retrieving changes from a remote repository.
  + Pushing, or sending changes to a remote repository
* The only difference between a pull and a fetch is that pulling merge changes after Git fetches them.
  + It’s a shorthand way to combine fetching and merging into one command.
* All of these changes being retrieved (Fetching) from remote repositories are stored inside your local repository in remote branches.
* Git treats remote repositories (remotes) as branches.
* You can treat remote branches like normal branches with one exception: you can’t commit to them.
* The only way to get commits into a remote branch is to push those changes to the remote.

**Task 19 - Adding and Removing Remotes**

* Git allows you to have as many remote repositories as you like.
* It’s common to have a different remote for each member of your team in a fully distributed architecture for your repositories.

Add a new remote repository

* Git requires that each remote have a unique name.
* You must tell Git where to access remote repositories using the git remote add command.
  + prompt> git remote add <name> <repository URL>
  + ... example ...
  + prompt> git remote add tswicegood\
  + git://github.com/tswicegood/bobby-tables.git
  + propmt>
* It requires two parameters: a name and a repository URL.
* A name – the short name you use to reference the remote repository by.
  + The name origin is the conventional name for the repository that you clone from.
  + Git uses this convention in several commands that allow you to skip the remote name when you’re working with the origin repository.
* The repository URL – It points to the actual location of the remote repository.
  + This can be in another directory on your system or, more commonly, a repository that is accessible via a network connection.
  + Git can transfer over its own git protocol, over git using SSH to encrypt the data transfer and handle authentication, and over HTTP/HTTPS.

Make the master branch a tracking branch

* Whenever you create a local branch from a remote branch, Git sets up tracking branches for you.
* If you want to set up your local branch as a tracking branch of the remote, run the following commands after you push to your remote repository.
* As an example, here’s the workflow in a project:
  + prompt> git checkout origin/master
  + Note: checking out 'origin/master'.
  + You are in 'detached HEAD' state. ... and so on ...
  + git checkout -b new\_branch\_name
  + HEAD is now at d7c8880... ignore stuff from virtualenv
  + prompt> git branch -d master
  + Deleted branch master (was d7c8880).
  + prompt> git checkout -b master
  + Switched to a new branch 'master'

Remove a remote

* You can remove remote repositories by using the git remote rm command.
* It removes any tracking branch information in addition to removing the remote definition.
  + prompt> git remote rm <name>
  + ... example ...
  + prompt> git remote rm tswicegood

**Task 20 – Retrieving (Fetching) Remote Changes**

* You must keep your repository in sync with the changes from everyone else that is collaborating on it with you.
* You do this with the git fetch command.
* Fetching is closely related to git pull.
* The only difference between a pull and a fetch is that pulling merge the changes after Git fetches them.

Fetch changes from remote repository

* Fetching changes from a remote repository, retrieves (literally fetches) the changes from that remote repository.
  + The changes are stored in the remote branches on your local repository.
  + You can use this to see what changes are on the remote repository without affecting your local repository.
  + prompt> git fetch <remote name>
  + ... example ...
  + prompt> git fetch tswicegood
  + remote: Counting objects: 39, done.
  + remote: Compressing objects: 100% (25/25), done.
  + remote: Total 39 (delta 16), reused 26 (delta 9)
  + Unpacking objects: 100% (39/39), done.
  + From git://github.com/tswicegood/bobby-tables
  + \* [new branch] master -> tswicegood/master
* Git fetches the changes from the origin remote repository, if you don’t specify a remote.
  + You can fetch from another repository by providing the name of that remote repository.
* By default, git fetch fetches all branches from a remote repository.
  + You can change this depending on the parameters you provide to git fetch.
* You can fetch a specific branch by calling git fetch with an explicit remote name and a refspec.
  + Refspecs provide the name of the remote branch and the branch in your local repository that it should be fetched into – separated by a colon.
  + For example, to fetch only the master branch from your origin branch, use this:
  + prompt> git fetch origin master:remotes/origin/master

Fetch changes from multiple remote repositories

* You can fetch changes from multiple remotes at one time.
* You can use the --multiple parameter to provide Git with multiple remotes to fetch changes from.
  + prompt> git fetch --multiple remote1 remote2 ... and so on ...

Fetch changes from all remote repositories

* Use --all to tell Git to go through all of your remotes and fetch the changes from them.
  + prompt> git fetch --all
  + ... example after adding another remote ...
  + prompt> git fetch --all
  + Fetching tswicegood
  + Fetching petdance
  + remote: Counting objects: 414, done.
  + remote: Compressing objects: 100% (161/161), done.
  + remote: Total 407 (delta 231), reused 397 (delta 227)
  + Receiving objects: 100% (407/407), 52.53 KiB, done.
  + Resolving deltas: 100% (231/231), completed with 2 local objects.
  + From http://github.com/petdance/bobby-tables
  + \* [new branch] master -> petdance/master

**Task 21 – Retrieving Remote Changes**

* We need to know the distinction between git fetch and git pull to understand how git handles remote repositories.

Pull changes from a remote repository

* You need to know that - remotes are read-only branches.
  + The changes from a remote repository are fetched into branches that are stored locally instead of committing directly to them. The fetched changes are merged as and when necessary.
  + You can use git pull to combine fetching and merging into one command.
* If you do not specify a remote, git pull assumes that you want to pull from the origin remote repository.
* You must specify a remote if you want to provide a specific branch to pull from.
  + prompt> git pull [name [branch name]]
  + ... example ...
  + prompt> git pull tswicegood master
  + From git://github.com/tswicegood/bobby-tables
  + \* branch master -> FETCH\_HEAD

Pull changes from a different branch into your local branch

* You can provide a full refspec (two branches separated by a colon) to control – which branch you are pulling from and which branch you want those changes to end up in.
  + You specify the remote branch before the colon and the local branch after the colon.
* You can pull into branches that don’t exist.
  + prompt> git pull origin <remote branch>:<local branch>
  + ... example ...
  + prompt> git pull origin development:team-dev

Pull changes and rebase instead of merge

* You can use the --rebase option to tell Git – To rebase your local changes on top of the remote changes (instead of performing a merge).
  + This is the equivalent of running git fetch followed by git rebase.
  + This *allows you to cleanly apply all your local changes on top of the remote changes that have already been shared*.
* To fetch from origin and rebase against its main branch, use this:
  + prompt> git pull --rebase origin master
  + From git://github.com/origin/bobby-tables
  + \* branch master -> FETCH\_HEAD
  + First, rewinding head to replay your work on top of it...
  + Applying: add <meta> tags

**Task 22 - Sending Changes to Remotes**

Push the local tracking branch to your origin

* You have to publish your repository somewhere that is accessible to other members of your team.
* You send your changes to that repository using the git push command.
  + prompt> git push.

Push first set of changes from master to your origin

* If you call git push without any parameters, it causes Git to assume that – you want to push the current local branch to a branch with same name on the origin repository.
* You can provide both a remote repository and a branch name.
  + prompt> git push origin master

Push changes from a specific branch to a specific remote repository

* git push takes both named remote repositories (that is, those that have been added via git remote add) and full URLs – to remote repositories.
* You can provide a branch name to specify which branch to push, but in order to provide a branch, you must specify a remote repository.
  + For example, to push your beta branch to your origin repository, you would use this: git push origin beta.
  + prompt> git push <remote name> <branch name>
  + ... example ...
  + prompt> git push tswicegood development
  + Counting objects: 42, done.
  + Delta compression using up to 2 threads.
  + Compressing objects: 100% (33/33), done.
  + Writing objects: 100% (42/42), 39.49 KiB, done.
  + Total 42 (delta 18), reused 5 (delta 4)
  + Unpacking objects: 100% (42/42), done.
  + To git://internal.domain51.pvt/sample-repo.git
  + \* [new branch] development -> development
* For example, to push to the production branch to the shared remote, use this:
  + prompt> git push shared production

Push changes from the local master branch to the remote production branch

* If you want to push to a remote branch that is named differently than your local branch, You can also use the refspec syntax.
  + The syntax of a refspec for git push is – two branches separated by a colon (the local branch first followed by the name of the remote branch).
  + For example, pushing your local master branch to a remote branch named beta, you use this: git push origin master:beta.
  + prompt> git push origin master:production

Force a remote branch to accept a push

* You normally don’t want to update a remote repository without using a fast-forward push, but in those rare cases, you can use the --force or -f option to force Git to allow the update.
  + For example, you committed sensitive information or are removing a commit that does not need to be recorded.
* Warning: Use this with extreme caution because – It can cause others to get out of sync with the repository to which you are pushing.
  + prompt> git push --force
  + ... or ...
  + prompt> git push -f

**Task 23 - Handling Remote Tags and Branches**

* You can push your tags to a remote repository by one of two mechanisms:
  1. You can call git push and supply the tag name as the reference to push.

Push tag v1.0 to the origin

* + prompt> git push origin v1.0
  + Total 0 (delta 0), reused 0 (delta 0)
  + To git://internal.domain51.pvt/sample-repo.git
  + \* [new tag] v1.0 -> v1.0

1. You can add the --tags parameter to git push to push all your tags to the remote.

Push all tags to the origin

* + prompt> git push --tags origin
  + Total 0 (delta 0), reused 0 (delta 0)
  + To git://internal.domain51.pvt/sample-repo.git
  + \* [new tag] v0.8 -> v0.8
  + \* [new tag] v0.9 -> v0.9

Fetch remote tags and update local tags

* Most tags are fetched automatically.
* Fetching changes from master – that has several tags in its history – causes Git to fetch those tags as well.
* You can explicitly fetch tags via git fetch --tags.
  + prompt> git fetch --tags origin
  + remote: Counting objects: 42, done.
  + remote: Compressing objects: 100% (37/37), done.
  + remote: Total 42 (delta 18), reused 0 (delta 0)
  + Unpacking objects: 100% (42/42), done.
  + From git://internal.domain51.pvt/sample-repo.git
  + \* [new tag] v0.8 -> v0.8
  + \* [new tag] v0.9 -> v0.9
  + \* [new tag] v1.0 -> v1.0
* Delete the remote branch called beta
* You can delete remote branches with a special-case refspec and git push:<branch to delete>.
* This is the equivalent of pushing an empty branch to the remote repository.
  + prompt> git push origin:beta
  + From git://internal.domain51.pvt/sample-repo.git
  + - [deleted] beta

|  |
| --- |
| **Branches and Merging Revisited** |

**Task 24 – Handling Conflicts**