# Git Intro Documentation

***Release 0.1***

**Jeffrey Poore**

**Jul 10, 2018**

# Contents:

CHAPTER 1

# Introduction

This document will focus on how you can do version control operations using Git. It is assumed that you already understand the basic concepts of version control like adding/removing files to a repository, committing changes to those files, etc. This is not an exhaustive guide on all the different ways that git commands can be used, but instead focuses on the most common. Git has extensive help documentation (for any command, the short version can be seen with a **-h** switch and the man page version can be seen with a **–help** switch).

Git was created in 2005 by Linux Torvalds for the development of the Linux kernel. It was inspired by BitKeeper and Monotone. It is a distributed version control system that has a number of characteristics that make it suitable for large numbers of developers (such as open source projects like the Linux Kernel):

- **Strong support for non-linear development**: git supports rapid branching and merging, and includes tools for visualizing a non-linear development history.
- **Distributed development**: git gives every developer a local copy of the development history and all changes are synchronized between repositories.
- **Simple APIs**: git has a number of different ways of interacting with tools, but they are all based on simple standard protocols such as HTTP, FTP, or SSH.
- **Scalable**: git has been developed to be fast even with large projects.

If you are familiar with version control repositories like SVN or CVS, you will find that git mostly works the same. There are some key differences, however:

- Git is a distributed repository, and each developer effectively maintains their own repository with a separate commit stream. Changes made in other repositories can be merged into a developer's local repository to make the two repositories effectively identical, but each repository can have commits that are not synchronized automatically to other repositories.
- Git is optimized for branching and merging, and so developers use branches often for single features. The main branch (called **master**) is intended to be always production ready, and developers usually work off a development or feature branch instead.
- Commits to a repository in git are not necessarily sequential. Normally they will be applied in order when doing merges, but in some cases, it may be required to reorder commits[1]. Because commits are not required to be

[1] A good example of this is when merging changes from a remote repository into a local repository when additional changes have occurred in

sequential, they are given random unique ids instead of sequential ones. These unique ids are long, but you can reference a commit by a portion of the id as long as it uniquely identifies the commit.

- There is only one commit stream across an entire repository. Unlike SVN or CVS where each branch has their own commit ids and history, git only has one history that is often visualized as lines that run in parallel with dots on those lines representing commits.

the local repository. If some of the incoming changes conflict, it could require a manual merge. These manual merges can often result in many different changes getting bundled into one big commit which then causes other developers to also have to do merges when those changes are pushed to other repositories. Git offers the option of doing a **rebase**, where some commits are rolled back, the incoming commits applied, and then the rolled back commits are re-applied with possible manual merges along the way. By doing this rebase, new changes are applied at the end which reduces the likelyhood of other developers having to manually merge when these changes are pushed.

CHAPTER 2

# Configuration

Git doesn't really require a lot of configuration to use (typically the minimum configuration is just a username and email address for when you make commits), but it does have some basic configuration options. Configuration options can be set for an individual repository, globally for all respositories owned by the current user, or system wide. To set a configuration for the current repository, you use the **config** command:

```
$ git config core.ignoreCase true
```

Adding the **–global** switch sets a configuration globally for the current user:

```
$ git config --global user.name "Jeffrey Poore"
```

Adding the **–system** switch sets a configuration globally for all users:

```
$ git config --system core.bigFileThreshold 2g
```

There are too many configuration options to really cover here, but you can find all the configuration options here:

https://git-scm.com/docs/git-config#_variables

CHAPTER 3

# Creating a Repository

It is very easy to start a project with git. You can create an empty project or you can start with a folder that already has content in it. You can also **clone** a repository from somewhere else, but that is covered in the *Using Remote Repositories* section. All you need to do to initialize a git repository is to run the **init** command:

```
$ mkdir newproject
$ cd newproject/
$ git init
Initialized empty Git repository in /home/tellmejeff/work/newproject/.git/
```

The working directory has been initialized, and a **.git** directory is created at the root of the project to store all the metadata about the repository.

CHAPTER 4

# Updating a Repository

## 4.1 The Working Set

Adding or updating files in git is a multi-step process. The first thing you need to understand is that there is a working set (often called the stage, index, or the files to be committed - I prefer the term working set, and this document will use that term) that determines what files will be added or updated. Unlike CVS or SVN, a commit doesn't automatically add all the changes that are working copy (it is possible, however, with git to do that). Most developers have at one point or another accidentally committed changes they didn't intend to commit, and git fixes this by requiring developers to explicitly add files to be included to the working set. For example, if I run a commit with no files in the working set, even though there are changes that could be committed, git will inform me that there are no files staged to be committed:

```
$ git commit
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
      modified:   README.txt

no changes added to commit
```

You can see what files are in the working set by running the **status** command, which clearly deliniates which files are in the working set and which files are not:

```
$ 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)

      modified:   runme.sh

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
```

(continues on next page)

(continued from previous page)

```
  (use "git checkout -- <file>..." to discard changes in working directory)

      modified:   README.txt
```

## 4.2 Adding/Removing Files to the Working Set

You can add one or more files to the working set with the **add** command:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
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:   README.txt
      modified:   fix.txt
      modified:   pom.xml
      modified:   runme.sh

no changes added to commit (use "git add" and/or "git commit -a")
$ git add README.txt fix.txt pom.xml runme.sh
$ 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)

      modified:   README.txt
      modified:   fix.txt
      modified:   pom.xml
      modified:   runme.sh
```

To remove one or more files from the working set, you use the **reset** command. Issuing a reset command with no arguments will clear out the working set (it will not revert the changes):

```
$ 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)

      modified:   README.txt
      modified:   fix.txt
      modified:   pom.xml
      modified:   runme.sh

$ git reset
Unstaged changes after reset:
M     README.txt
M     fix.txt
M     pom.xml
M     runme.sh
$ git status
```

(continues on next page)

(continued from previous page)

```
On branch master
Your branch is up-to-date with 'origin/master'.
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:   README.txt
      modified:   fix.txt
      modified:   pom.xml
      modified:   runme.sh

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

The **reset** command can also change where the special **HEAD** tag in the repository points to, based on some switches and an optional commit id:

```
$ git reset [--soft|--mixed|--hard|--keep] [commit_id]
```

Depending on the following switches, you can reset changes in the commit stream, working set, or working copy:

- **–soft**: if a commit id is specified, the **HEAD** tag will be moved to the specified id. The working set and working copy are unchanged. If no id is specified, this results in no action.
- **–mixed**: this will clear the working set but not the working copy, so changes are essentially unstaged. If a commit id is also specified, the **HEAD** tag will be moved to the specified id. This is the default.
- **–hard**: this will clear the working set and discard any changes in the working copy. If a commit id is specified, only the changes since that id are discarded, and the **HEAD** tag will be moved to the specified id.
- **–merge**: this will clear the working set and then reset any files that had changes between the specified commit id and **HEAD** (you are resetting the repository back to the given commit id) but if any of those files also have changes that have not yet been committed to the repository, those changes will be preserved. If no commit id is specified, this behaves the same as **–mixed**.
- **–keep**: this will clear the working set and reset any files that had changes between the specified commit id and **HEAD**. If any of those files have changes that have not been committed to the repository, the reset will abort. If no commit id is specified, this behaves the same as **–mixed**.

If you just want to undo all changes and return to the current **HEAD** commit, you can also use the **checkout** command with a **-f** switch:

```
$ 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)

      modified:   README.txt
      modified:   fix.txt
      modified:   pom.xml
      modified:   runme.sh

$ git checkout -f
Your branch is up-to-date with 'origin/master'.
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

## 4.3 Ignoring Files

Git only updates files in a repository when they are part of the working set, so files that are temporary (like build files) won't show up in commands like **status** unless they are in directories under the root that are already versioned. If you want git to ignore files, you can create a **.gitignore** file at any level of the repository (it applies to the containing directory and any subdirectories). The lines inside the file are patterns that should be matched against. Pattern syntax:

- a blank line matches no files - it is just whitespace for readability
- a line starting with **#** is a comment
- a line starting with **!** is a negated pattern, so any file not matching the rest of the pattern is ignored
- a line ending in a slash is considered a directory and will match that directory and any files under it (**foo/** will match the directory foo and all paths below it, but will not match a symbolic link **foo**)
- a line not containing a slash is by default a glob pattern that matches files in the same directory as the .gitignore file
- a line starting with a slash matches the beginning of the pathname (**/*.c** will match all files with a **.c** extension in the same directory as the .gitignore file, but will not match **foobar/menu.c**)
- a single asterisk in a line will match any valid filename character except the file separator (**abc*.txt** will match files starting with **abc** and ending in **.txt** but doesn't include directories starting with **abc** containing files ending with **.txt**)
- a double asterisk followed by a slash will match all directories (****/foo** will match file or directory **foo** in any directory at the same level as the .gitignore file)
- a slash followed by a double asterisk will match everything inside that directory (**abc/**** will match everything inside the directory **abc**)
- a slash followed by a double asterisk followed by another slash matches zero or more directories (**a/**/b** will match **a/b**, **a/x/b**, **a/x/y/b**, etc.)

## 4.4 Committing the Working Set

When you are ready to commit your changes to the local respository, you just run the **commit** command. The command will either start your favorite editor so that you can provide a commit message, or you can pass the message with the **-m** switch:

```
$ 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)

      modified:   README.txt
      modified:   fix.txt
      modified:   pom.xml
      modified:   runme.sh

$ git commit -m 'modified a few files for a demo'
[master c040ec2] modified a few files for a demo
 4 files changed, 5 insertions(+), 3 deletions(-)
```

If you want to just commit all changes in the working copy without an explicit **add**, you can add the **-a** switch to your commit.

CHAPTER 5

# Comparing the Working Copy and Repository

Like in CVS and SVN, to see the changes in the working copy versus the current **HEAD** state of the repository, you use the **diff** command:

```
$ vi Main.groovy
$ git diff
diff --git a/src/Main.groovy b/src/Main.groovy
index 6191a10..29df7f5 100644
--- a/src/Main.groovy
+++ b/src/Main.groovy
@@ -3,12 +3,13 @@ import static Division.divide
 import static Subtract.subtract
 import static Sum.sum

-def name = "Matthew"
-int programmingPoints = 10
+def name = "Jeffrey"
+int programmingPoints = 20

 println "Hello ${name}"
 println "${name} has at least ${programmingPoints} programming points."
 println "${programmingPoints} squared is ${square(programmingPoints)}"
 println "${programmingPoints} divided by 2 bonus points is $
↪{divide(programmingPoints, 2)}"
 println "${programmingPoints} minus 7 bonus points is ${subtract(programmingPoints,␣
↪7)}"
-println "${programmingPoints} plus 3 bonus points is ${sum(programmingPoints, 3)}"
\ No newline at end of file
```

You can also compare the state of the repository against a given commit (compare working copy against repository at the time of the given commit) or compare two different commits (compare the state of the repository at those two commits):

```
$ git diff ebbbf77 45a30ea
diff --git a/src/test/java/com/ambientideas/AppTest.java b/src/test/java/com/
↪ambientideas/AppTest.java
```

(continues on next page)

(continued from previous page)

```
new file mode 100644
index 0000000..c1ea083
--- /dev/null
+++ b/src/test/java/com/ambientideas/AppTest.java
@@ -0,0 +1,43 @@
+package com.ambientideas;
+
+import junit.framework.Test;
+import junit.framework.TestCase;
+import junit.framework.TestSuite;
+
+//Pending comments
...
```

Note that the diff output is relative to the order of the commit ids. If you reverse the commit ids, the diff output will similarly be reversed:

```
$ git diff 45a30ea ebbbf77
diff --git a/src/test/java/com/ambientideas/AppTest.java b/src/test/java/com/
↪ambientideas/AppTest.java
deleted file mode 100644
index c1ea083..0000000
--- a/src/test/java/com/ambientideas/AppTest.java
+++ /dev/null
@@ -1,43 +0,0 @@
-package com.ambientideas;
-
-import junit.framework.Test;
-import junit.framework.TestCase;
-import junit.framework.TestSuite;
-
-//Pending comments
```

CHAPTER 6

# Viewing Repository History

Like CVS or SVN, you can see the history of changes to the repository by using the **log** command:

```
$ git log
commit ef7bebf8bdb1919d947afe46ab4b2fb4278039b3
Author: Jordan McCullough <jordan@github.com>
Date:   Fri Nov 7 11:27:19 2014 -0700

    Fix groupId after package refactor

commit ebbbf773431ba07510251bb03f9525c7bab2b13a
Author: Jordan McCullough <jordan@github.com>
Date:   Wed Nov 5 13:00:35 2014 -0700

    Update package name, directory

commit 45a30ea9afa413e226ca8614179c011d545ca883
Author: Jordan McCullough <jordan@github.com>
Date:   Wed Nov 5 12:59:55 2014 -0700

    Update package name, directory
...
```

Because of the long ids that git uses, the **log** command provides some alternative ways of specifying filter criteria for the log. The most commonly used options:

- **–author=<pattern> / –committer=<pattern>**
- **–until=<date> / –before=<date>**
- **–since=<date> / –after=<date>**
- **-<number> / -n <number> / –max-count=<number>**
- **-grep=<pattern>**

CHAPTER 7

# Moving or Removing Files From the Repository

If you want to move or remove files from the repository, use the **mv** or **rm** command (note that this doesn't immediately move or delete the files from the repository but instead adds a move/delete change to the working set that must then be committed):

```
$ git mv README.txt README.md
$ git rm fix.txt
rm 'fix.txt'
$ ls
build.gradle  pom.xml  README.md  resources  runme.sh  src
$ 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:    README.txt -> README.md
      deleted:    fix.txt
```

The mv or rm command will fail if your local copy is not up to date with the repository unless you pass a **-f** switch. Moving/removing directories requires the **-r** switch if the directory contains other files. If you aren't certain about the effects of the command, you can pass a **-n** switch to do a dry run:

```
$ ls
build.gradle  pom.xml  README.md  resources  runme.sh  src
$ git rm -r src -n
rm 'src/Division.groovy'
rm 'src/Main.groovy'
rm 'src/Square.groovy'
rm 'src/Subtract.groovy'
rm 'src/Sum.groovy'
rm 'src/main/java/com/github/App.java'
rm 'src/test/java/com/github/AppTest.java'
$ ls src
```

(continues on next page)

(continued from previous page)

```
Division.groovy  Main.groovy    Subtract.groovy  test
main             Square.groovy  Sum.groovy
```

If you would like to remove a file from the repository but keep it in your local filesystem, add the **–cached** switch:

```
$ ls
build.gradle  fix.txt  pom.xml  README.txt  resources  runme.sh  src
$ git rm --cached README.txt
rm 'README.txt'
$ ls
build.gradle  fix.txt  pom.xml  README.txt  resources  runme.sh  src
$ 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)

      deleted:    README.txt

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

      README.txt
```

# CHAPTER 8

## Stashing Changes

Every now and then, you have some changes in flight, but you are not ready to commit those changes. Imagine that you need to switch to a different branch or tag. You could clone the source repository from the specific branch or tag, but then you might end up with lots of copies of the repository, and this can get messy. Instead of making new repositories, you can instead **stash** your changes which effectively makes the working copy in sync with **HEAD** in the local repository (note that this stashes the working directory and the working set that is staged for commit):

```
$ vi src/Main.groovy
$ vi src/Square.groovy
$ git add src/Main.groovy
$ 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)

      modified:   src/Main.groovy

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

      modified:   src/Square.groovy

$ git stash
Saved working directory and index state WIP on master: ef7bebf Fix groupId after␣
↪package refactor
HEAD is now at ef7bebf Fix groupId after package refactor
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Notice that you can stash changes multiple times, and you can list the different stashes you have:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

      modified:   src/Sum.groovy

no changes added to commit (use "git add" and/or "git commit -a")
$ git stash
Saved working directory and index state WIP on master: ef7bebf Fix groupId after␣
↪package refactor
HEAD is now at ef7bebf Fix groupId after package refactor
$ git stash list
stash@{0}: WIP on master: ef7bebf Fix groupId after package refactor
stash@{1}: WIP on master: ef7bebf Fix groupId after package refactor
```

To see the contents of a specific stash, you can use the **show** command (you can add a **-p** switch to see the diff output):

```
$ git stash show stash@{1}
 src/Square.groovy | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)
$ git stash show -p stash@{1}
diff --git a/src/Square.groovy b/src/Square.groovy
index fde3319..2c0e09e 100644
--- a/src/Square.groovy
+++ b/src/Square.groovy
@@ -1,3 +1,4 @@
+// square a number
 static int square(int base) {
         base * base
-}
\ No newline at end of file
+}
```

To restore the most recent stash, you can use the **pop** command:

```
$ git stash pop
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

      modified:   src/Sum.groovy

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (a752e03b89490bd696d5c7234ebfe4456b295ec4)
```

You can also pop a specific stash:

```
$ git stash list
stash@{0}: WIP on master: ef7bebf Fix groupId after package refactor
stash@{1}: WIP on master: ef7bebf Fix groupId after package refactor
stash@{2}: WIP on master: ef7bebf Fix groupId after package refactor
$ git stash pop stash@{1}
```

(continues on next page)

(continued from previous page)

```
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

      modified:   src/Main.groovy

no changes added to commit (use "git add" and/or "git commit -a")
Dropped stash@{1} (218d1217408bfda1e7441a466d2bb84cbe97533a)
```

CHAPTER 9

# Using Branches

Branches in git are similar to branches in CVS and SVN, but they are also a little different. In CVS and SVN, a branch was effectively a separate commit stream - each commit in one branch is independent of any commits made in another branch. Git doesn't have separate commit streams; it's as if the branches are just subfolders of the overal project, and every commit includes everything in the repository. A lot of git tools will visualize the branches as vertical lines that are separate when the branches have differences, but the lines will come together when the branches become the same because of a merge:

Dots on the lines represent points at which a commit was made, and branches that don't have any changes at a given commit will not have a dot at that horizontal location. Branches may be displayed as stopping in the past (the last commit made on that branch was a while ago), but if a commit is made on that branch later on, the line will be extended up to the point where the commit was made. Two branches that were merged effectively look like a single line, but if there is a change made to one of the branches but not the other, the two lines will diverge.

## 9.1 The Master Branch

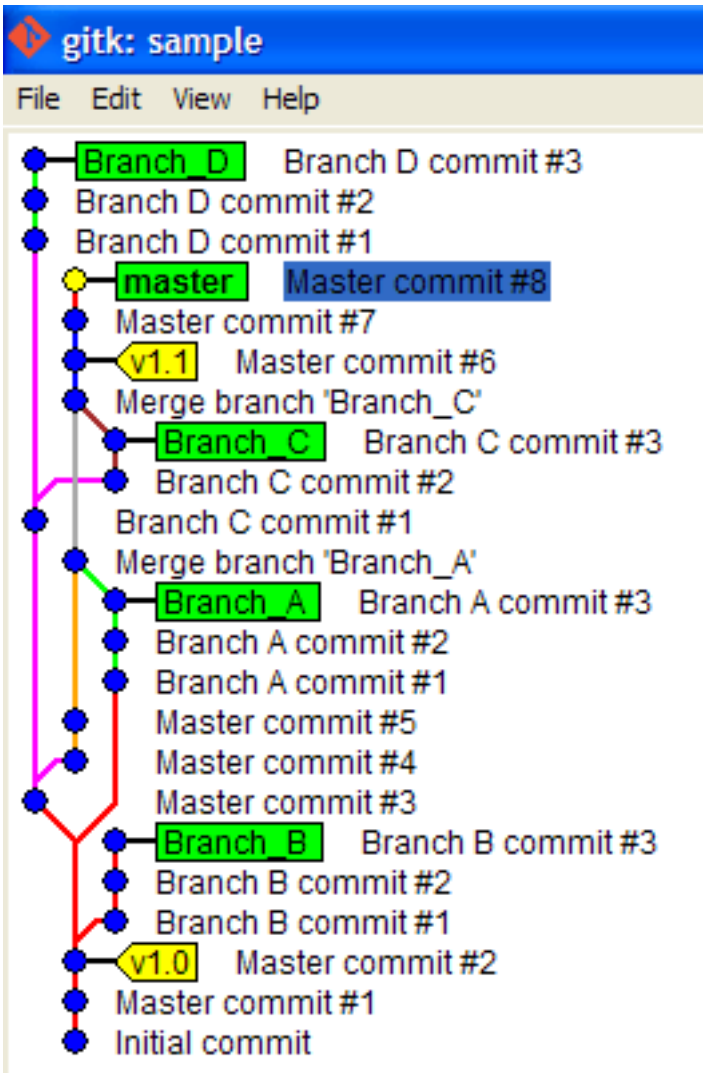
A new repository is always created with a default **master** branch. Most git methodologies consider the master branch to be production-ready and therefore it should not be the branch where main day-to-day development occurs. Instead, developers should always create a development branch and then when the development of a feature is done and tested, the changes are merged over from the development branch to the master.

## 9.2 Creating a New Branch

To create a new branch, you execute the **branch** command:

```
$ git branch dev
$ git branch
  dev
* master
```

As you can see, running the **branch** command without any arguments will list the existing branches.


gitk: sample
File Edit View Help
Branch_D
Branch D commit #3
Branch D commit #2
Branch D commit #1
master
Master commit #8
Master commit #7
v1.1
Master commit #6
Merge branch 'Branch_C'
Branch_C
Branch C commit #3
Branch C commit #2
Branch C commit #1
Merge branch 'Branch_A'
Branch_A
Branch A commit #3
Branch A commit #2
Branch A commit #1
Master commit #5
Master commit #4
Master commit #3
Branch_B
Branch B commit #3
Branch B commit #2
Branch B commit #1
v1.0
Master commit #2
Master commit #1
Initial commit

## 9.3 Switching Branches

To switch to a different branch, you use the **checkout** command:

```
$ git checkout dev
Switched to branch 'dev'
```

Note that when you issue a **checkout** command, it will replace the working copy with the version on the branch. For example, imagine that I had a LICENSE file on my dev branch, but not on master. If I switch to master, that file will disappear:

```
$ ls
build.gradle  fix.txt  LICENSE  pom.xml  README.txt  resources  runme.sh  src
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
$ ls
build.gradle  fix.txt  pom.xml  README.txt  resources  runme.sh  src
```

Fortunately, if there are any modifications in the working copy, git will tell you that you need to either commit the changes or stash them before switching branches:

```
$ vi LICENSE
$ git checkout master
error: Your local changes to the following files would be overwritten by checkout:
      LICENSE
Please, commit your changes or stash them before you can switch branches.
Aborting
```

Another thing to note is that if you add a file to the working set and then switch branches, that file can remain in the working set if the change is logical on the new branch (for example, adding a new file):

```
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
$ ls
build.gradle  fix.txt  pom.xml  README.txt  resources  runme.sh  src
$ vi versions.txt
$ git add versions.txt
$ git checkout dev
A     versions.txt
Switched to branch 'dev'
$ git status
On branch dev
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

      new file:   versions.txt
```

If, however, the change would not logically make sense on the new branch (such as changes to a file that doesn't exist on the new branch), git will tell you to commit or stash the change first (LICENSE exists in the dev branch but doesn't exist in master):

```
$ git checkout dev
Switched to branch 'dev'
$ vi LICENSE
```

(continues on next page)

(continued from previous page)

```
$ git add LICENSE
$ git checkout master
error: Your local changes to the following files would be overwritten by checkout:
        LICENSE
Please, commit your changes or stash them before you can switch branches.
Aborting
```

## 9.4 Merging Changes Between Branches

To merge changes from one branch to another, you use the **merge** command. For example, if I had added several files to my **dev** branch and wanted to bring them to the **master** branch, you would checkout the master branch and then run the merge command like this:

```
$ git merge dev
Updating ef7bebf..e4007d8
Fast-forward
 LICENSE             | 1 +
 README.txt          | 2 --
 fix.txt             | 1 +
 pom.xml             | 2 ++
 runme.sh            | 3 ++-
 src/Circle.groovy   | 1 +
 src/Triangle.groovy | 1 +
 versions.txt        | 1 +
 8 files changed, 9 insertions(+), 3 deletions(-)
 create mode 100644 LICENSE
 create mode 100644 src/Circle.groovy
 create mode 100644 src/Triangle.groovy
 create mode 100644 versions.txt
```

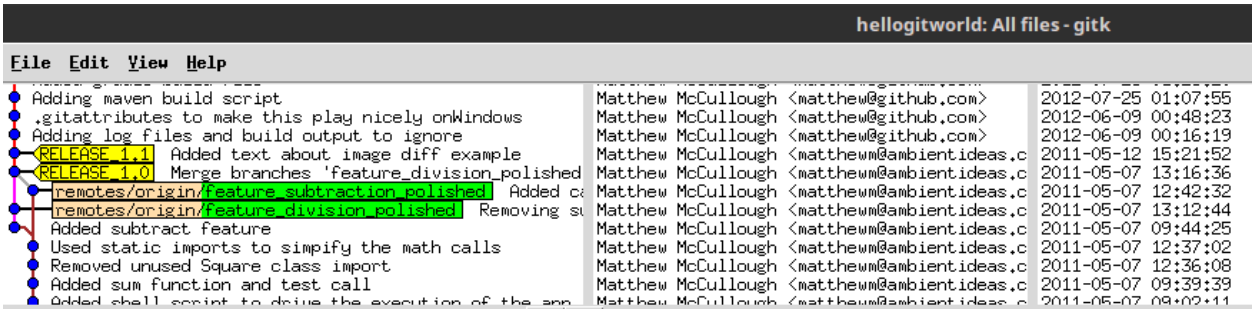
You will notice that git used a **Fast-forward** approach on this merge. If the source branch was created at an ancestral point of the target branch and there were no other changes to the target branch before the merge (in this case, I branched and then made changes to the dev branch while master sat idle), adding the changes from the source branch would effectively just be like applying the commits from the source branch to the current branch. In this case, git effectively rearranges the commits and doesn't create a merge commit. If you look at a visual representation of the two branches, it will effectively look like they are just a single series of changes and both branch pointers point to the most recent change. If a fast-forward is not possible due to intervening changes or the user requests to not use a fast-forward, the changes are effectively merged into a new commit on the target branch.

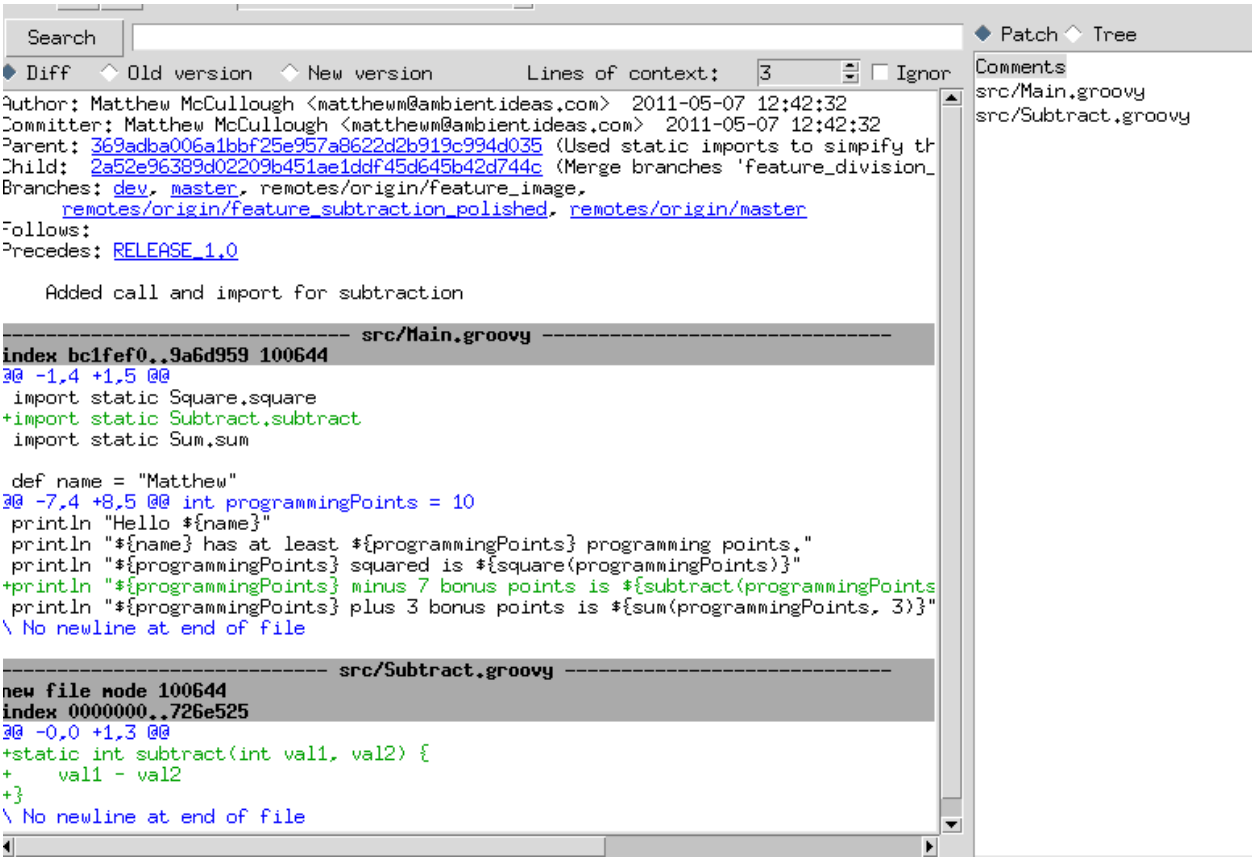
## 9.5 Visualizing Branches

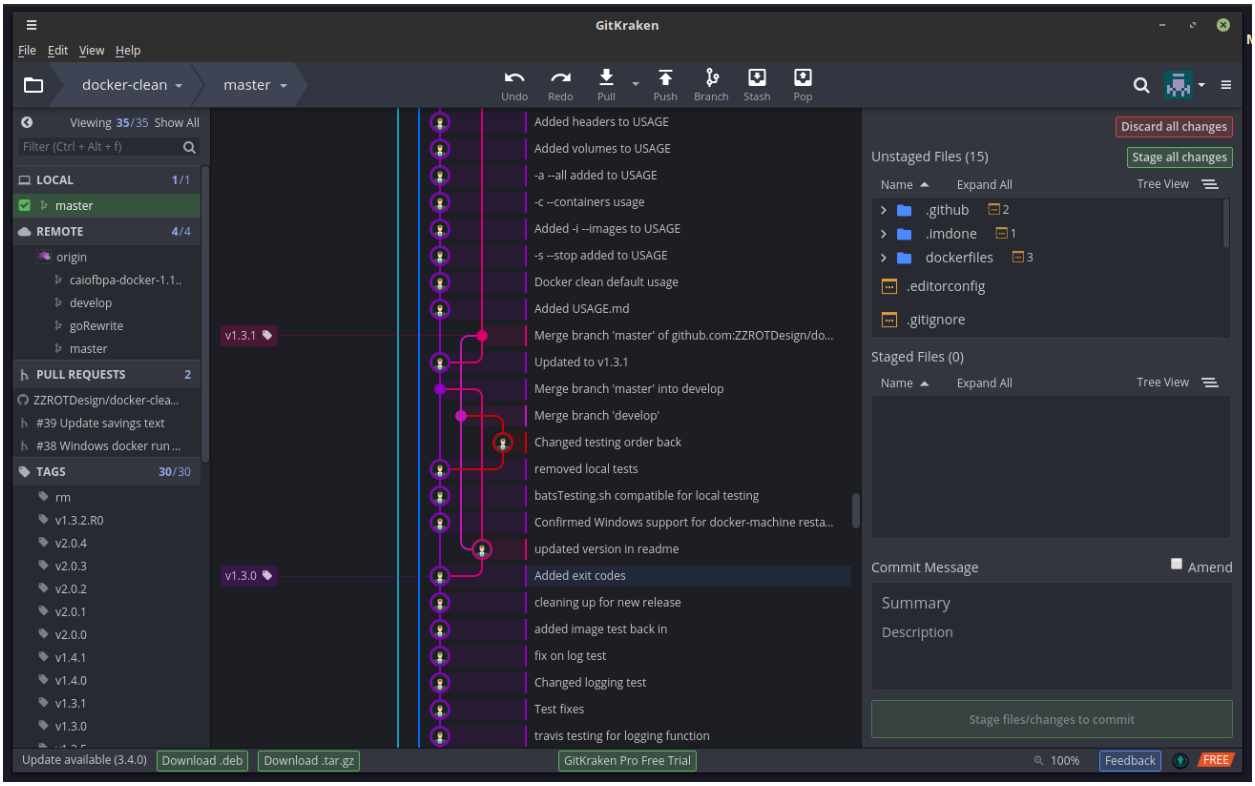
As you saw at the beginning of this section, branches can start to get a little confusing if you are looking at them in a **log** command because the commits are all one stream. Fortunately a lot of visual tools exist to help clear this up. One such tool is **gitk**, which allows you to browse the entire history of the repository. The top part of the application has three columns: the commit stream and messages, who made the commit, and the date/time of the commit.

The bottom part has the details of the commit selected in the top including what files were changed, a diff output of the changes, and other details such as what branches had changes as part of the commit.

The gitk tool is only a visualizer. Other tools let you do all the git operations. One of my favorites is GitKraken:


hellogitworld: All files - gitk
File Edit View Help
Adding maven build script
.gitattributes to make this play nicely onWindows
Adding log files and build output to ignore
RELEASE_1.1 Added text about image diff example
RELEASE_1.0 Merge branches 'feature_division_polished
remotes/origin/feature_subtraction_polished
remotes/origin/feature_division_polished
Added subtract feature
Used static imports to simpify the math calls
Removed unused Square class import
Added sum function and test call
Matthew McCullough <matthew@github.com> 2012-07-25 01:07:55
Matthew McCullough <matthew@github.com> 2012-06-09 00:48:23
Matthew McCullough <matthew@github.com> 2012-06-09 00:16:19
Matthew McCullough <matthewm@ambientideas.c 2011-05-12 15:21:52
Matthew McCullough <matthewm@ambientideas.c 2011-05-07 13:16:36
Matthew McCullough <matthewm@ambientideas.c 2011-05-07 12:42:32
Matthew McCullough <matthewm@ambientideas.c 2011-05-07 13:12:44
Matthew McCullough <matthewm@ambientideas.c 2011-05-07 09:44:25
Matthew McCullough <matthewm@ambientideas.c 2011-05-07 12:37:02
Matthew McCullough <matthewm@ambientideas.c 2011-05-07 12:36:08
Matthew McCullough <matthewm@ambientideas.c 2011-05-07 09:39:39


Search
Patch Tree
Diff Old version New version Lines of context: 3 Ignor
Comments
src/Main.groovy
src/Subtract.groovy
Author: Matthew McCullough <matthewm@ambientideas.com> 2011-05-07 12:42:32
Committer: Matthew McCullough <matthewm@ambientideas.com> 2011-05-07 12:42:32
Parent: 369adba006a1bbf25e957a8622d2b919c994d035 (Used static imports to simpify th
Child: 2a52e96389d02209b451ae1ddf45d645b42d744c (Merge branches 'feature_division_
Branches: dev, master, remotes/origin/feature_image,
remotes/origin/feature_subtraction_polished, remotes/origin/master
Follows:
Precedes: RELEASE_1.0
Added call and import for subtraction
--- src/Main.groovy ---
index bc1fef0..9a6d959 100644
@@ -1,4 +1,5 @@
import static Square.square
+import static Subtract.subtract
import static Sum.sum
def name = "Matthew"
@@ -7,4 +8,5 @@ int programmingPoints = 10
println "Hello ${name}"
println "${name} has at least ${programmingPoints} programming points."
println "${programmingPoints} squared is ${square(programmingPoints)}"
+println "${programmingPoints} minus 7 bonus points is ${subtract(programmingPoints
println "${programmingPoints} plus 3 bonus points is ${sum(programmingPoints, 3)}"
\ No newline at end of file
--- src/Subtract.groovy ---
new file mode 100644
index 0000000..726e525
@@ -0,0 +1,3 @@
+static int subtract(int val1, val2) {
+ val1 - val2
+}
\ No newline at end of file


GitKraken
File Edit View Help
docker-clean
master
Undo
Redo
Pull
Push
Branch
Stash
Pop
Viewing 35/35 Show All
Filter (Ctrl + Alt + f)
LOCAL 1/1
master
REMOTE 4/4
origin
caiofbpa-docker-1.1..
develop
goRewrite
master
PULL REQUESTS 2
ZZROTDesign/docker-clea...
#39 Update savings text
#38 Windows docker run ...
TAGS 30/30
rm
v1.3.2.R0
v2.0.4
v2.0.3
v2.0.2
v2.0.1
v2.0.0
v1.4.1
v1.4.0
v1.3.1
v1.3.0
Added headers to USAGE
Added volumes to USAGE
-a --all added to USAGE
-c --containers usage
Added -i --images to USAGE
-s --stop added to USAGE
Docker clean default usage
Added USAGE.md
v1.3.1
Merge branch 'master' of github.com:ZZROTDesign/do...
Updated to v1.3.1
Merge branch 'master' into develop
Merge branch 'develop'
Changed testing order back
removed local tests
batsTesting.sh compatible for local testing
Confirmed Windows support for docker-machine resta...
updated version in readme
v1.3.0
Added exit codes
cleaning up for new release
added image test back in
fix on log test
Changed logging test
Test fixes
travis testing for logging function
Discard all changes
Unstaged Files (15)
Stage all changes
Name
Expand All
Tree View
.github 2
.imdone 1
dockerfiles 3
.editorconfig
.gitignore
Staged Files (0)
Name
Expand All
Tree View
Commit Message
Amend
Summary
Description
Stage files/changes to commit
Update available (3.4.0)
Download .deb
Download .tar.gz
GitKraken Pro Free Trial
100%
Feedback
FREE

# CHAPTER 10

## Using Tags

If you have used CVS or SVN in the past, you are probably familiar with tags already. A tag in SVN, however, is more like a branch (they are logically distinct but function the same way - you can make commits on a tag). Tags in git work like they do in CVS; they are just an alias for a commit id.

Each commit in the commit stream has a long identifier associated with it. Git will let you get away with only typing part of the identifier if it can uniquely identify the commit:

```
$ git show d2280d
commit d2280d000c84f1e595e4dec435ae6c1e6c245367
Author: Matthew McCullough <matthew@github.com>
Date:   Tue Jul 24 22:07:55 2012 -0700

    Adding maven build script

diff --git a/.gitignore b/.gitignore
index 6a03ed1..7c82083 100644
--- a/.gitignore
+++ b/.gitignore
@@ -2,3 +2,4 @@
 *.log
 output/
 build/
+target/
...
```

However, it can be useful to apply more meaningful names to certain commits, such as release numbers. To do this, you use the **tag** command:

```
$ git tag v1.0 37dd
```

You can then use this tag anywhere that a commit id is normally expected. For example, you could compare the repository at the v1.0 commit against the special HEAD tag that always points to the last commit in the repository:

```
$ git diff v1.0 HEAD | head
diff --git a/.travis.yml b/.travis.yml
new file mode 100644
index 0000000..17eccd6
--- /dev/null
+++ b/.travis.yml
@@ -0,0 +1,2 @@
+sudo: false
+language: java
diff --git a/LICENSE b/LICENSE
new file mode 100644
...
```

CHAPTER 11

# Reverting Changes

Every now and then, you are going to make mistakes and need to undo some changes. To do this, you use the **revert** command. You will be prompted for a commit message and the actual revert will be added as a new commit (the old commit will still remain):

```
$ git revert bc055
```

You can revert changes from any previous commit in the repository. Should a straight reversal of the change not be possible due to a conflict (other changes have occurred that overlap), you may have to manually perform a merge of the revert. If you request to revert more than one commit, git will process the reverts in order, and should a conflict arise, it will stop. You can resume the revert by adding the **–continue** switch:

```
$ git revert --continue
```

If you are in the middle of applying several reverts and decide to abort, you can simply issue a **revert** command with the **–abort** switch if you want to discard all changes (remove any commits that were added) or with the **–quit** switch if you want to just stop and preserve any commits that already have been added.

# CHAPTER 12

# Using Remote Repositories

Up until this point, all the commands that we have talked about could be done just on the local machine. Git, however, is meant to interact with remote repositories as well[1]. Usually teams have a central remote repository that they copy to their local and then merge changes back to when they are done.

## 12.1 Cloning a Remote Repository

To start working with a remote repository, you can use the **clone** command. You can clone any repository, even one that is located on the same machine. You just specify the path or URL of the respository to clone:

```
$ git clone /home/tellmejeff/work/newproject newprojectclone
Cloning into 'newprojectclone'...
done.
$ git clone https://github.com/tellmejeff/mesos-marathon
Cloning into 'mesos-marathon'...
remote: Counting objects: 13, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 13 (delta 1), reused 10 (delta 1), pack-reused 0
Unpacking objects: 100% (13/13), done.
Checking connectivity... done.
```

Now that the repository is cloned, you can start using it like any regular local repository. You can add new files or update existing ones and then commit the changes. The changes won't be applied to the remote repository until you do a push (covered later).

## 12.2 Connecting an Existing Repository to a Remote Repository

There are a few different scenarios in which you may want to link your local repository to a remote repository. Imagine first that you and a friend want to work on code together, and he has already created a branch on a remote server. You

[1] Technically the term remote is a misnomer. Any repository you synchronize with doesn't have to be at a different location, it could even be on the same machine. What is really meant by the term remote is a completely separate repository that has its own commit stream.

want to create a branch in your local that tracks changes on the remote (you can pull any changes there down to your local). First, you need to tell git about this remote server using the **remote** command (origin is the default name for a remote, but you can use any name you like):

```
$ git remote add origin https://github.com/tellmejeff/newproject.git
```

Now you need to execute a **fetch** command. This will update the remote tracking branches of the remote repository into your repository in a safe way that won't affect your local branches (if you don't specify a name after the **fetch**, the **origin** remote is used):

```
git fetch
From https://github.com/tellmejeff/newproject
* [new branch]      dev        -> origin/dev
* [new branch]      master     -> origin/master
```

If you execute the **branch** command with the **-r** switch, you will now see these remote branches:

```
$ git branch -r
origin/dev
origin/master
```

You now have a couple of options of how you make a local branch that is connected to the remote branch. You can checkout a new branch based on the remote branch by using the **checkout** command with the **–track** and **-b** switch:

```
$ git checkout --track -b dev origin/dev
Branch dev set up to track remote branch dev from origin.
Switched to a new branch 'dev'
```

You can also reuse an existing branch by switching to that branch and then merging the remote branch:

```
$ git checkout master
$ git merge origin/master
```

You can also explictly set the upstream branch for a branch:

```
$ git branch --set-upstream-to=origin/master master
Branch master set up to track remote branch master from origin.
```

## 12.3 Updating From a Remote Repository

In CVS or SVN, when you want to get the latest changes from the repository, you just execute an **update** command. Git has similar commands, but because git is a distributed repository, it splits the update into two commands, **fetch** and **pull**. If you want to get the latest changes but you aren't sure whether it will disrupt something in your local repository, you can use **fetch** to pull it down to a separate branch. I did this in the previous section when we were connecting the local repository to a remote one. You can then use **merge** to bring those changes over from the remote branch to your local branch.

If you just want to bring all the changes down like a normal update would, you use the **pull** command. If you haven't made any changes since the last time you pulled from the remote repository, the commits pulled down are simply added on to the end of your commit stream in your local repository. If, however, you have made your own commits that aren't in the remote repository, the pull command will attempt to merge the changes in and then you will have to commit the merge if there is any overlap in the files from your commits and the updates pulled down[2].

[2] Because your commits are already in your repository, adding the commits from the remote repository wouldn't make sense. Instead, they'll just have to be added in as a new commit. However, the log message that is saved with the merge will reference the commits from the remote

## 12.4 Rebasing

Merge commits can be problematic[3], so if you do have changes that overlap, you may want do what is called a **rebase**. Effectively, you want to reconstruct the changes that you made in your local repository, but you want to apply them after the changes that were made in the remote repository. Git will unwind your commits (roll back to the point where you last pulled from the remote repository), apply all of the commits from the remote repository, and then start replaying your commits afterwards. You do this by adding the **–rebase** switch to the pull command:

```
$ git pull --rebase
```

If there are any conflicts that git can't resolve when replaying your commits, it will stop so that you can manually fix the conflict. You then can resume the rebase using the **rebase** command with a **–continue** switch:

```
$ git rebase --continue
```

You can also skip over a commit that has a conflict by using the **rebase** command with a **–skip** switch:

```
$ git rebase --skip
```

If you want to abort the rebase and return the repository back to the original state, you use the **rebase** command with an **–abort** switch:

```
$ git rebase --abort
```

repository so that there is still a reference to them should someone want to understand the history of how these changes came to be. Similarly, because your commits diverged from the remote repository, when you go to push, you'll end up pushing a merge commit to the remote. This is problematic, however, because you could be collapsing several commits into one, and if someone else pulls your changes down and has a conflict that git can't automatically resolve, they'll have to manually merge the changes in, and that can be quite disruptive. See the *Rebasing* section for details about how to avoid this issue.

[3] A merge commit is when you pulled some changes down from the remote repository but they weren't able to just be applied in a fast-forward merge, so you had to manually merge the changes and then commit them as a new commit. This leads to the problem mentioned in the previous footnote where pushing the merge and your additional changes as well can be problematic.

CHAPTER 13

# Git Cheat Sheet

## 13.1 Configuration

Configure user information for all local repositories.

```
$ git config --global user.name "[name]"
Sets the name you want atached to your commit transactions
$ git config --global user.email "[email address]"
Sets the email you want atached to your commit transactions
$ git config --global color.ui auto
Enables helpful colorization of command line output
```

## 13.2 Create Repositories

Start a new repository or obtain one from an existing URL.

```
$ git init [project-name]
Creates a new local repository with the specified name
$ git clone [url]
Downloads a project and its entire version history
```

## 13.3 Make Changes

Review edits and craf a commit transaction.

```
$ git status
Lists all new or modified files to be commited
$ git add [file]
Snapshots the file in preparation for versioning
```

(continues on next page)

(continued from previous page)

```
$ git reset [file]
Unstages the file, but preserve its contents
$ git diff
Shows file differences not yet staged
$ git diff --staged
Shows file differences between staging and the last file version
$ git commit -m "[descriptive message]"
Records file snapshots permanently in version history
```

## 13.4 Branches

Name a series of commits and combine completed efforts.

```
$ git branch
Lists all local branches in the current repository
$ git branch [branch-name]
Creates a new branch
$ git checkout [branch-name]
Switches to the specified branch and updates the working directory
$ git merge [branch]
Combines the specified branch's history into the current branch
$ git branch -d [branch-name]
Deletes the specified branch
```

## 13.5 Refactor Filenames

Relocate and remove versioned files.

```
$ git rm [file]
Deletes the file from the working directory and stages the deletion
$ git rm --cached [file]
Removes the file from version control but preserves the file locally
$ git mv [file-original] [file-renamed]
Changes the file name and prepares it for commit
```

## 13.6 Suppress Tracking

Exclude temporary files and paths. The .gitignore file suppresses accidental versioning of files and paths matching the specified patterns.

```
$ git ls-files --other --ignored --exclude-standard
Lists all ignored files in this project
```

## 13.7 Save Fragments

Shelve and restore incomplete changes.

```
$ git stash
Temporarily stores all modified tracked files
$ git stash list
Lists all stashed changesets
$ git stash pop
Restores the most recently stashed files
$ git stash drop
Discards the most recently stashed changeset
```

## 13.8 Review History

Browse and inspect the evolution of project files.

```
$ git log
Lists version history for the current branch
$ git log --follow [file]
Lists version history for a file, including renames
$ git diff [first-branch]...[second-branch]
Shows content differences between two branches
$ git show [commit]
Outputs metadata and content changes of the specified commit
```

## 13.9 Redo Commits

Erase mistakes and craft replacement history.

```
$ git reset [commit]
Undoes all commits afer [commit], preserving changes locally
$ git reset --hard [commit]
Discards all history and changes back to the specified commit
```

## 13.10 Synchornizing Changes

Register a repository bookmark and exchange version history.

```
$ git fetch [bookmark]
Downloads all history from the repository bookmark
$ git merge [bookmark]/[branch]
Combines bookmark's branch into current local branch
$ git push [alias] [branch]
Uploads all local branch commits to GitHub
$ git pull
Downloads bookmark history and incorporates changes
```