



[ANDROID](#) |
 [CORE JAVA](#) |
 [DESKTOP JAVA](#) |
 [ENTERPRISE JAVA](#) |
 [JAVA BASICS](#) |
 [JVM LANGUAGES](#) |
 [SOFTWARE DEVELOPMENT](#) |
 [DEVOPS](#)

[Home](#) »
 [Software Development](#) »
 [Git](#) »
 Git Commands Tutorial

ABOUT MAYANK GUPTA



Senior JEE developer with experience in large scale IT projects, especially in the telecommunications and financial services sectors. Mayank has been designing and building J2EE applications since 2007. Fascinated by all forms of software development; keen to explore upcoming areas of technology like AI, machine learning, blockchain development and AR. Lover of gadgets, apps, technology and gaming.



Git Commands Tutorial

Posted by: Mayank Gupta |
 in Git |
 June 2nd, 2017

If you are a software developer, there is a pretty good chance that you must have used version control software in one form or the other. Git has surged in popularity over the past few years succeeding SVN and CVS. With almost every open source project migrated to Git as a version control of choice, it currently seems like a de-facto standard for the version control in the programming community.

Want to be a GIT Master ?

Subscribe to our newsletter and download GIT Tutorial eBook right now!

In order to help you master GIT, we have compiled a kick-ass guide with all the basic concepts of the GIT version control system! Besides studying them online you may download the eBook in PDF format!

Email address:

[Sign up](#)

Tip

You may skip the introductory part and jump directly to the **commands tutorial** below.

Table Of Contents

1. What is version control?
 - 1.1 Benefits of VCS
 - 1.2 Centralized Vs Distributed VCS
2. Why Git?
3. Installing Git Client
4. Git Commands
 - 4.1 Initial Git Configuration [git config]
 - 4.2 Asking Git for help [git help]
 - 4.3 Creating a new repository [git init]
 - 4.4 Checking out existing repository [git clone]
 - 4.5 Check the status of your project [git status]
 - 4.6 Start tracking new files [git add]
 - 4.7 Files never to be committed [git ignore]
 - 4.8 What exactly got changed [git diff]
 - 4.9 Commit your changes [git commit]
 - 4.10 Removing and Deleting tracked files [git rm]
 - 4.11 Tracking renamed files [git mv]
 - 4.12 Show me the history [git log]
 - 4.13 Show me the remote location [git remote]
 - 4.14 Get Remote Data/Files [git fetch, git pull]
 - 4.15 Send local changes to remote [git push]

NEWSLETTER

177,617 insiders are already weekly updates and complimentary whitepapers!

Join them now to gain access to the latest news in as well as insights about Android, Groovy and other related technologies.

Email address:

[Sign up](#)

JOIN US



With **1,500** unique videos placed at your disposal, you can learn at your own pace. Constantly updated content to keep you motivated and encouraged.

So if you want to be a **guest writer** for Java Code Geeks, check out our **JCG** partners program. We'll be happy to have you on board and improve your writing skills!

1. What is version control?

Version control (a.k.a Version Control System or VCS) is a system that records changes to a file or set of files over time so that you can recall specific versions later. It keeps the entire change history of your project file by file which you call changeset.

1.1 Benefits of VCS:

You need version control so that:

1. you can revert back to to last working edition of the file or the complete code in the event of a failure or error
2. multiple parties can work in a collaborative way over one code base; they don't need to share pieces of code or patches. VCS helps merging the code from many developers easier.
3. developers don't need to comment out certain pieces of code to disable certain functionality. They can maintain different versions of the same codebase containing different changes which can be merged with the main repository when desired.

1.2 Centralized Vs Distributed VCS

VCS like SVN and CVS are examples of Centralized VCS (CVCS) while Git is a Distributed VCS (DVCS).

CVCS has only one single place where full version history of the software is maintained. In DVCS, every developer's working copy of the code is also a repository that can contain the full history of all changes.

CVCS being centralized has single point of failure. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they're working on.

In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files: they fully mirror the repository. Thus if any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.

Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren't possible in centralized systems, such as hierarchical models.^[1]

2. Why Git?

While many VCS are in existence today, Git has fast picked up pace as a version control of choice especially by the open source programming community.

1. **Distributed** – This is at the heart of Git and is a fundamental concept in Git. In CVCS like SVN, SVN keeps the track of all the changes in a central repository and all the clients contain no change history, if the master repository is lost all the history is lost. Git, on the other hand, is a DVCS. Every client has the complete history of all the changes.
2. **Speed (Fast branching and merging)** – In Subversion, branch creation was easy but it used to get created as another directory. When you need to switch to a different branch, it is like switching to a different codebase which consumes a lot of time and also a lot of space on your local machine. Unlike SVN, Git contains just a single copy for all the branches and switching from one branch to another happens in a breeze; saves both time and space.
3. **Strong support for non-linear development (thousands of parallel branches)** – Git stores all the information of different branches locally as well in a single copy and not creating a cloned copy for every branch results in a lot of space saving. Thus developers have come up with unique development models like creating a new branch for every enhancement and/or bugfix annotating them as such; thus making the release management process easier.

3. Installing Git Client

For the purpose of this tutorial we are going to use the following tools on a Windows 10 platform:

1. **Git Bash** – It is the command line tool for Git. We will use all our commands for the purpose of this tutorial from this tool. The most official build is available for download on the Git website. Just go to <http://git-scm.com/download/win> and the download will start automatically. Note that this is a project called Git for Windows, which is separate from Git itself; for more information on it, go to <https://git-for-windows.github.io/>. *You may refer this page for installation on other platforms. Please note, irrespective of the platform used, the git commands remain the same.*
2. **GitHub** – This is going to serve as the remote location where we will keep our code/files for this tutorial. Go to <https://github.com/> and create an account. You can create public and private repositories here which can then be accessed from anywhere as long as you have an internet connection.

4. Git Commands

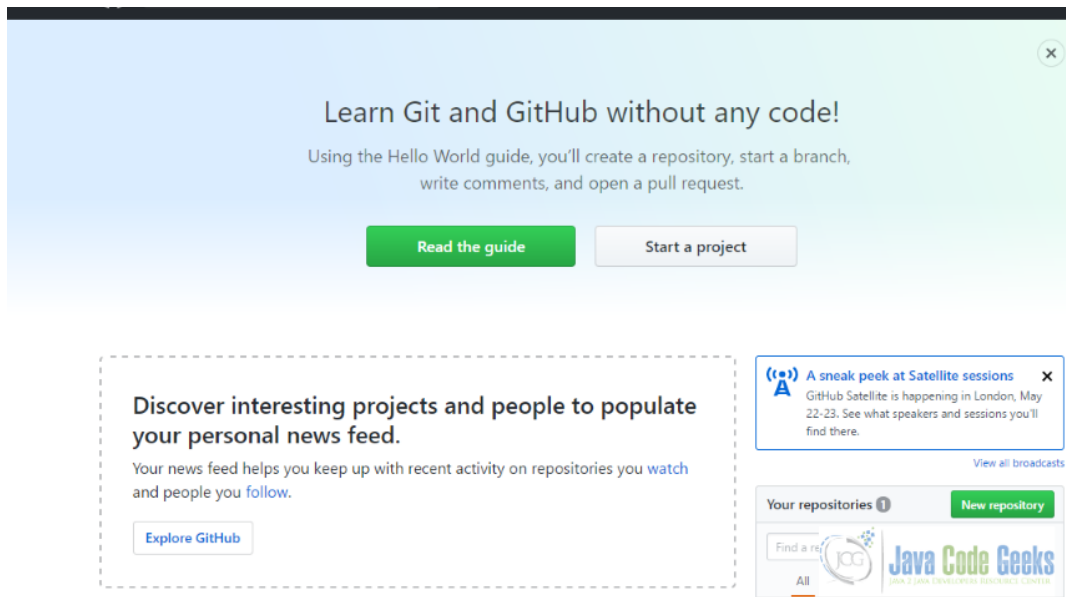
There are a lot of different ways to use Git. There are the original command line tools, and there are many graphical user interfaces of varying capabilities. This tutorial provides a succinct overview of the most important Git commands.

All the commands will be run in Git Bash.

Before beginning with the commands, let's create our first repository on Github:

1. Once you have created your account, you should see this page on <https://github.com/>.







Github Welcome Page

2. Click on "Start a project" and you will taken to screen with the following options.

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner **Repository name**

 mayankbindas / 


Great repository names are short and memorable. Need inspiration? How about **turbo-invention**.


Description (optional)

☒ **Public**
Anyone can see this repository. You choose who can commit.

☐ **Private**
You choose who can see and commit to this repository.

☐ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

| 



Create New Repository on Github

3. Put repository name as "git-commands-tutorial" and a description. You may mark the repository as public or private. If marked private, it will be accessible only to those you want to. Click on create repository, you will see screen like this.

Quick setup — if you've done this kind of thing before

or

We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```

echo "# git-commands-tutorial" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/mayankbindas/git-commands-tutorial.git
git push -u origin master
  
```


...or push an existing repository from the command line

```

git remote add origin https://github.com/mayankbindas/git-commands-tutorial.git
git push -u origin master
  
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.



Repository Created Screen

- Click on README link and it will take you to another page. Rename README.md to project-plan.txt. Scroll down a bit and put comments for the commit – “creating project-plan.txt”. Click “Commit New File” and your first commit to git repository will be there. We will later do the same through command.

4.1 Initial Git Configuration [git config]

Git comes with a tool called

```
git config
```

that lets you get and set configuration variables that control all aspects of how Git looks and operates.

The first thing you should do when you install Git is to set your user name and email address. This is important because every Git commit uses this information.

```

1 git config --global user.name "Mayank Gupta"
2
3 git config --global user.email "mayankbindas@gmail.com"
  
```

You can set global settings or project specific settings with the help of this command. If you want to override this with a different name or email address for specific projects, you can run the command without the

```
--global
```

option when you're in that project.

If you want to check your settings, you can use the

```
git config --list
```

command to list all the settings Git can find at that point.

You can also check what Git thinks a specific key's value is by typing

```
git config <key>.
```

4.2 Asking Git for help [git help]

There are three ways to get the manual help page or manpage for any git command:

```

1 git help <verb>
2
3 git <verb> --help
4
5 man git-<verb>
  
```

For example, if we want help on config command, we have the following three options:

```

1 git help config
2
3 git config --help
4
5 man git-help
  
```



git-config(1) Manual Page

NAME

git-config - Get and set repository or global options

SYNOPSIS

```
git config [<file-option>] [type] [--show-origin] [-z|--null] name [value [value_regex]]
git config [<file-option>] [type] --add name value
git config [<file-option>] [type] --replace-all name value [value_regex]
git config [<file-option>] [type] [--show-origin] [-z|--null] --get name [value_regex]
git config [<file-option>] [type] [--show-origin] [-z|--null] --get-all name [value_regex]
git config [<file-option>] [type] [--show-origin] [-z|--null] [--name-only] --get-regexp name_regex [value_regex]
git config [<file-option>] [type] [-z|--null] --get-urlmatch name URL
git config [<file-option>] --unset name [value_regex]
git config [<file-option>] --unset-all name [value_regex]
git config [<file-option>] --rename-section old_name new_name
git config [<file-option>] --remove-section name
git config [<file-option>] [--show-origin] [-z|--null] [--name-only] -l | --list
git config [<file-option>] --get-color name [default]
```



Git Man Page

4.3 Creating a new repository [git init]

git init is used to convert a project or a file system into a Git project. Go to an existing directory on your file system and type git init.

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.

Executing git init creates a .git subdirectory in the project root, which contains all of the necessary metadata for the repo. Aside from the .git directory, an existing project remains unaltered (unlike SVN, Git doesn't require a .git folder in every subdirectory).

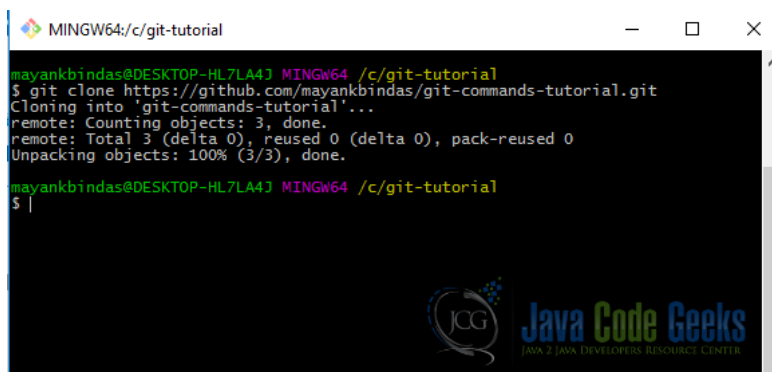
However, for most projects, git init only needs to be executed once to create a central repository—developers typically don't use git init to create their local repositories.

Since we have created our project using Github, we will skip to run this command.

4.4 Checking out existing repository [git clone]

Create a folder called "git-repository" on your Windows machine. Go inside this folder and right click, you will see "Git Bash Here" as one of the option. Click on it. Git Bash will open and we will run all our git commands in it.

Type the following command and this will checkout our project created on github.



git clone



In the lifecycle of a project, git init, git clone are mostly one time operations. git config (with global settings) is also mostly a one time operation on your system.

4.5 Check the status of your project [git status]

Run git status in Git Bash. You will see an output like this

```
$ cd git-commands-tutorial/
mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ |
```

git status

Running git status shows you which files are ignored (untracked files), which files are modified and needs to be committed. Currently we don't have a clean directory with no modified tracked files and no untracked files. We will continue to explore git status more as we use other commands.

Let's create a new file called "README.txt" and run git status again.

```
MINGW64:/c/git-tutorial/git-commands-tutorial
mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README.txt

nothing added to commit but untracked files present (use "git add" to track)
mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ |
```

git status – Untracked Files

README.txt is untracked file for now.

4.6 Start tracking new files [git add]

Untracked file means they are new files (weren't present in the last snapshot or commit). Git won't start tracking it unless it is explicitly told to do so. To start tracking new files, you run

```
1 | git add README.txt
```

Git will start tracking changes to README.txt from now on. Let's run git status now.

```
MINGW64:/c/git-tutorial/git-commands-tutorial
mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ git add README.txt
mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README.txt
mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ |
```

git add

The git add command takes a path name for either a file or a directory; if it's a directory, the command adds all the files in that directory recursively.

git add is also used to stage an existing file which was being tracked previously and has been modified. It is often called termed as "add this content to the next commit" rather than "add this file to the project".

```

mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   project-plan.txt

mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ |

```

git status – modified files

Run git add project-plan.txt and hit git status again

```

MINGW64:/c/git-tutorial/git-commands-tutorial
mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ git add project-plan.txt
mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README.txt
    modified:   project-plan.txt

mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ |

```

git add – modified files

Suppose you modify any of these file again then you have to stage them again by running git add otherwise the last snapshot will be committed (this can be checked by running git status before a commit which is always a good idea). We have two staged files now which can be committed.

4.7 Files never to be committed [git ignore]

Create project.temp file in your local git project directory. This is the file which we want to keep on our local and never want to commit.

Run git status now, you will see an output like this

```

MINGW64:/c/git-tutorial/git-commands-tutorial
mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README.txt
    modified:   project-plan.txt

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

    project.temp

mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ |

```

git status – untracked files

Git has tracked files and untracked files. There are some files (compiled code files, system generated temp files, build files etc.) which you may not ever want to check-in into the repository and also don't want git to show you that they needs to be added.

```

1 cat .gitignore
2 *~
3 *.temp

```

First line tells git to ignore files ending with an underscore (.). Second line tells git to ignore files with temp extension.

The rules for the patterns you can put in the .gitignore file are as follows:[1]

1. Blank lines or lines starting with # are ignored.
2. Standard glob patterns work.
3. You can start patterns with a forward slash (/) to avoid recursivity.
4. You can end patterns with a forward slash (/) to specify a directory.
5. You can negate a pattern by starting it with an exclamation point (!).

4.8 What exactly got changed [git diff]

Currently we have three files; one new addition, one modified and one that we want to ignore. Let's run git status

```

MINGW64:/c/git-tutorial/git-commands-tutorial
mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   README.txt
        modified:   project-plan.txt

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

        .gitignore

mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ |

```

git status

If you want to know exactly what you changed, not just which files were changed – you can use the git diff command. This command compares what is in your working directory with what is in your staging area. The result tells you the changes you've made that you haven't yet staged.

If you want to see what you've staged that will go into your next commit, you can use git diff --staged. This command compares your staged changes to your last commit:

```

MINGW64:/c/git-tutorial/git-commands-tutorial
mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ git diff
mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ git diff --staged
diff --git a/README.txt b/README.txt
new file mode 100644
index 0000000..e69de29
diff --git a/project-plan.txt b/project-plan.txt
index e69de29..1ae82b8 100644
--- a/project-plan.txt
+++ b/project-plan.txt
@@ -1,2 +1,4 @@
# git-commands-tutorial
Git Commands Tutorial
+dummy project plan

mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ |

```

git diff



git diff by itself doesn't show all changes made since your last commit – only changes that are still unstaged. This can be confusing, because if you've staged all of your changes, git diff will give you no output.

Run git diff --cached to see what you've staged so far.


```

er)
$ git diff --cached
diff --git a/README.txt b/README.txt
new file mode 100644
index 0000000..e69de29
diff --git a/project-plan.txt b/project-plan.txt
index e6cc8f9..1ae82b8 100644
--- a/project-plan.txt
+++ b/project-plan.txt
@@ -1,2 +1,4 @@
# git-commands-tutorial
Git Commands Tutorial
+
+dummy project plan
mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ |

```

git diff --cached

4.9 Commit your changes [git commit]

Remember that anything that is still unstaged – any files you have created or modified that you haven't run git add on since you edited them – won't go into this commit. That's why it is always a good idea to run git status before you commit your changes.

To commit your changes run

```
git commit -m <commit_message>
```

```
1 | git commit -m "first commit"
```



Commit creates a snapshot of the project's changes which were staged by using git add command. Anything you didn't stage is still sitting there modified; you can do another commit to add it to your history. Every time you perform a commit, you're recording a snapshot of your project that you can revert to or compare to later.

Run git status again and you will again see a clean workarea.

```

MINGW64:/c/git-tutorial/git-commands-tutorial
mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ git commit -m "first commit"
[master f4b58f0] first commit
2 files changed, 2 insertions(+)
create mode 100644 README.txt
mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
(use "git push" to publish your local commits)
Untracked files:
  (use "git add <file>..." to include in what will be committed)

.gitignore

nothing added to commit but untracked files present (use "git add" to track)
mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ |

```

git commit

4.10 Removing and Deleting tracked files [git rm]

There are two ways to delete files from your workarea and remove it from staging so that it won't be tracked anymore:

- 1) Run git rm <filename> which will remove the file from being tracked and also from your working area.
- 2) Delete the file and then run git rm <filename>. Removing the file from your working directory will show the file as changed and not deleted. Hence, it is necessary to run git rm to remove it from being tracked.

Note, you have to run git commit in both the cases so that git stops tracking them.

You can use the same command even when you don't want to delete the file but want to untrack it.

```
git rm --cached <filename>
```

4.11 Tracking renamed files [git mv]

If you rename a file, Git would have no idea that it happened. You would have to run git rm old_file and git add new_file. However, with this approach git would lose all the previous history of the old file. Run git mv <old_file> <new_file> to do it correctly. Try it as shown below.

```

$ git mv project-plan.txt code-plan.txt
mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
(use "git push" to publish your local commits)
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    project-plan.txt -> code-plan.txt

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

        .gitignore

```

git mv

4.12 Show me the history [git log]

git log by default shows the commit history in the reverse chronological order. There are variety of options available to filter out the results.

For example, git log -3 will show you last 3 commits only. (git log -<n>). git log --since=2.weeks will show the commits made in the past 2 weeks.

You can also filter the list to commits that match some search criteria. The --author option allows you to filter on a specific author, and the --grep option lets you search for keywords in the commit messages. (Note that if you want to specify both author and grep options, you have to add --all-match or the command will match commits with either.)

```

MINGW64:/c/git-tutorial/git-commands-tutorial
mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ git log
commit f4b58f904befa5a257f5c8bc48c2fb04ebe11e09
Author: Mayank Gupta <mayankbindas@gmail.com>
Date:   Wed May 31 09:53:51 2017 +0530

    first commit

commit 8870dbbdc543b4a592520aece184566f802853a
Author: mayankbindas <mayankbindas@users.noreply.github.com>
Date:   Wed May 31 08:49:00 2017 +0530

    creating project-plan.txt

mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ git log -1
commit f4b58f904befa5a257f5c8bc48c2fb04ebe11e09
Author: Mayank Gupta <mayankbindas@gmail.com>
Date:   Wed May 31 09:53:51 2017 +0530

    first commit

mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ git log --since="2017-01-01" --author=mayankbindas
commit f4b58f904befa5a257f5c8bc48c2fb04ebe11e09
Author: Mayank Gupta <mayankbindas@gmail.com>
Date:   Wed May 31 09:53:51 2017 +0530

    first commit

commit 8870dbbdc543b4a592520aece184566f802853a
Author: mayankbindas <mayankbindas@users.noreply.github.com>
Date:   Wed May 31 08:49:00 2017 +0530

    creating project-plan.txt

mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ |

```

git log

4.13 Show me the remote location [git remote]

This command shows you from where you have cloned your repository. "origin" is the default name Git gives to the server you cloned your repository from.

```

(r)
$ git remote
origin

mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ git remote -v
origin https://github.com/mayankbindas/git-commands-tutorial.git (fetch)
origin https://github.com/mayankbindas/git-commands-tutorial.git (push)

mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ |

```

git remote

"-v" option shows the remote URL of the repository. Notice (fetch) and (push) against the remote URL. This shows the permission the author has on the repository. Having both means you have write access to the remote repository.

4.14 Get Remote Data/Files [git fetch, git pull]

The syntax is like:

```
git fetch [remote-name]
```

The command git fetch automatically adds the remote repository under the name "origin". It's the same as git fetch origin.

NOTE: git fetch only downloads the data to your local repository and doesn't merge it with any file that has changed locally. The changes have to be merged manually.

Running git pull merges the code as well. Let's create a new file remotely on github "test-fetch.txt" and run git fetch.

```

MINGW64:/c/git-tutorial/git-commands-tutorial
mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ git fetch
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 2 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (2/2), done.
From https://github.com/mayankbindas/git-commands-tutorial
 8870dbb..927e40d master -> origin/master

mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ git pull
error: Your local changes to the following files would be overwritten by merge:
code-plan.txt
Please commit your changes or stash them before you merge.
Aborting

mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ |

```

git fetch and git pull

4.15 Send local changes to remote [git push]



Unless you do a git push, all the changes whether staged or committed happens locally on your machine and remote is not aware of it. Git creates the snapshot for all these changes and you push this snapshot to the remote.

The command for this is: git push [remote-name] [branch-name]. If you want to push your master branch to your origin server, then you can run this to push any commits you've done back up to the server:

```
1 | git push origin master
```

```

MINGW64:/c/git-tutorial/git-commands-tutorial
mayankbindas@DESKTOP-HL7LA4J MINGW64 /c/git-tutorial/git-commands-tutorial (master)
$ git push origin master

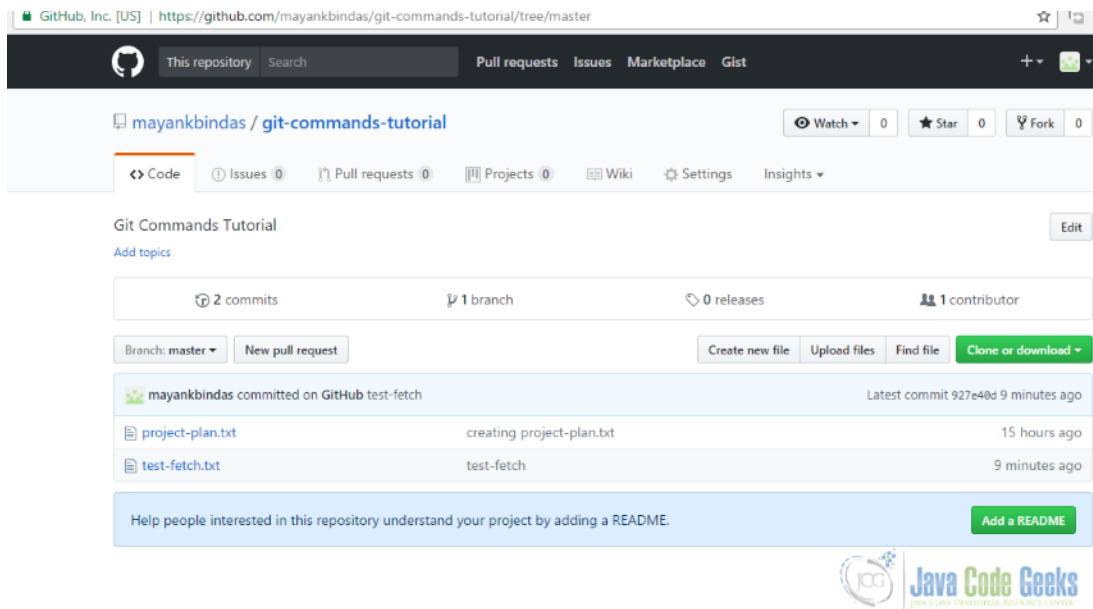
Counting objects: 8, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (8/8), 826 bytes | 0 bytes/s, done.
Total 8 (delta 1), reused 0 (delta 0)

remote: Resolving deltas: 100% (1/1), done.
To https://github.com/mayankbindas/git-commands-tutorial.git
 927e40d..c0895a3 master -> master

```

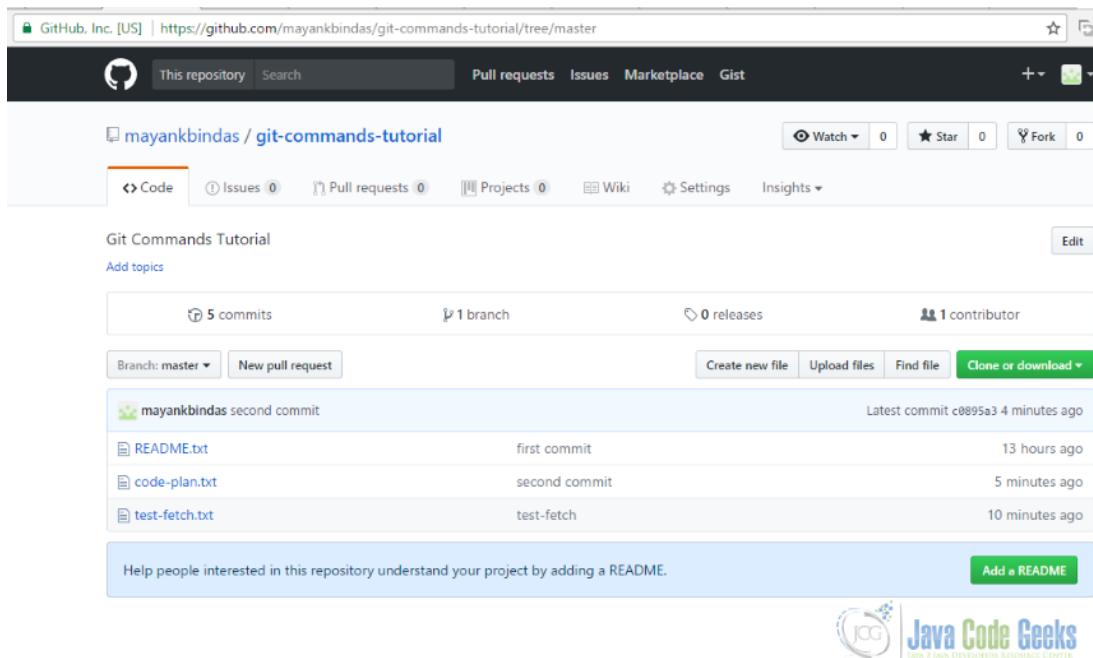
git push

Now all our local changes should be pushed to remote and anyone else tracking our project from remote can pull these changes.



Remote Repository – Before pushing the changes

After pushing the changes:



Remote Repository – After pushing the changes

NOTE: git push command will only work if you have write access on remote. Also, if there are changes on the remote that you don't have locally, you would have to pull them first before pushing yours.

5. Summary

Here is a list of all the commands we learnt in this tutorial. Git has a very long list of commands. Here we have learnt the most important ones that can easily get you started with Git.

1. Initial Git Configuration [git config]
2. Asking Git for help [git help]
3. Creating a new repository [git init]
4. Checking out existing repository [git clone]
5. Check the status of your project [git status]
6. Start tracking new files [git add]
7. Files never to be committed [git ignore]
8. What exactly got changed [git diff]
9. Commit your changes [git commit]
10. Removing and Deleting tracked files [git rm]

13. Show the the remote location [git remote]
14. Get Remote Data/Files [git fetch, git pull]
15. Send local changes to remote [git push]

6. References

1. <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>
2. <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>

Do you want to know how to develop your skillset to become a Java Rockstar?

Subscribe to our newsletter to start Rocking right now!
To get you started we give you our best selling eBooks for **FREE!**

1. JPA Mini Book
2. JVM Troubleshooting Guide
3. JUnit Tutorial for Unit Testing
4. Java Annotations Tutorial
5. Java Interview Questions
6. Spring Interview Questions
7. Android UI Design

and many more

Email address:

Sign up

KNOWLEDGE BASE

Courses

News

Resources

Tutorials

Whitepapers

THE CODE GEEKS NETWORK

.NET Code Geeks

Java Code Geeks

System Code Geeks

Web Code Geeks

HALL OF FAME

Android Alert Dialog Example

Android OnClickListener Example

How to convert Character to String and a String to Character Array in Java

Java Inheritance example

Java write to File Example

java.io.FileNotFoundException – How to solve File Not Found Exception

java.lang.arrayindexoutofboundsexception – How to handle Array Index Out Of Bounds Exception

java.lang.NoClassDefFoundError – How to solve No Class Def Found Error

JSON Example With Jersey + Jackson

Spring JdbcTemplate Example

ABOUT JAVA CODE GEEKS

JCGs (Java Code Geeks) is an independent online community focused on ultimate Java to Java developers resource center; targeted at the technical team lead (senior developer), project manager and junior developers. JCGs serve the Java, SOA, Agile and Telecom communities with daily new domain experts, articles, tutorials, reviews, announcements, code snippets and source projects.

DISCLAIMER

All trademarks and registered trademarks appearing on Java Code Geeks are the property of their respective owners. Java is a trademark or registered trademark of Oracle Corporation in the United States and other countries. Examples Java Code Geeks is not connected to Oracle Corporation and is not sponsored by Oracle Corporation.