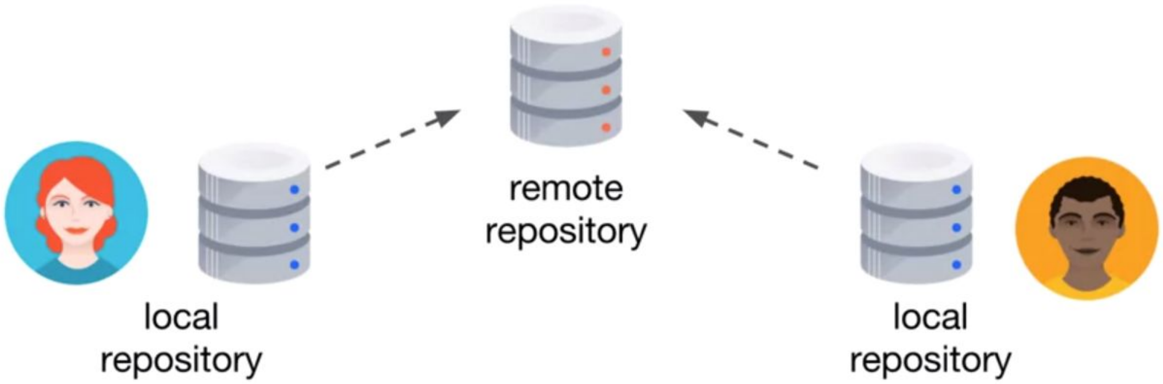
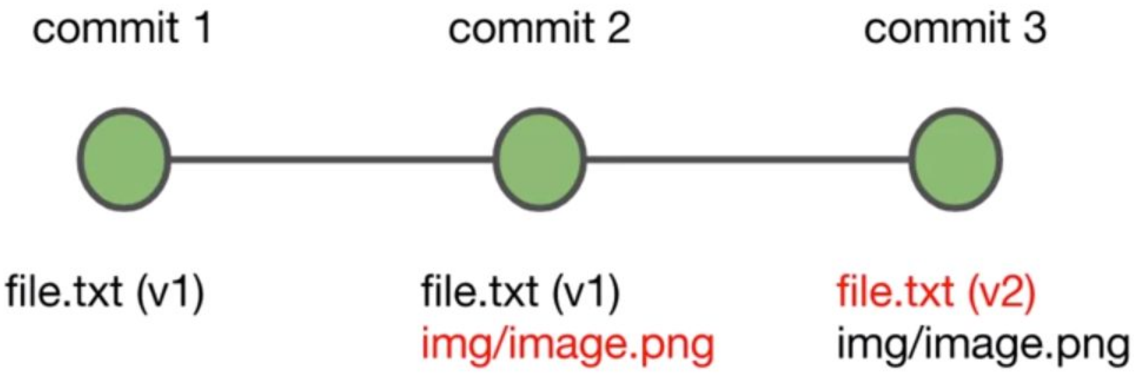
Git Overview

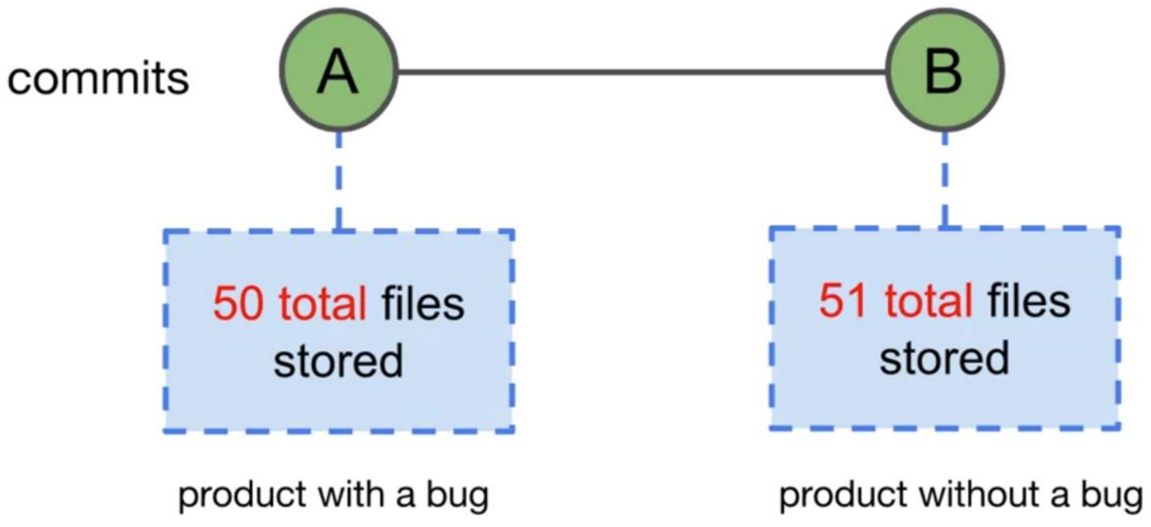
Git is a Distributed Version Control System.



Each commit is a snapshot of the entire project at a given point in time.



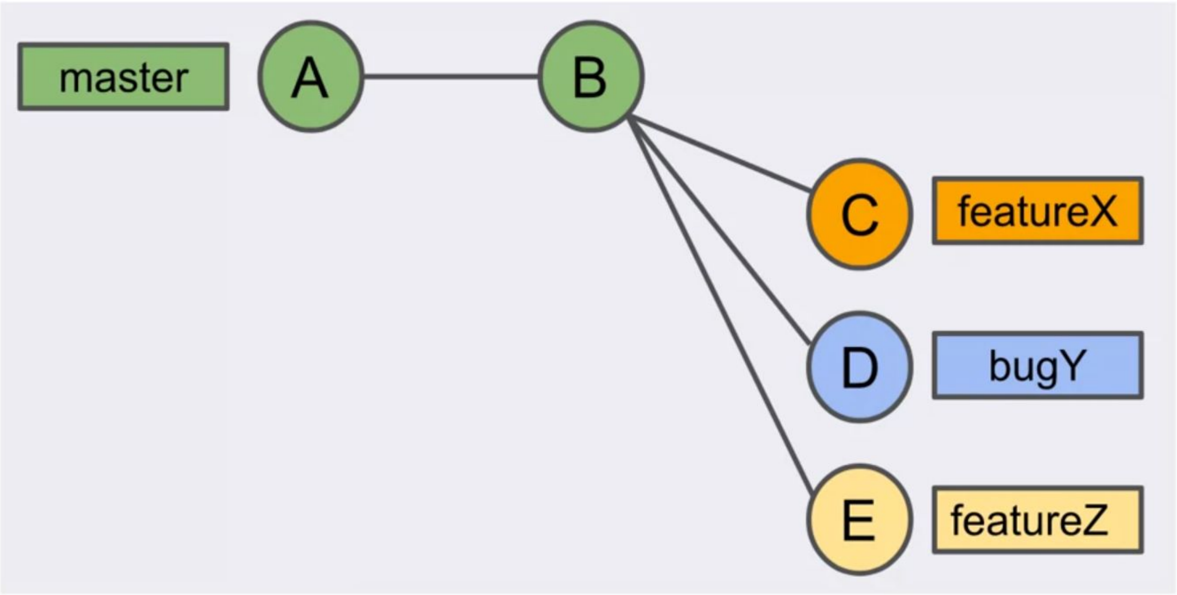
With many small changes to the project, you might think that Git stores many copies of the same file. This is not how Git works however. Behind the scenes, Git is very efficient at storing commits. Each unique file is stored only once.



All commits belong to a branch.

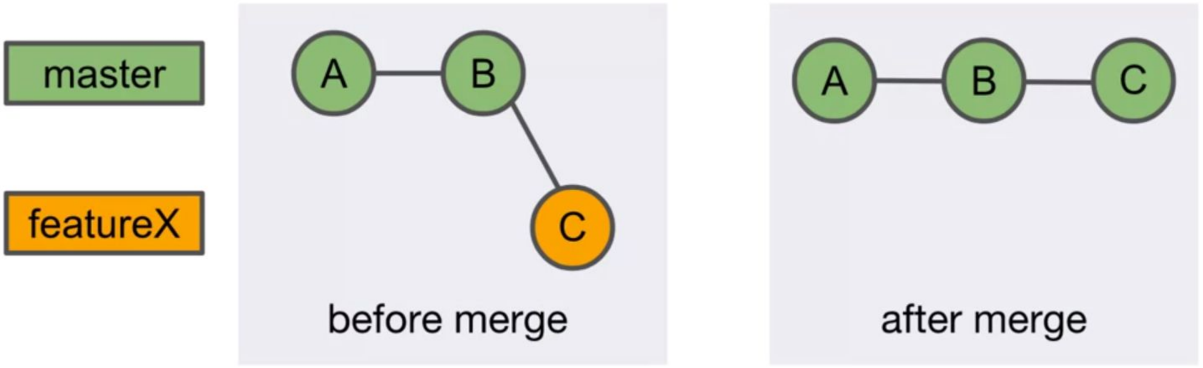
A branch can be thought of as an independent line of development of the project. By default, there is a single branch and it's called master.

How do you maintain a stable project at the same time that you are working on it? You can create a separate branch and work on it independently of the master branch.



So… Git can manage the many versions of projects with commits and that the project can be worked on independently using branches.

When a branch is ready to become part of the master branch, it can be merged into the master branch. A merge combines the work of separate branches.



Before the merge, the master branch has no knowledge of the featureX branch. After the merge, there's a single master branch with the latest commit, including the code that implements featureX.

Before you merge content into the master branch, how do you know that your changes are good?

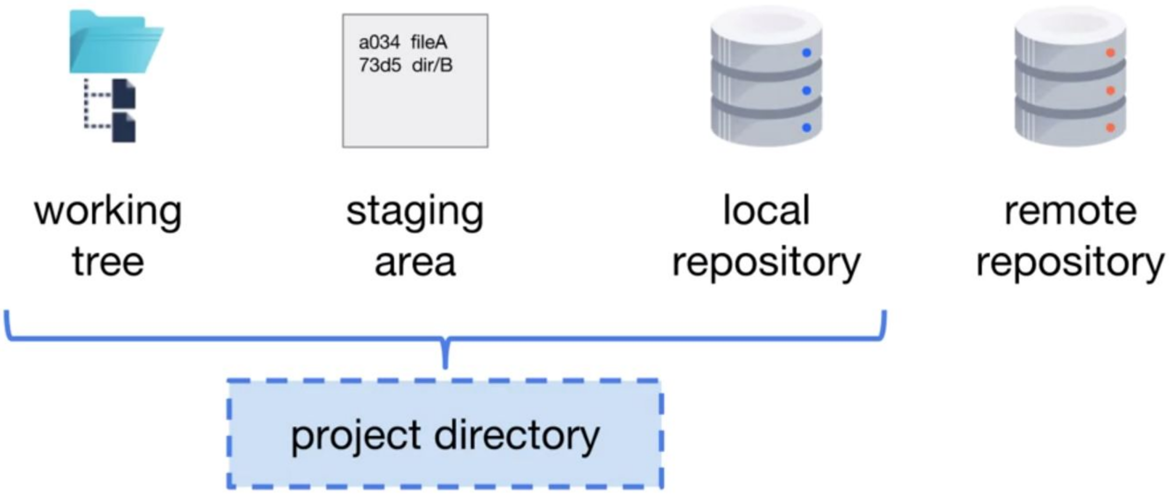
A pull request is a request to merge your branch into another branch.

This request is usually made by developer of the branch when the feature, bug fix or other change is complete. In this example, the pull request is a request to merge the commit that includes featureX into the master branch. During a pull request team members can discuss, review, and approve your changes.

You can also require that automated test pass before the merge is allowed to happen. This helps ensure that the changes introduced by the merge don't cause problems for the customer.

If the pull request is accepted, your version of the project is merged and becomes the latest commit on the master branch.

Locations of Git



Working Tree

The working tree is the location on your computer that contains the directories and files of a single commit. This is where you can view and edit the files of the project, preparing them for the next commit.

Staging Area

The staging area contains a list of files that are planned to be included in the next commit that you make. You prepare the staging area just the way that you want it, so that the next commit is a meaningful snapshot of the project. We use the add command to add new or modified files to the staging area.

Local Repository

The local repository contains all of the commits that have been made for the project. These commits represent the version history of the project.

Project Directory

The working tree, staging area, and local repository are commonly all contained in a single directory on your local computer. This is called the project directory.
The project directory contains the working tree. The working tree contains the directories and files of a single commit or snapshot of your project. You can view and edit these files to prepare

them for the next commit. The project directory also contains a hidden directory named.git. This is where the staging area and local repository are located.

Remote Repository

The remote repository contains the commits of the project, and is often considered the source of truth or official state of the project. When the local and remote repositories are synchronized, they contain exactly the same commits. The remote repository is usually located in a data center or in the cloud.

In Summary

The working tree contains the project files for a single commit, the staging area holds a list of files that will be included in the next commit, the local repository contains all of the commits of the project. On your local computer, you have a project directory that contains the working tree as well as a hidden.git directory. The staging area and local repository are located in this directory. The remote repository contains the commits of the project on a remote computer.

Lab

Create a Local Repository

```
saroar:~$ mkdir workspace
saroar:~$ cd workspace

saroar:~/workspace$ mkdir sample-project
saroar:~/workspace$ cd sample-project

saroar:~/workspace/sample-project$ git init
Initialized empty Git repository in
/home/saroar/workspace/sample-project/.git/
```

Commit to a Local Repository

View file status using git status

Use git status to view the status of files in the working tree and staging area.

If you tried to create a commit right now, it would not work because you haven't changed anything since the previous commit.

```
saroar:~/workspace/sample-project$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Let's add a file named fileA.txt to your working tree.

```
saroar:~/workspace/sample-project$ touch fileA.txt
saroar:~/workspace/sample-project$ git status
On branch master

No commits yet

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

      fileA.txt

nothing added to commit but untracked files present (use "git add" to
track)
```

Git status shows the file as untracked. This file is only in the working tree and is not managed by git. Notice that git gives helpful messages on what you would normally do next. Here, git is suggesting that you use the git add command to add the file to the staging area.

Stage content using git add

Use git add to add content to the staging area. Staged content is part of the next commit.

Commit content using git commit

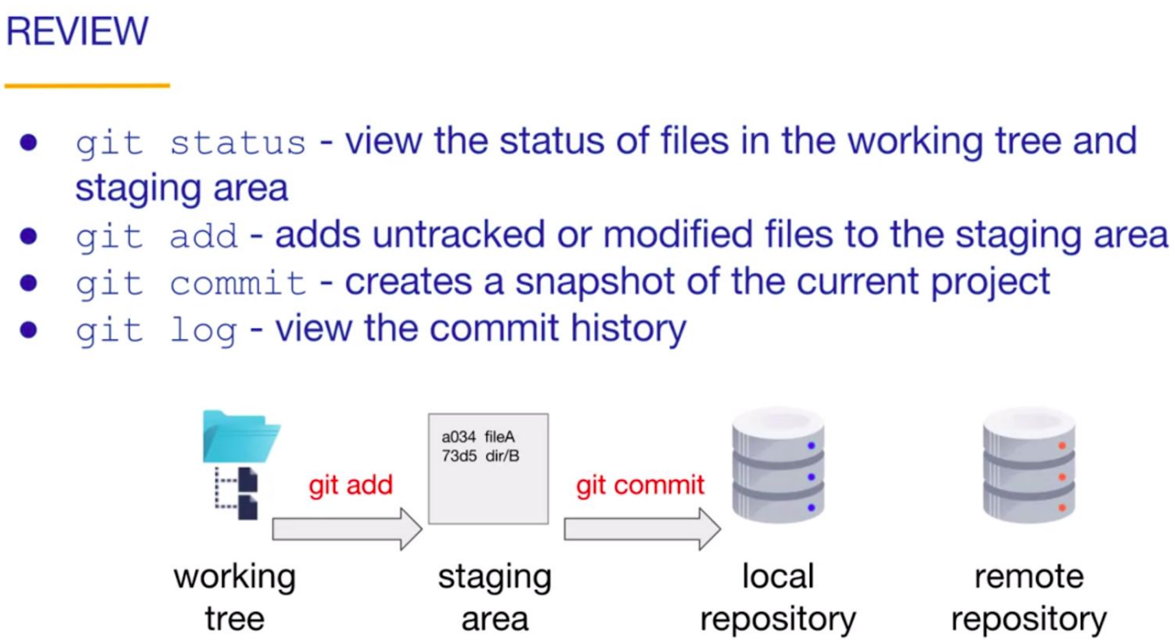
Use the git commit command to add staged content to the local repository as a commit. Once you have committed a file, it will remain in the staging area and in all commits, unless you specifically remove it.

Viewing the commit history with git log

You can view the local repository's commit history using git log.
Add the --oneline option to git log to see a condensed version of the log.

```
saroar:~/workspace/sample-project$ git add fileA.txt
saroar:~/workspace/sample-project$ git commit -m "add fileA.txt"
[master (root-commit) fa35b4b] add fileA.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fileA.txt
saroar:~/workspace/sample-project$ git log
commit fa35b4b917b847fcbc729097cf0529db6175e1a9 (HEAD -> master)
Author: Sk Golam Saroar <emailsaroar@gmail.com>
Date:   Wed Sep 30 00:04:36 2020 +0600

    add fileA.txt
```



REVIEW

- `git status` - view the status of files in the working tree and staging area
- `git add` - adds untracked or modified files to the staging area
- `git commit` - creates a snapshot of the current project
- `git log` - view the commit history

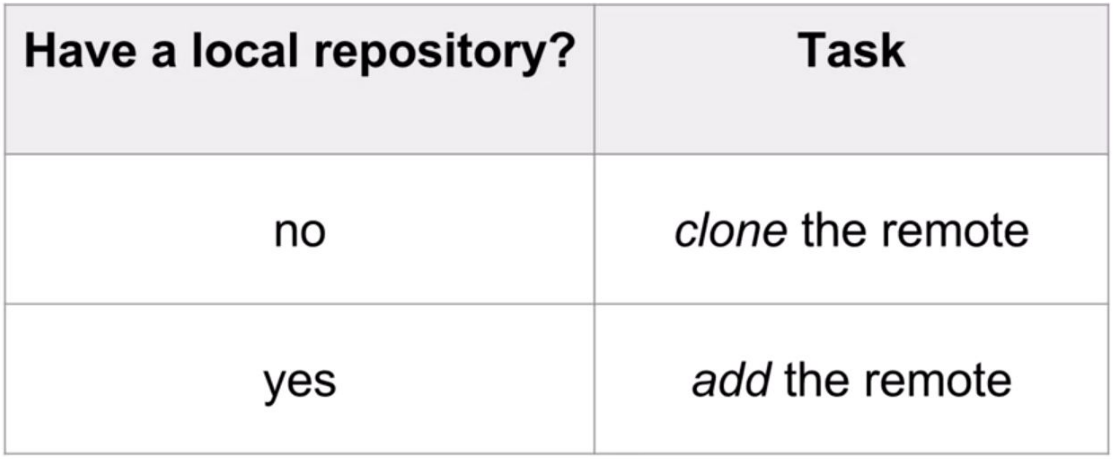
Create a Remote Repository

A remote repository is often a bare repository. Because nobody works with the repository locally, there is usually no working tree or staging area on a remote repository. The root directory of a remote repository is similar to the ".git" directory in a local repository. By convention, remote repository names end with ".git".

Push to a Remote Repository

git clone vs git remote add

There are two fundamental ways to start working with a remote repository, it depends on if you already have a local repository, in other words, if depends on if you already have some work done via commits in a local repository that you want to push to a remote repository.



Have a local repository?	Task
no	*clone* the remote
yes	*add* the remote

If you do not have an existing local repository, then you will clone the remote repository, creating a local repository that is associated with the remote repository. If you already have a local repository with commits that you want to push to a remote repository, then you will add the remote repository to your local repository.

Clone a remote repository

Git clone is a command used to create a local copy of a remote repository. Once you have cloned a repository, you can work with the local repository optionally pushing your commits to and pulling new commits from the remote repository. A reference to the remote repository is included in the local repository. This allows you to synchronize the repositories.

After performing a git clone, information on the remote repository is always available using the git remote command.
The git remote command displays information about remote repositories associated with the local repository.

Add a remote repository to a local repository

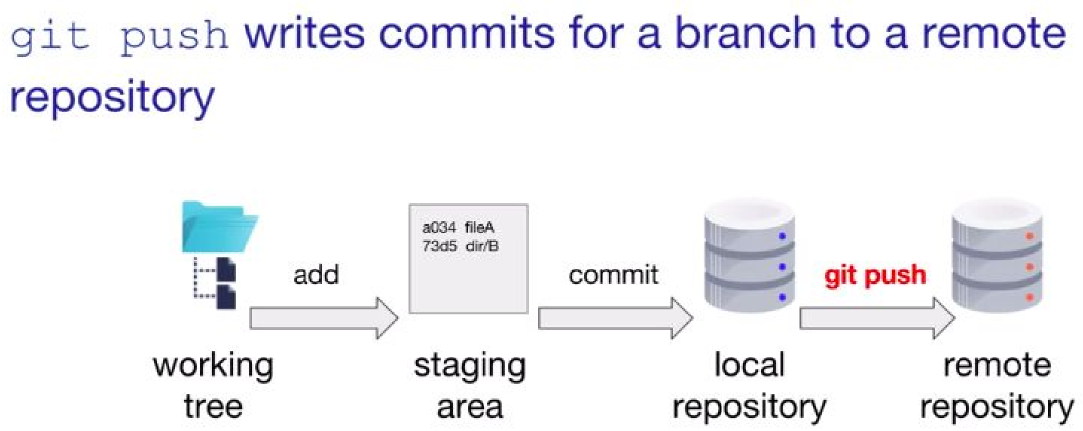
If you already have a local repository with commits that you want to push to a remote repository, you can use the git remote add command. This command add's information about the remote repository to the local repository.

Then from our existing local repository, we execute the git remote add command.

Push commits to a remote repository

All commits belong to a branch, a branch can be thought of as an independent line of development of the project. So far we have not created any branches. So all of our commits have been on a default master branch.

Git push writes commits for a branch from the local repository to the remote repository. A successful push synchronizes the branches on the local and remote repositories so that they contain exactly the same commits. Pushing to the remote repository is primarily done to share your work with the team, but it also serves as a good back up of the local repository.



`git push` writes commits for a branch to a remote repository

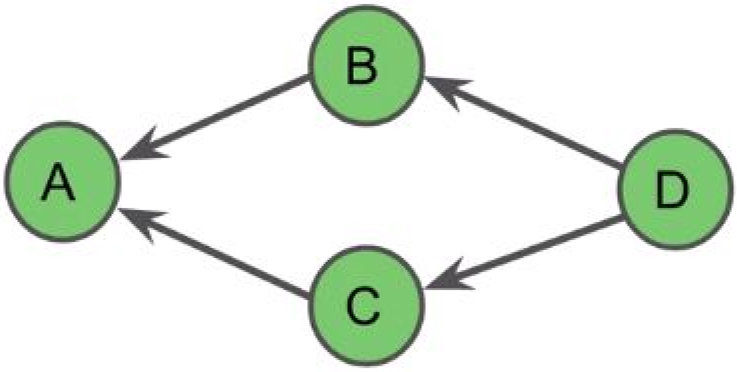
You execute the git push command to push commits from the local repository to the remote repository. The first time you push, you should pass the repository shortcut name or Url. The shortcut name is often origin.

You should also include the branch name that you would like to push. The set upstream, or -u option, is used to set up a tracking relationship between your local branch and the corresponding remote branch. Git can then inform you when the branches are out of synch.

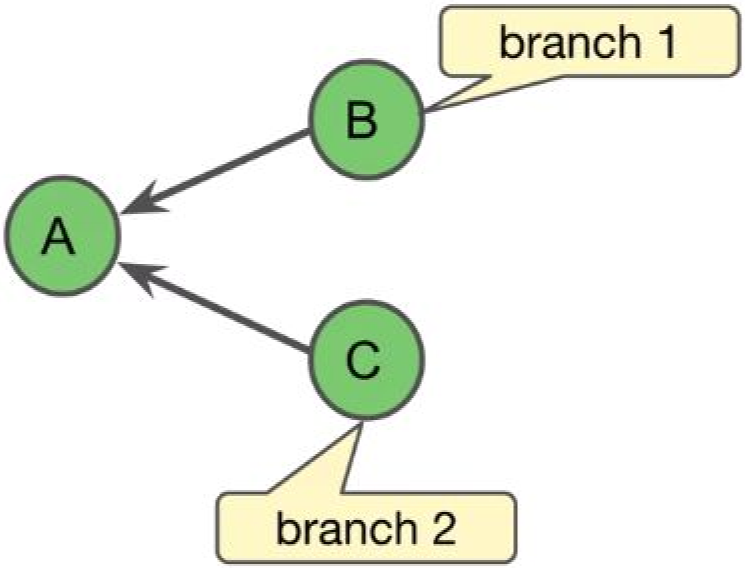
The values after git push are all optional, because git will assume default information or use previous values after you've executed the first push.

Git’s Graph Model

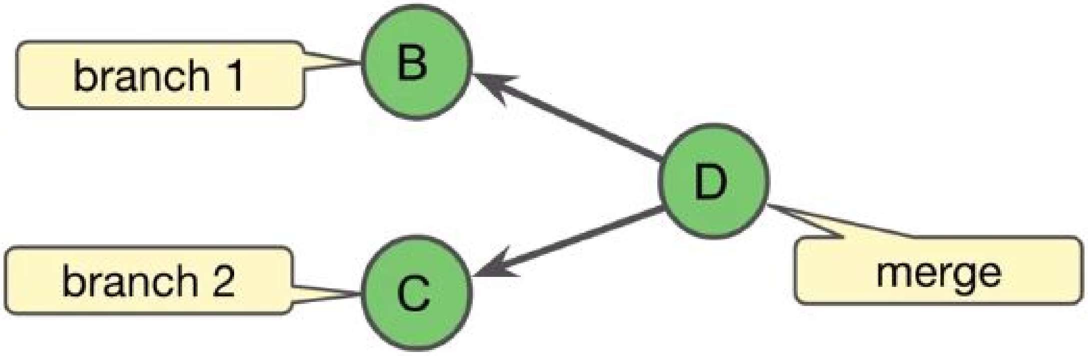
Git models the relationship of commits with the directed acyclic graph. The entire graph contains a project's history. Each node in Git represents a commit. The arrows point at a commit's parents.



In this example, you can see that commit D has two parents, commit B and commit C. The commits and the relationship between them is what forms the project's history.

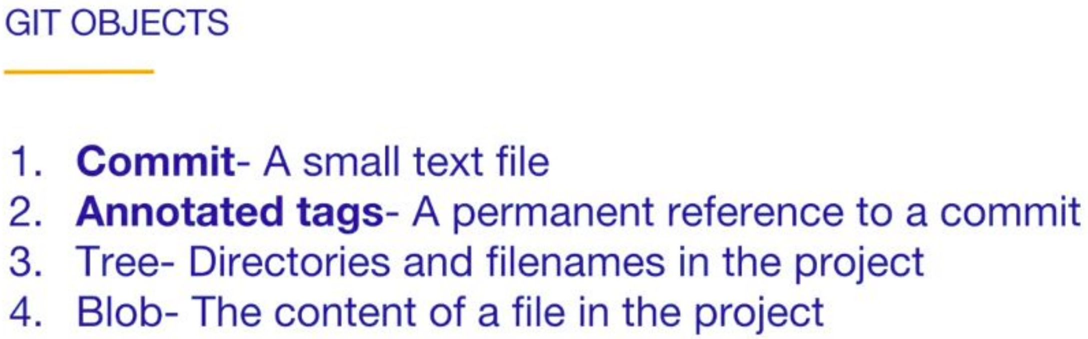


A branch occurs if a commit has more than one child. This graph contains a branch because commit A has two children, commit B and commit C.



A merge occurs when a commit has more than one parent. In this example, commit D is a merge of branch one and branch two.

Git Objects



GIT OBJECTS

1. **Commit**- A small text file
2. **Annotated tags**- A permanent reference to a commit
3. Tree- Directories and filenames in the project
4. Blob- The content of a file in the project

Internally, Git uses objects to store four types of things. A commit object is a simple text file that contains information such as the commit user information, commit message, a reference to the commit's parent or parents, and a reference to the root tree of the project. That information is all that Git needs to rebuild the full contents of a commit. An annotated tag is a reference to a specific commit. A tree is an object that contains a list of the filenames and directories inside of a directory. A blob is an object that stores the content of a file that is being managed by Git. A typical Git user may only interact with commit objects and tags, letting Git worry about the details related to trees and blobs.

Git ID

A Git ID is the name of a Git object. All of the objects stored by Git are named with a 40-character hexadecimal string. These strings are commonly known as Git IDs, but they are also known as object IDs, SHA-1s, hashes and checksums. You commonly see these Git IDs as you work with Git. For example, the `git log` command will show 40-character commit IDs like what we see here. That string is the name of a commit object.

```
saroar:~/workspace/sample-project$ git log
commit fa35b4b917b847fcbc729097cf0529db6175e1a9 (HEAD -> master)
Author: Sk Golam Saroar <emailsaroar@gmail.com>
Date:   Wed Sep 30 00:04:36 2020 +0600

    add fileA.txt
```

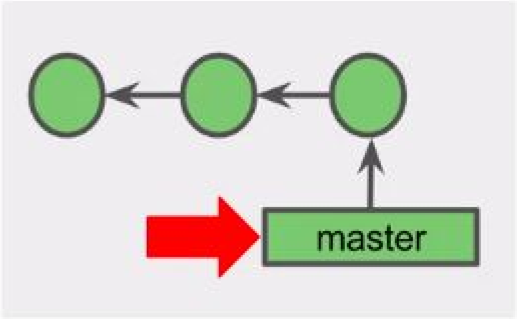
Since the 40-character Git ID names are not very people friendly, Git sometimes shortens them to the first seven characters. Here you can see that the Git log command with the one line option returns a shortened Git ID for the commit object. If we execute the Git log command with no options, you can see that the full 40-character version of the commit object name is used.

Git References

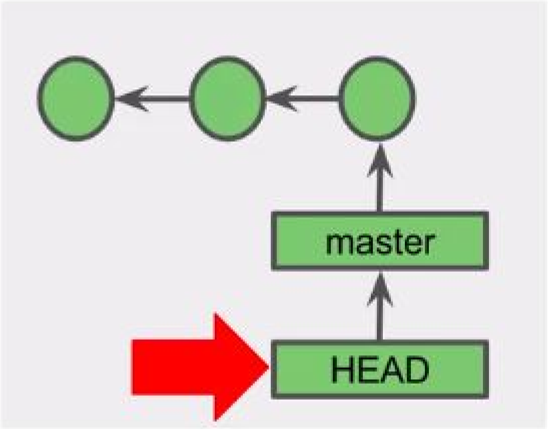
Commits can be associated with references. A reference is a user-friendly name that points to a commit's SHA-1 value or another reference.

Branch labels and HEAD

Master is the default name of the main branch in the repository. A branch is an independent path for a set of commits. If we never create a branch in a repository, then by default, all commits are part of the master branch. Here, we execute the git status command. And Git informs us that we are currently on the master branch. Any commits that we would make at this point would be to this branch.



A branch label points to the most recent commit in the branch. That commit is commonly called the tip of the branch. Branch labels are implemented as references. In this example there are three commits on the master branch, with the master branch label pointing to the tip of the branch. Note the subtle difference between a branch and a branch label. All three commits belong to the master branch, even though the master branch label is only at the tip of the branch.



HEAD is a reference to the current commit.
It usually points to the branch label of the current branch.
Because it only points to the current commit and there can only be one current commit, there is only one HEAD reference in your local repository.
In this example we have three commits on the master branch.
The master branch label reference points to the most recent commit.
The HEAD reference points to the master branch label.

~ and ^ characters

In git commands, the ~ character can be appended to git IDs and references to refer to prior commits.

The caret character can be appended to IDs and references in git commands, primarily to refer to a specific parent in a merge commit.

Tags

A tag is a reference attached to a specific commit. It acts as a user-friendly label for the commit. There are two types of tags.
The first type is a lightweight tag. This is implemented as a simple reference, much like a branch label or HEAD. It refers to the commit object.
The second type of tag is an annotated tag. This is one of the four Git object types mentioned earlier. An annotated tag is similar to a commit object in that it includes metadata such as tag author information, tag date, tag message, and the ID of the commit object referenced by the tag.

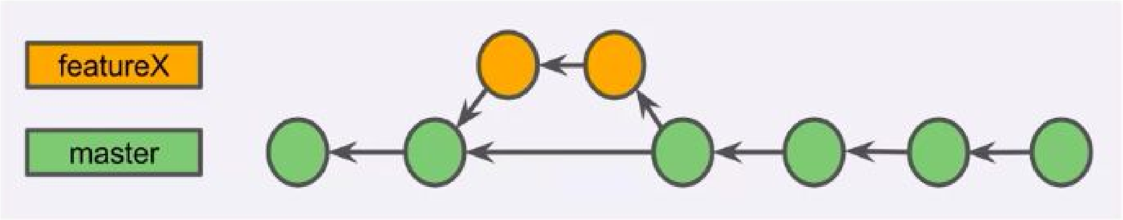
To view all the tags in the repository, use the git tag command.
To tag a commit with a lightweight tag, use the git tag command followed with an argument specifying the tag name. You can optionally specify a commit to be tagged. If you don't specify a commit, the commit that HEAD points to will be tagged.
To tag a commit with an annotated tag, you also use the git tag command, but specify the -a option. This will create a Git object. You must specify a tag message.

The git push command alone does not automatically transfer tags to the remote repository. To transfer a single tag, use the git push command and specify the remote name, such as origin, as well as the tag name. To transfer all of your tags, you can execute git push remote, followed by the tags option.

Git Branches

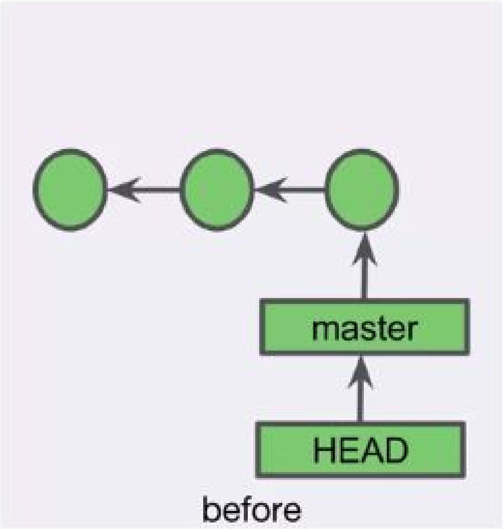
All commits of a project belong to a branch. By default, commit belong to the master branch. A branch is a set of commits starting with the most recent commit in the branch and tracing back to the project's first commit.

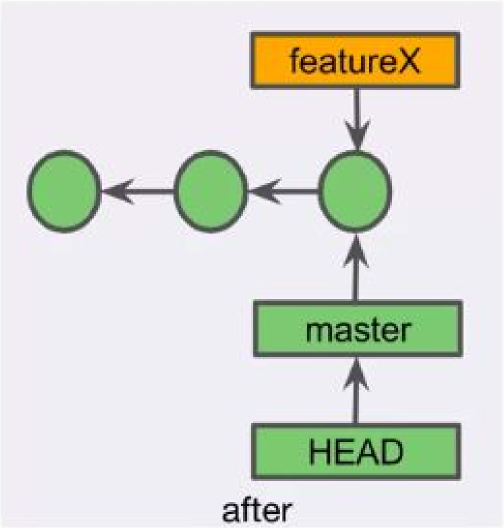
creating a branch is fast and easy. Creating a branch only creates one tiny file, a reference. Branches enable experimentation. Team members can isolate their work so that it doesn't impact others until the work is ready. Branches are not aware of other branches. This allows you to experiment with changes to the project, while at the same time the team retains a stable version of the project. If you have an idea for a change, you can create a branch and test your idea. Later you can throw out your branch, or merge it into the official project. Branches enable multiple team members to concurrently work on the project, without stepping on each other's work. Merging the work together later usually is not too difficult. Branches allow you to support multiple versions of the project simultaneously. For example, if the project is a software project with customers on several supported versions, you can update each version if an important hot fix is necessary.

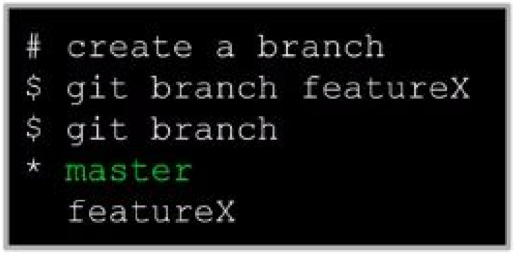


Branches can be short lived or long running. Short lived branches are commonly called topic or feature branches, and usually contain one small change to the project. For example, a topic branch may contain a new feature, a bug fix, a hot fix, a configuration change, or any other change that the project requires. When it's ready, a topic branch is merged into a long running branch. In this example, the feature X branch is a topic branch. Two commits were needed to create the feature, and then the feature was merged into the master branch. Long running branches like the master branch live longer than topic branches, and can even live for the life of the project. Examples of branches that might live for the life of a project are a master branch or a develop branch.

Creating a branch





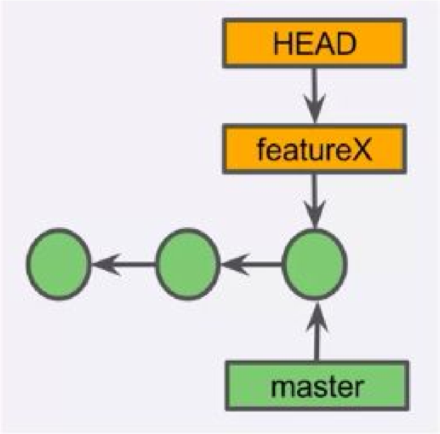


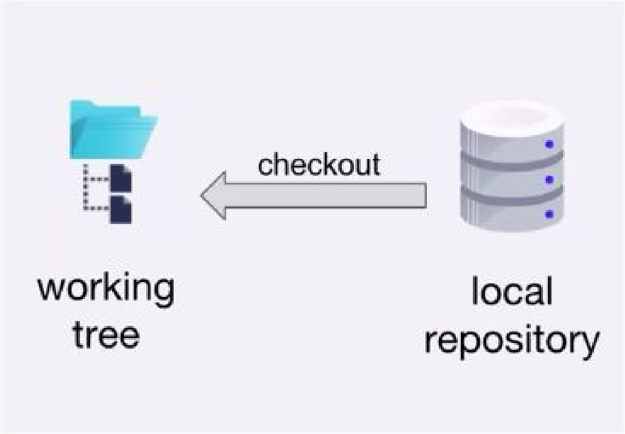
```
# create a branch
$ git branch featureX
$ git branch
* master
  featureX
```

git branch command is used to create branches. You specify the new branch name as an argument to the git branch command. Creating a branch simply creates a new branch label reference. You remain on the original branch.

Checkout

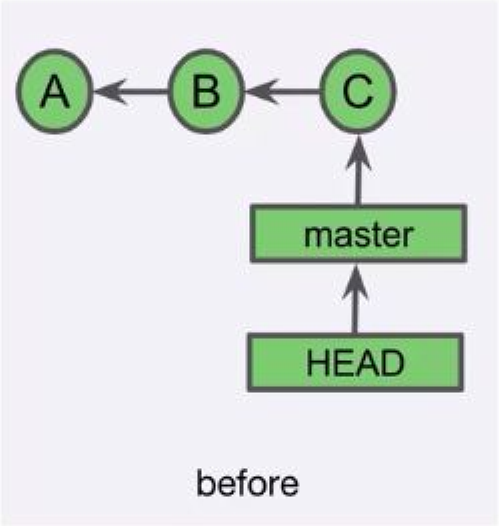
checkout is usually related to branches and does two main things. First, it switches the current commit which is the commit that the HEAD reference points to, to the checked out branch label or commit.
The second thing that checkout does, is it updates the working tree with the files from the checked out commit. Once you have checked out a branch, you can work on and commit to the branch.

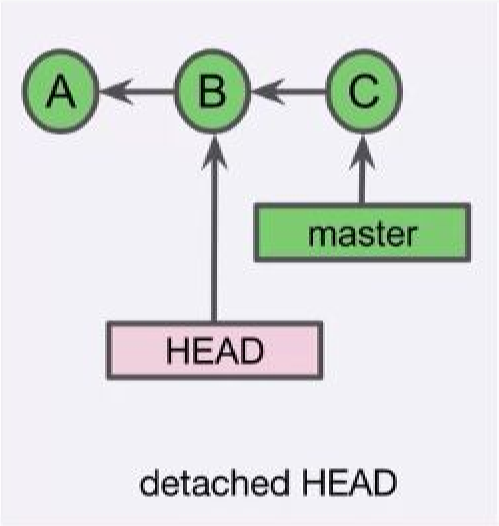




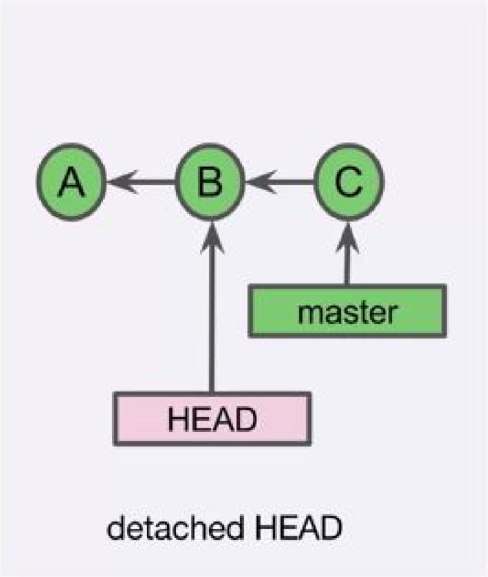
Detached Head

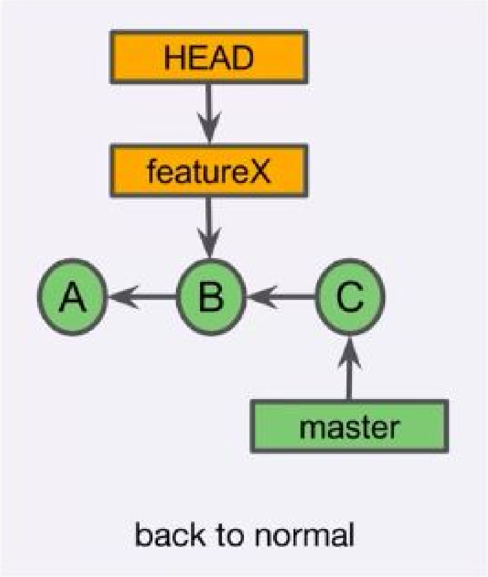
Usually you checkout a branch, which checks out the commit that the branch label points to, but you also have the option of directly checking out a commit. For example, you might checkout a commit to temporarily view an older version of the project. Checking out a commit rather than a branch leads to a detached HEAD state. This means that instead of the HEAD reference pointing to a branch label, HEAD points directly to the SHA-1 of a commit. So a detached HEAD basically means that the HEAD reference is detached from a branch label.



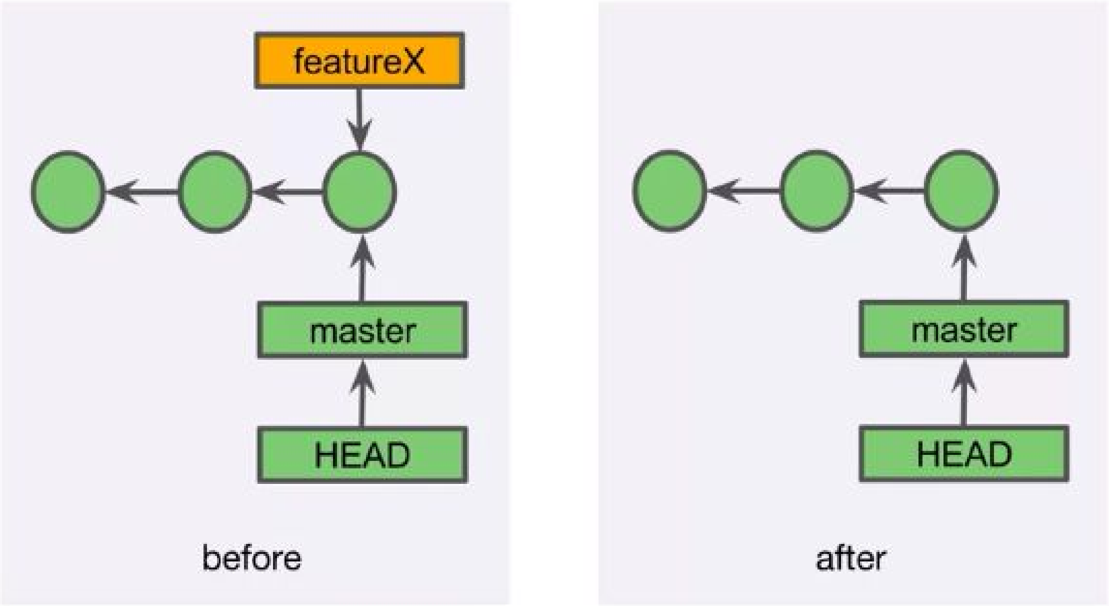


Temporarily viewing the files of a commit while in a detached HEAD state is perfectly fine. However, if you want to work on files of the checked out commit and create new commits, you should create a branch on that commit first. On the left, we've checked out commit B, creating a detached HEAD. If we want to create commits based off of commit B, we should first create a feature X branch label, and checkout that branch.. That puts us back in the normal state where the HEAD reference points to a branch label. The next commit will then be made to the feature X branch.





Deleting a Branch



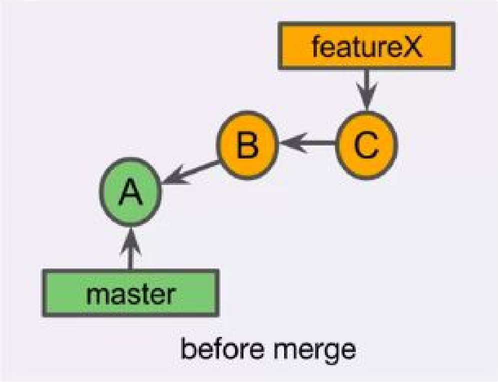
Deleting a branch really just means that you're deleting a branch label. Deleting a branch label normally does not delete any commits, at least not right away. In the example on the left, we created a feature X branch. This simply added the feature X label to the latest commit. If we then immediately delete the feature X branch, all that happens is that the feature X label is deleted. This shows how truly lightweight git branches are. Used the -d option on the git branch command to delete a branch. Here we start with two branches, but use the -d option, to delete the feature X branch. A call to git branch shows that the feature X branch has been deleted. Branch labels are commonly deleted after a topic branch has been merged.

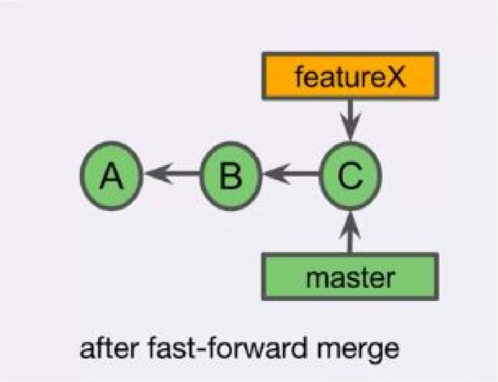
Merging

Merging combines the work of independent branches. Usually, this involves merging a topic branch, such as the featureX branch, into what is called a base branch, such as the master branch. The base branch is usually a longer running branch than the topic branch.

There are four main types of merges, fast-forward merge, merge commit, squash merge, and rebase.

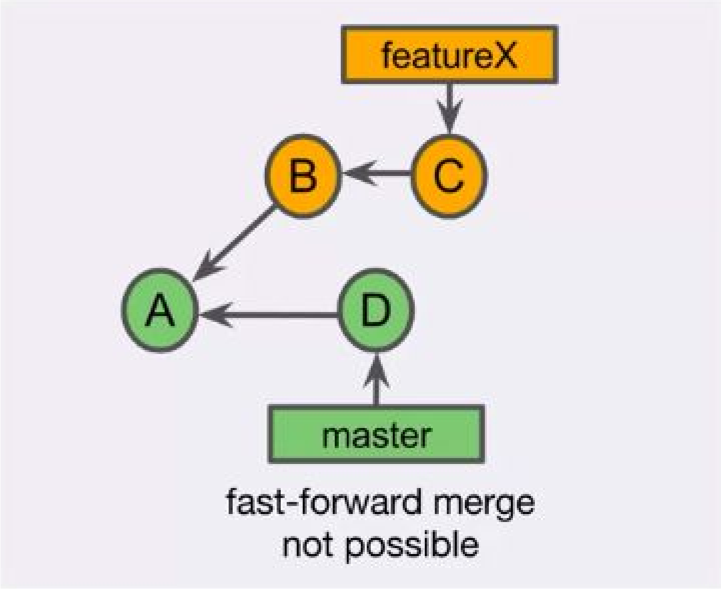
Fast-Forward Merge





A fast-forward merge moves the base branch label to the tip of the topic branch. Resulting commit history is linear.

A fast forward merge is possible only if no other commits have been made to the base branch since the topic branch was created.

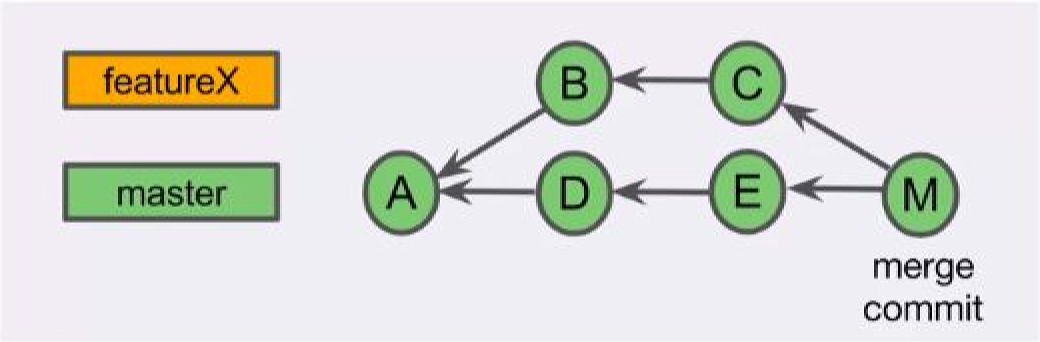


In this example, a fast-forward merge is not possible because commit D was created after the featureX branch was created. If git had allowed a fast-forward merge, the master branch label would move to commit C, and the work of commit D would be bypassed. Whoever did the work of commit D would not be very happy with that, so the merge is not fast-forwardable.

After a branch has been merged, its branch label can be deleted. This prevents a continuous increase in the number of merged branch labels as the project grows. Dealing with an ever increasing number of feature branch labels can be confusing.

Merge Commit

A merge commit combines the work of the tips of the feature and base branches and places the result into a single merge commit.

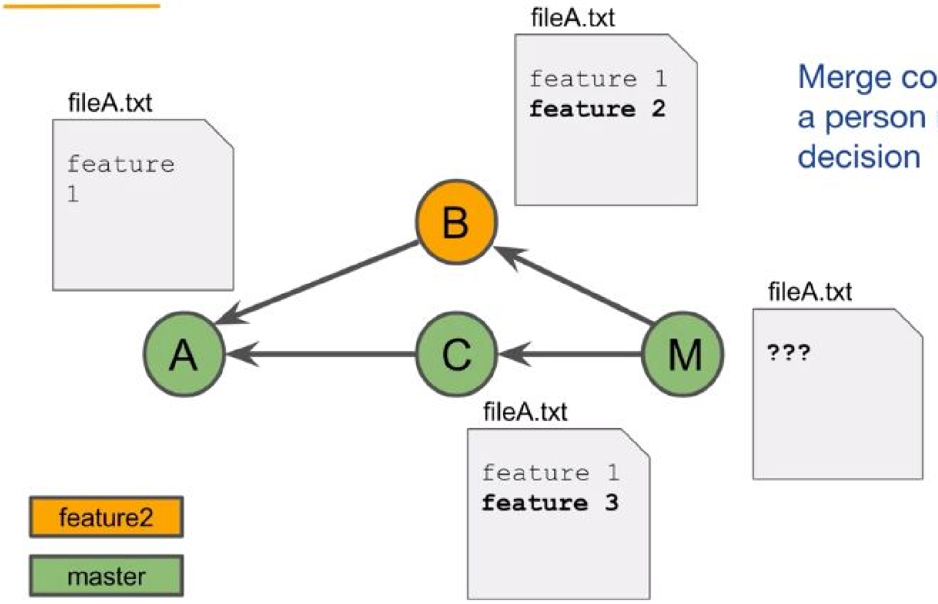


A merge commit always has multiple parents. We can see here that the commit M has two parents, commits C and E. Because of this non-linear commit graph, it's easy to see the branch in the commit history. Combining the work of multiple commits may result in what's called a merge conflict, if both branches change the same thing in different ways.

Merge Conflict

If you perform a merge with a merge commit, Git takes on the responsibility of combining the work of multiple branches and placing the result into a single merge commit. Git will try to do this automatically. However, there are cases where multiple branches make different changes to the same part of a file. In that case, a merge conflict occurs and a person needs to make a decision on how to resolve it.

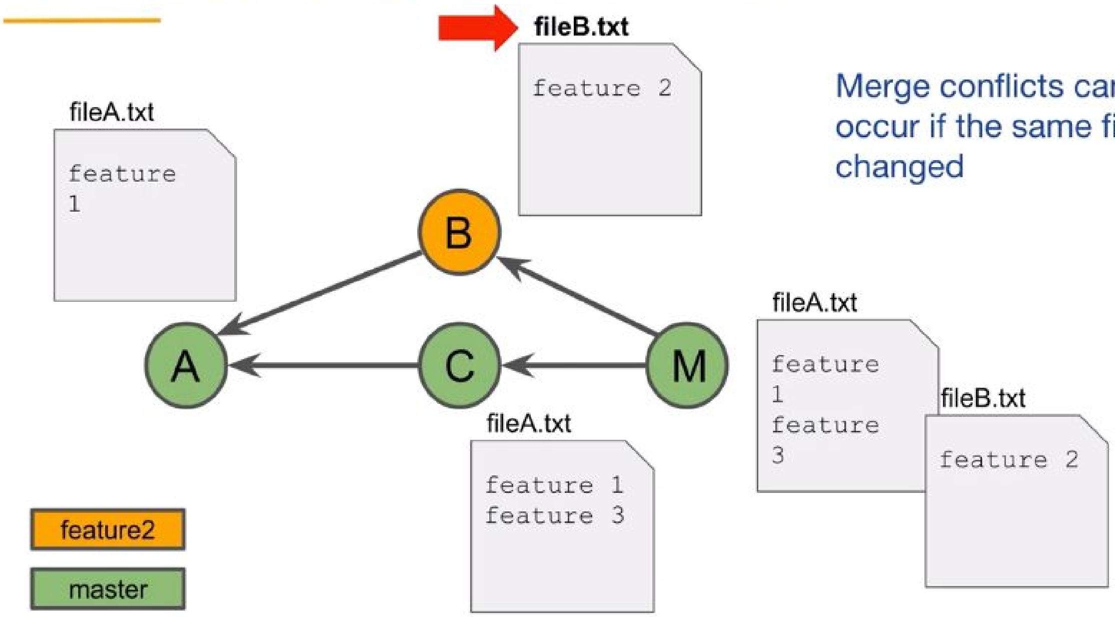
MERGE CONFLICTS



Merge conflicts occur when a person needs to make a decision

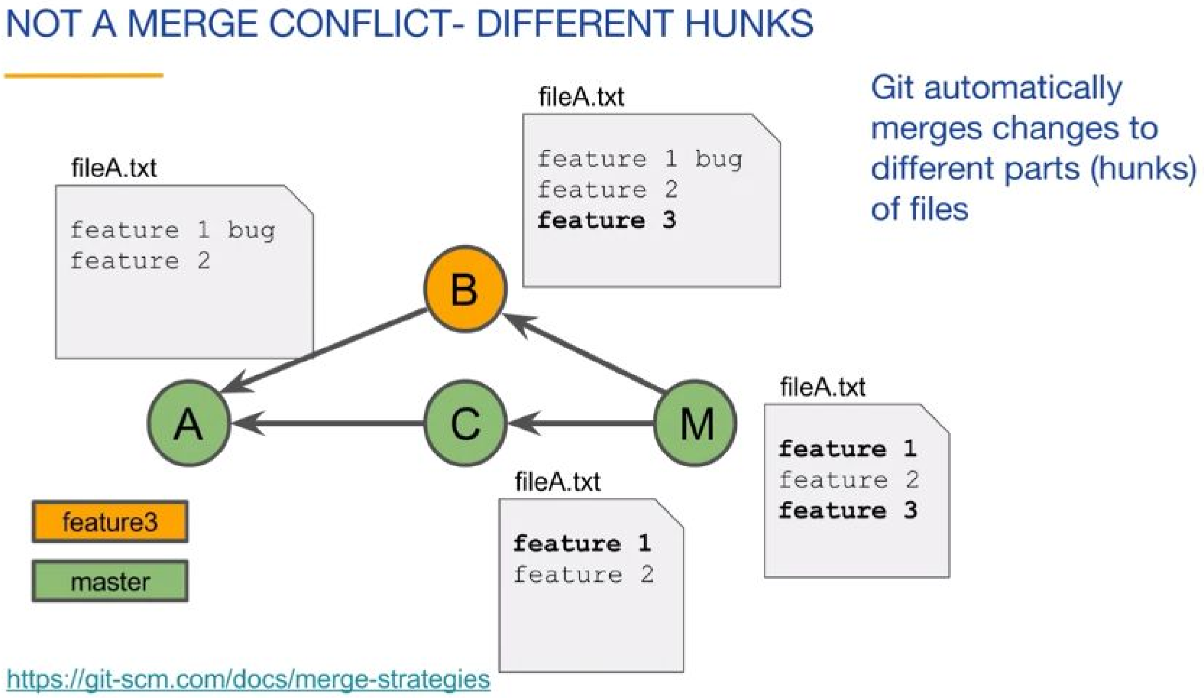
A merge conflict can occur if different branches change the same part of a file in different ways. If only one branch changes any file, you'll not have a merge conflict.

NOT A MERGE CONFLICT- DIFFERENT FILES



Merge conflicts can only occur if the same file is changed

Git also can automatically merge changes to different parts of the same file. In Git, a part of a file is called a hunk.



NOT A MERGE CONFLICT- DIFFERENT HUNKS

Git automatically merges changes to different parts (hunks) of files

fileA.txt

```
feature 1 bug
feature 2
```

fileA.txt

```
feature 1 bug
feature 2
feature 3
```

fileA.txt

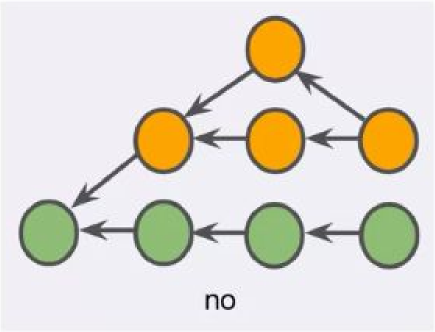
```
feature 1
feature 2
```

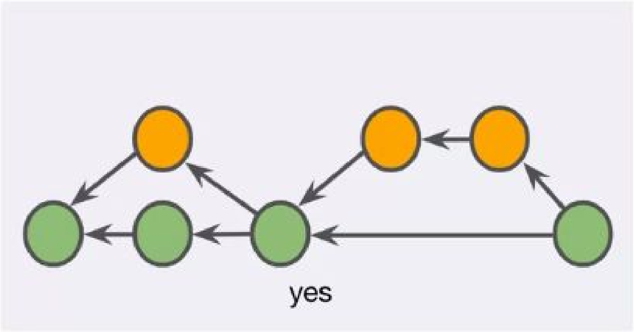
fileA.txt

```
feature 1
feature 2
feature 3
```

https://git-scm.com/docs/merge-strategies

We have seen that merge conflicts occur when two branches modify the same hunk of the same file. To make merging easier, avoid making a lot of changes over a long period of time without merging. Smaller frequent mergers are usually the best approach. It's better to create many small merge problems than one giant merge problem.



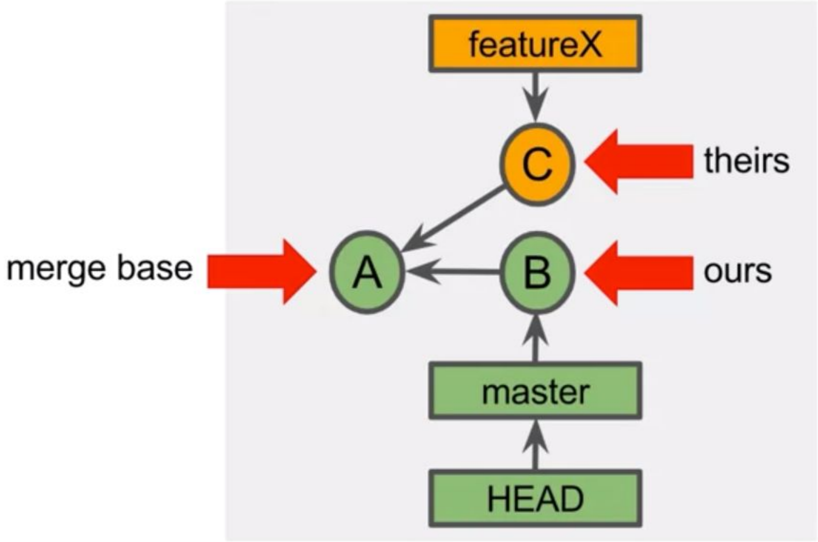


The diagram on the left shows what to avoid with Git. A branch was made off of the initial commit and then more branches and commits were made without merging back into the master branch. Notice that the master branch has been changing as we've been working. It's better to frequently merge, as shown in the diagram on the right. If your project is a software project, decoupled modular code is much less likely to have merged conflicts. In some ways, the number and complexity of merge conflicts is a test of how modular your code is.

RESOLVING A MERGE CONFLICT

Involves three commits:

1. The tip of the current branch (B)- "ours" or "mine"
2. The tip of the branch to be merged (C)- "theirs"
3. A common ancestor (A)- "merge base"



BASIC STEPS TO RESOLVE A MERGE CONFLICT

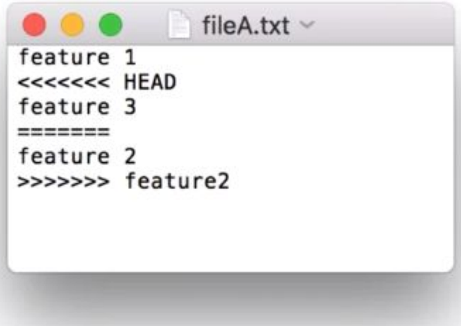
1. Checkout master
2. Merge featureX
    a. CONFLICT- Both modified fileA.txt
3. Fix fileA.txt
4. Stage fileA.txt
5. Commit the merge commit
6. Delete the featureX branch label

When attempting a merge, files with conflicts are **modified by Git** and placed in the working tree

The key point here is that, when attempting a merge, files with conflicts are modified by Git and placed in the working tree. You need to fix those files before executing a successful merge.

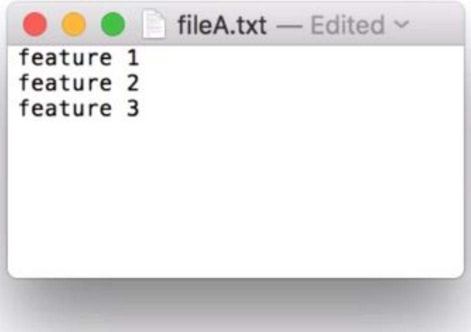
FIXING A CONFLICTED FILE

1. Checkout master
2. Merge featureX
3. **Fix fileA.txt**
4. Add fileA.txt
5. Commit the merge
6. Delete featureX



```
feature 1
<<<<<<< HEAD
feature 3
=======
feature 2
>>>>>>> feature2
```

before fixing a file

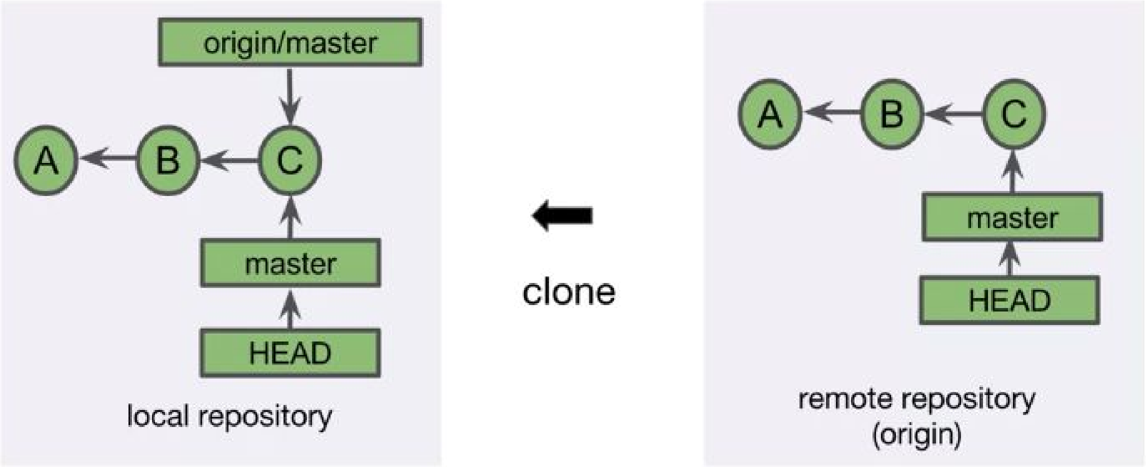


```
feature 1
feature 2
feature 3
```

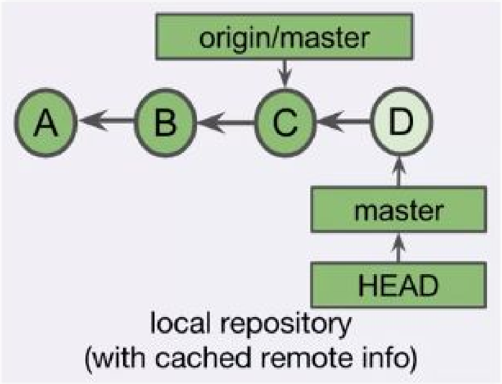
after fixing a file

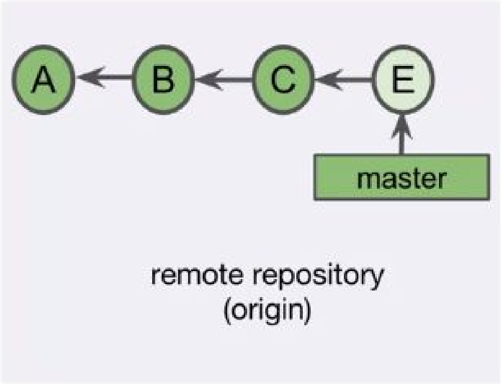
Tracking Branches

A tracking branch is a local branch that represents a remote branch. Locally, a tracking branch name looks like this: <remote name>/<branch name>. If you clone a repository, you'll have a default local tracking branch.



Tracking branches updates separately from both the remote branch and the local branch. This is because tracking branches are only updated with network commands like clone, fetch, pull and push. The tracking branch acts as a sort of intermediary between the local and remote branches.





In this example, our tracking branch points to commit C. This means that at the time of the most recent network command, commit C was the latest commit on the remote master branch. Locally, we then created commit D. This moves the head reference and master branch label to commit D. The tracking branch label stays behind because it only moves with network commands. Committing is a local command. If we look at the remote repository, we can see that since our last network command, someone pushed a new commit E. We are now in a state where the three master branch labels point to different commits. The local master branch label points to commit D, the master tracking branch label points to commit C, and the remote master branch label points to commit E. The three related branches are no longer in sync.

Viewing tracking branch names

By default, the git branch command only lists the names of local branches, use the all option of the git branch command to display all local and tracking branch names.

Viewing tracking branch status

The git status command includes tracking branch status. This command will inform you if the cached tracking branch information is out of sync with your local branch.

git status uses cached information from the last time that a command like fetch was executed. The tracking branch is a representation of the associated branch on the remote repository. If someone else made a commit since you last executed any network command like fetch, you would not know it by executing git status.

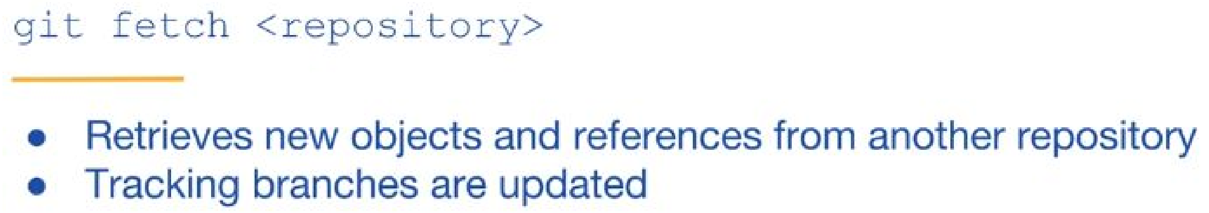
Network Commands

clone

The Clone command copies a remote repository and creates a local repository.

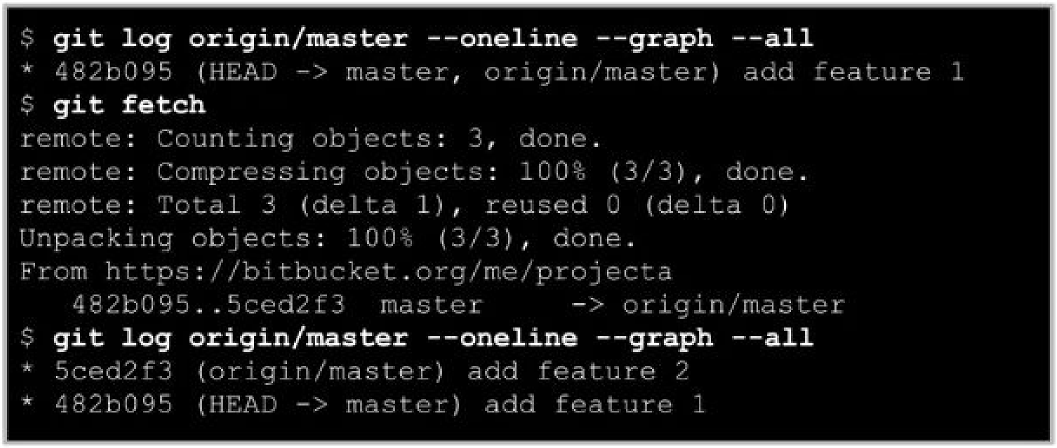
fetch

The fetch command retrieves the latest objects and references from the remote repository. The git fetch command retrieves new objects and references from another repository. It mainly updates all of your tracking branch information. If the repository argument isn't specified and you only have one remote repository set up locally, that remote repository will be used by default. Git fetch allows you to download and view changes on the remote repository without having to immediately merge them into your current work.



```
git fetch <repository>
```

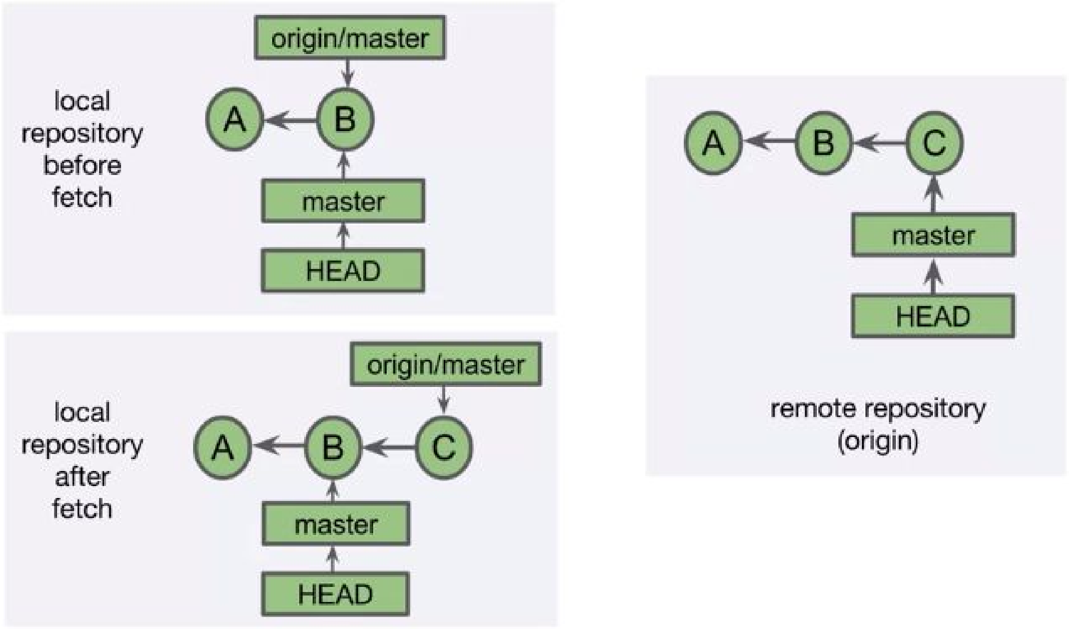
- Retrieves new objects and references from another repository
- Tracking branches are updated



```
$ git log origin/master --oneline --graph --all
* 482b095 (HEAD -> master, origin/master) add feature 1
$ git fetch
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://bitbucket.org/me/projecta
   482b095..5ced2f3  master     -> origin/master
$ git log origin/master --oneline --graph --all
* 5ced2f3 (origin/master) add feature 2
* 482b095 (HEAD -> master) add feature 1
```

Let's look at an example, we start by viewing our commit graph. It has one commit and we can see that this commit is located on the local and remote repositories, because the local master branch label and the tracking branch label point to it. We then execute git fetch. Git fetch is objects and references from the remote repository. Because it returned results we know that something has changed on the remote repository since our last network command. Next, we again execute the git log command, we can see that the tracking branch is one commit ahead of our local master branch. Somebody implemented feature 2 since our last network command. We can see that out master branch label is still pointing to the same commit that it was before the fetch.

FETCHING THE LATEST COMMITS

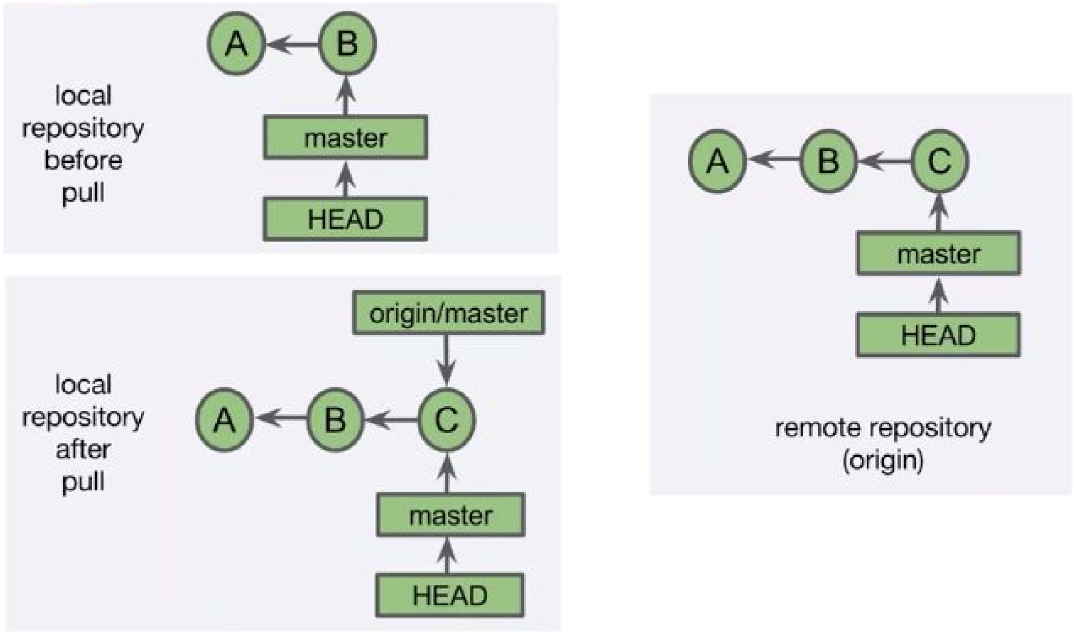

local repository before fetch
origin/master
A
B
master
HEAD
local repository after fetch
origin/master
A
B
C
master
HEAD
A
B
C
master
HEAD
remote repository (origin)

After we execute the fetch, the local repository has all of the commits and references from the remote repository, but they are on the tracking branch. You can see that this part of the local repository matches this part of the remote repository. Our local master branch label has not moved because of the fetch. A fetch does not impact the local branch labels. After the fetch, the tracking branch has the same commits as the remote repository's master branch.
After a fetch, executing git status will inform you if your current local branch is behind the tracking branch.

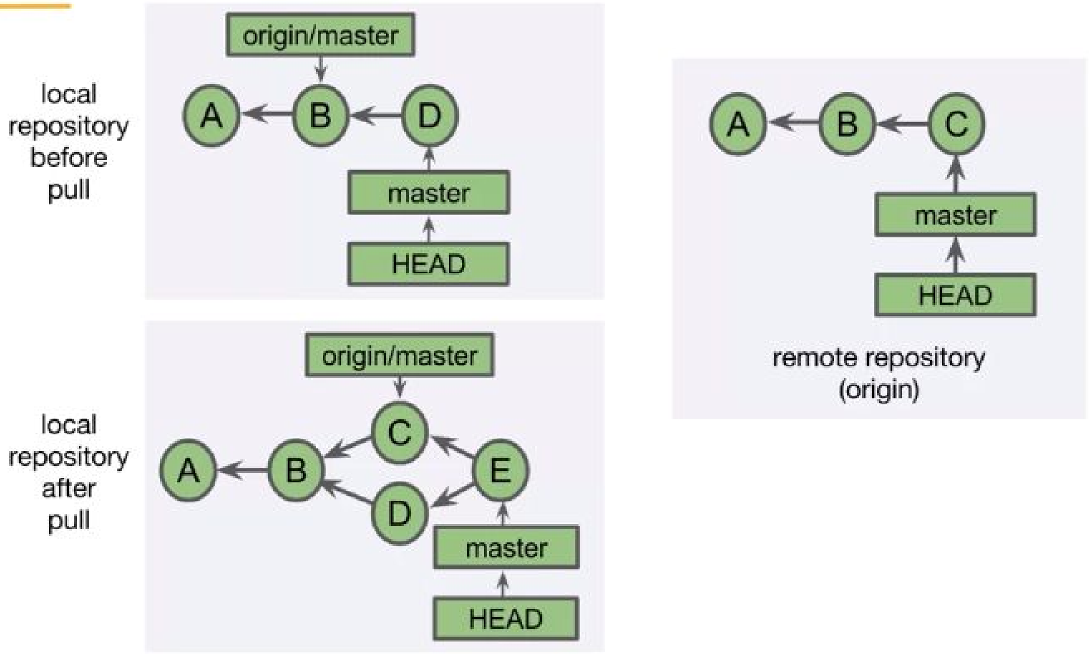
pull

The Pull command combines a fetch and a merge.
Git pull is a network command that combines git fetch and git merge FETCH_HEAD. FETCH_HEAD is an alias for the tip of the tracking branch. First new objects and references from the remote repository are fetched. If new objects are added to the tracking branch, the tracking branch is merged into the local branch. The current branch is assumed if branch is not specified.

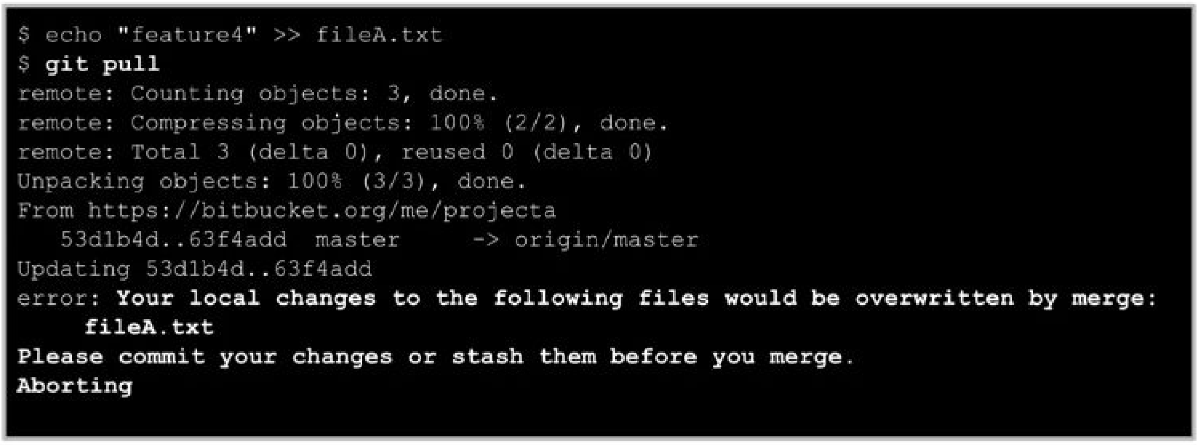
PULL WITH A FAST-FORWARD MERGE


local repository before pull
A
B
master
HEAD
local repository after pull
origin/master
A
B
C
master
HEAD
A
B
C
master
HEAD
remote repository (origin)

PULL WITH A MERGE COMMIT


local repository before pull
origin/master
A
B
D
master
HEAD
local repository after pull
origin/master
A
B
C
D
E
master
HEAD
A
B
C
master
HEAD
remote repository (origin)

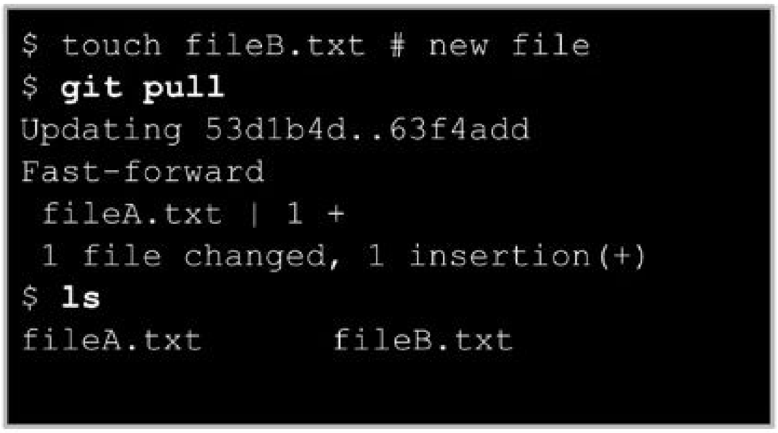
git pull WITH CONFLICTING UNCOMMITTED CHANGES



```
$ echo "feature4" >> fileA.txt
$ git pull
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://bitbucket.org/me/projecta
   53d1b4d..63f4add  master     -> origin/master
Updating 53d1b4d..63f4add
error: Your local changes to the following files would be overwritten by merge:
     fileA.txt
Please commit your changes or stash them before you merge.
Aborting
```

We then execute git pull. Git fetches some objects, but then when it tries to perform a merge, it notices that you have local changes to a file that it will be replacing. It suggests that you commit or stash your changes before the merge, then it aborts the merge. Stashing is a way to save files modified in the working tree for later access.

git pull WITH SAFE UNCOMMITTED CHANGES

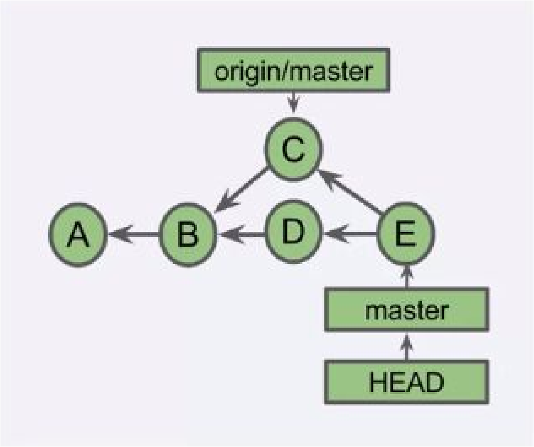


```
$ touch fileB.txt # new file
$ git pull
Updating 53d1b4d..63f4add
Fast-forward
 fileA.txt | 1 +
 1 file changed, 1 insertion(+)
$ ls
fileA.txt       fileB.txt
```

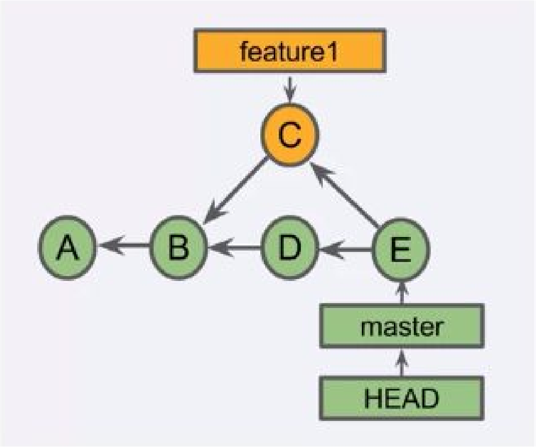
Git pull only aborts the merge if you have uncommitted changes that would be overwritten by git. If you've modified a file that's not going to be replaced by git it lets the merge continue. Let's look at an example. First we create a file in our working tree. Since it's brand new, git is not tracking it and it shouldn't cause any problems. We then execute git pull. It fetches an object and performs a fast forward merge, replacing fileA.txt in the working tree.

We then list the files of the directory and we see that our uncommitted fileB.txt is still there, git hasn't touched it.

THE TRACKING BRANCH IS LIKE A TOPIC BRANCH



pull with merge commit



feature branch merge

Notice the similarities between merging after a pull and emerging in a feature branch. On the left we create a merge commit after a pull. Notice that a merge commit was created even know that only branch is the master branch. This is because the tip of the tracking branch is acting like a topic branch. To git, the tracking branch is just another branch. On the right, we merge in a topic branch. Notice that the commit graphs are basically the same. Anything that can happen when merging in a topic branch can happen when merging in a tracking branch.

push

The Push command adds new objects and references to the remote repository.
You can use the -u option to set up the local tracking branch with this remote branch. Once you've done this, the repository and branch name no longer need to be specified in the command, because that information is known from the tracking branch.

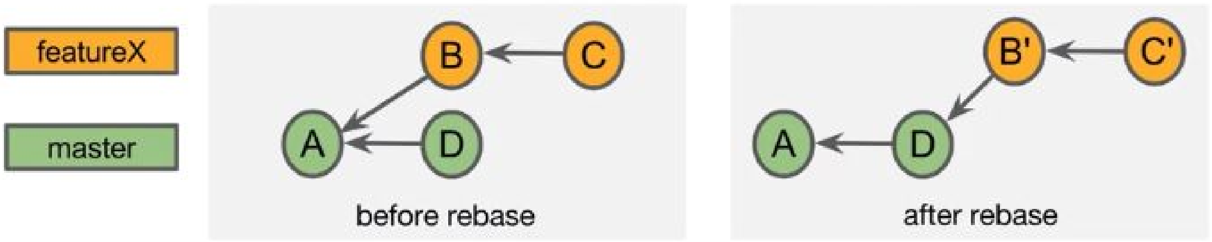
A general rule is to fetch or pull before you do a push. You don't have to do this but if you try to push when you don't have the latest remote changes, the push will fail. If you execute a fetch and no objects are retrieved, we can safely push.

Rebase

There is a general rule related to Rebase. Do not rewrite history that has been shared with others. If you've been working locally or if you know that no one else has used your branch you can safely Rebase it.

There are two types of Rebase, a regular Rebase and an interactive Rebase.

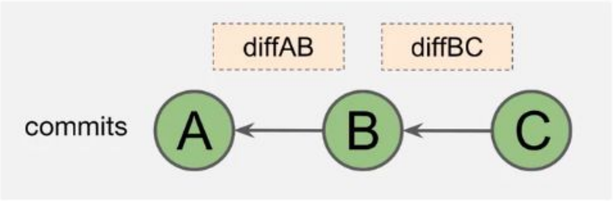
A rebase moves commits to a new parent or base.

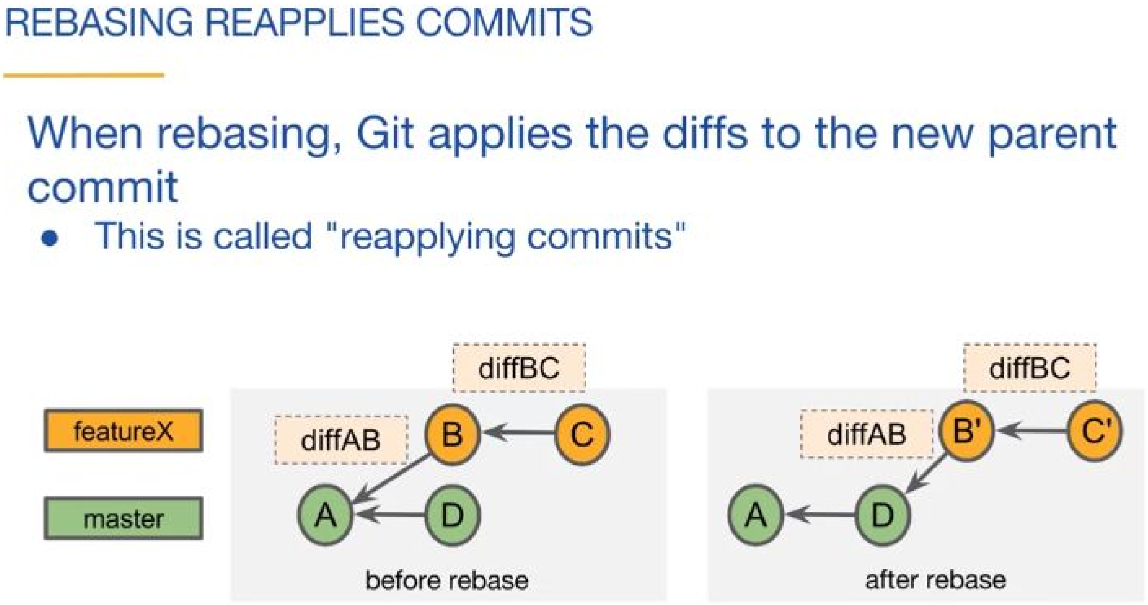


If we look at the example on the left, we have a situation that would typically involve a "merge commit" because commit D has been made after the featureX branch was made. However, there is another option and that is to rebase. What rebasing does is take commit B and C and moves them to a new parent commit D. The result is that you no longer need a merge commit and the merge can be fast forwarded. Because the commits have been moved, they are reapplied on top of the new commit. This creates a different ancestor chain and as a result each of the commit IDs change. So in this example commit B changes to B prime and commit C changes to C prime. You can see that this is necessary because before the rebase commit B's parent was A. And after the rebase commit B prime's parent is D.

DIFFS

- Each commit contains a snapshot of the complete project
- Git can *calculate* the difference between commits
    - This is known as a *diff* or a *patch*





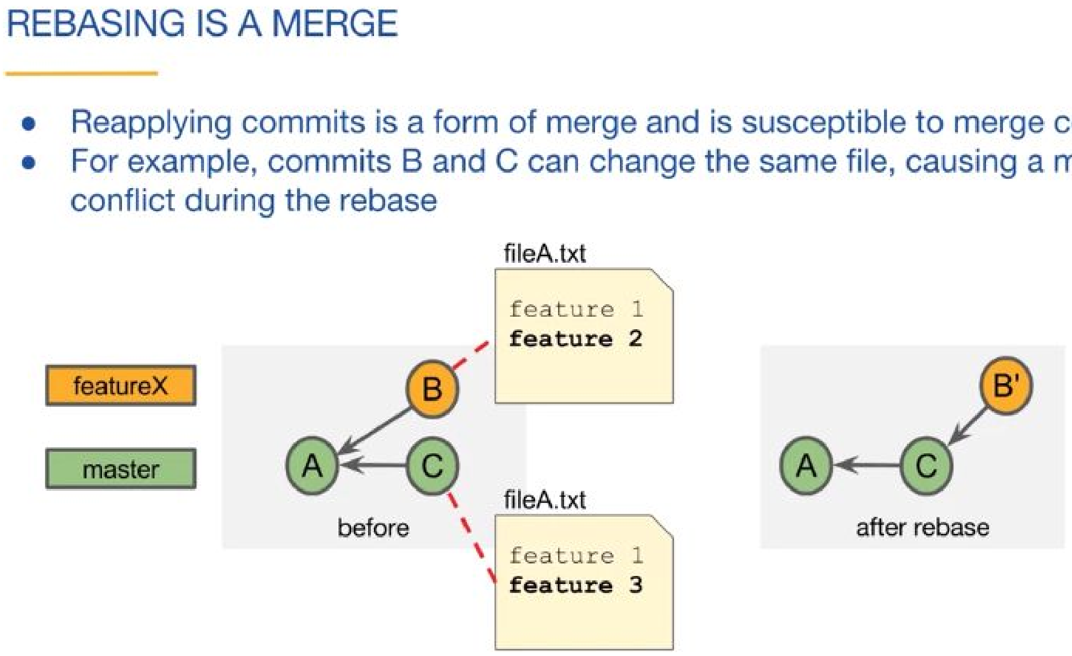
REBASING REAPPLIES COMMITS

When rebasing, Git applies the diffs to the new parent commit

- This is called "reapplying commits"

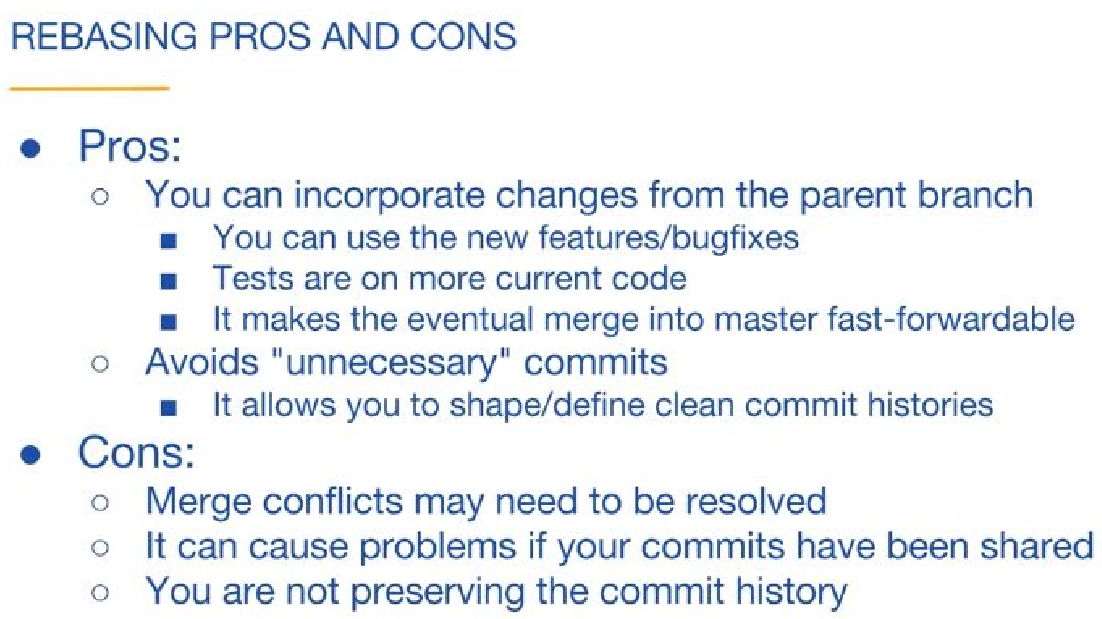
In the before commit graph on the left, we can see that the parent of B is A and the parent of C is B and the difference between B and A is diff AB and the difference between C and B is diff BC. With rebasing, Git takes the difference between A and B and applies it with a parent of commit D. This creates a new commit ID. It then applies the difference between commits B and C and creates commit C prime.

Reapplying commits is a form of merge and is susceptible to merge conflicts.



REBASING IS A MERGE

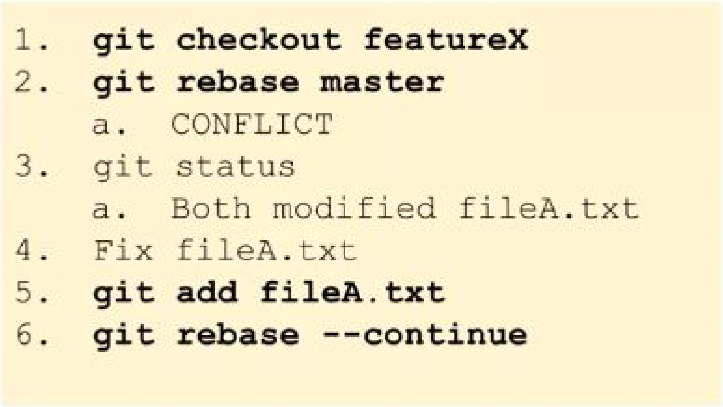
- Reapplying commits is a form of merge and is susceptible to merge conflicts
- For example, commits B and C can change the same file, causing a merge conflict during the rebase



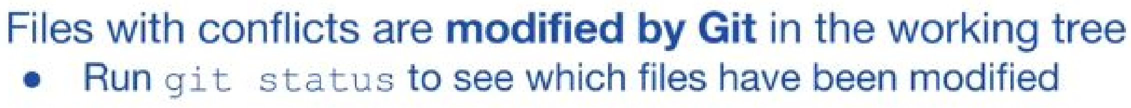
REBASING PROS AND CONS

- Pros:
  - You can incorporate changes from the parent branch
    - You can use the new features/bugfixes
    - Tests are on more current code
    - It makes the eventual merge into master fast-forwardable
  - Avoids "unnecessary" commits
    - It allows you to shape/define clean commit histories
- Cons:
  - Merge conflicts may need to be resolved
  - It can cause problems if your commits have been shared
  - You are not preserving the commit history

Rebasing with Conflicts



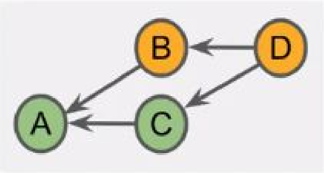
```
1.  git checkout featureX
2.  git rebase master
    a.  CONFLICT
3.  git status
    a.  Both modified fileA.txt
4.  Fix fileA.txt
5.  git add fileA.txt
6.  git rebase --continue
```



Files with conflicts are **modified by Git** in the working tree

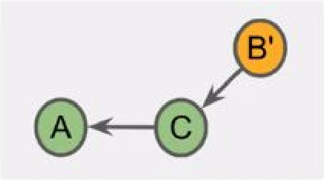
- Run `git status` to see which files have been modified

RESOLVING MERGE CONFLICTS- COMPARING MERGE TO REBASE



```
1.  git checkout master
2.  git merge featureX
    a.  CONFLICT
3.  git status
    a.  Both modified fileA.txt
4.  Fix fileA.txt
5.  git add fileA.txt
6.  git commit
```

merge



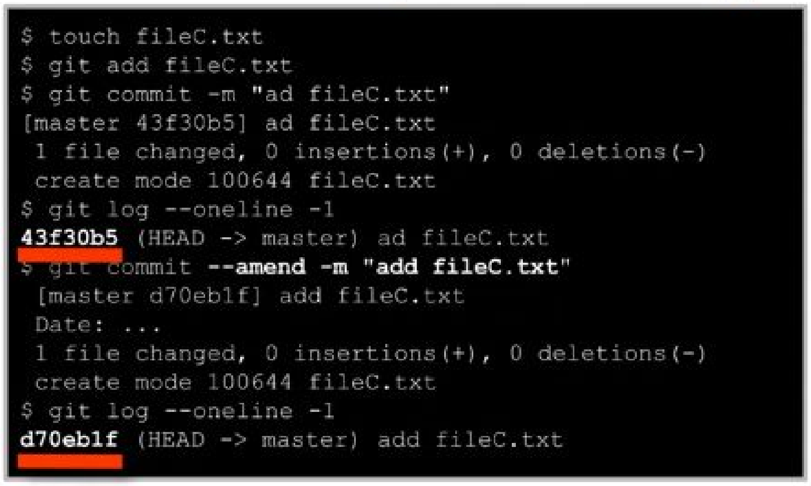
```
1.  git checkout featureX
2.  git rebase master
    a.  CONFLICT
3.  git status
    a.  Both modified fileA.txt
4.  Fix fileA.txt
5.  git add fileA.txt
6.  git rebase --continue
```

rebase

Rewriting History

AMENDING A COMMIT

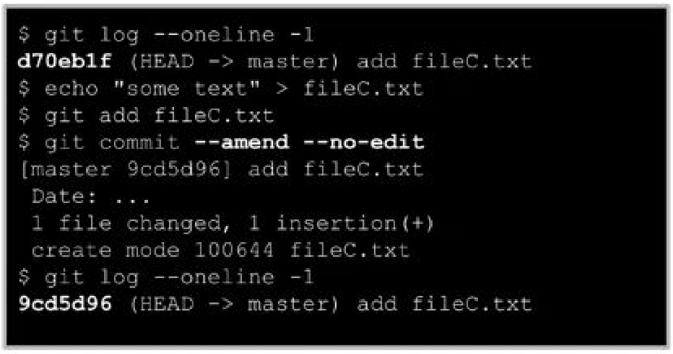
- You can change the most recent commit:
  - change the commit message
  - change the project files
- This creates a new SHA-1 (rewrites history)



```
$ touch fileC.txt
$ git add fileC.txt
$ git commit -m "ad fileC.txt"
[master 43f30b5] ad fileC.txt
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fileC.txt
$ git log --oneline -1
43f30b5 (HEAD -> master) ad fileC.txt
$ git commit --amend -m "add fileC.txt"
 [master d70eb1f] add fileC.txt
 Date: ...
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fileC.txt
$ git log --oneline -1
d70eb1f (HEAD -> master) add fileC.txt
```

AMENDING A COMMIT- CHANGING COMMITTED FILES

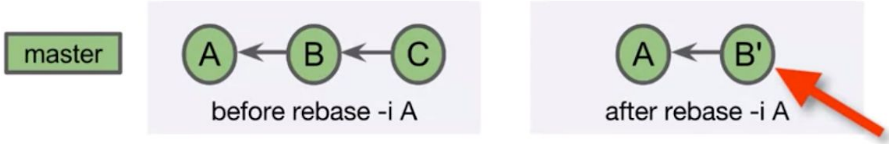
- You can modify the staging area and amend a commit
- Optionally use the `--no-edit` option to reuse the previous commit message

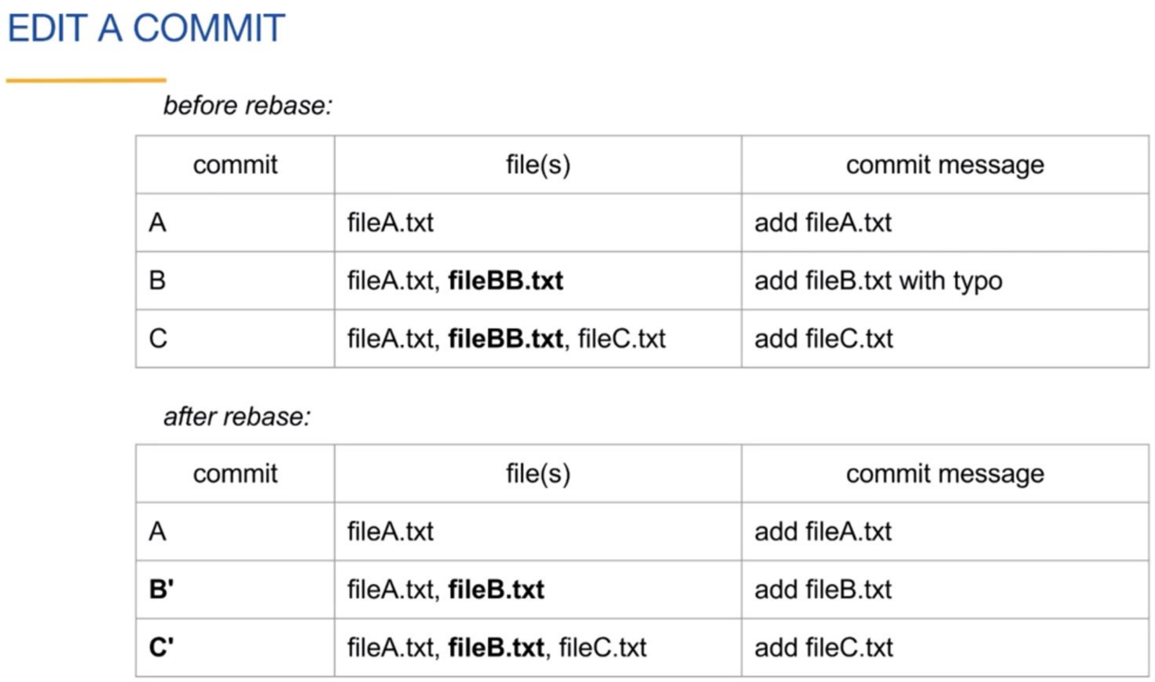


```
$ git log --oneline -1
d70eb1f (HEAD -> master) add fileC.txt
$ echo "some text" > fileC.txt
$ git add fileC.txt
$ git commit --amend --no-edit
[master 9cd5d96] add fileC.txt
 Date: ...
 1 file changed, 1 insertion(+)
 create mode 100644 fileC.txt
$ git log --oneline -1
9cd5d96 (HEAD -> master) add fileC.txt
```

Interactive Rebase

- Interactive rebase lets you edit commits using commands
  - The commits can belong to any branch
  - The commit history is changed- do not use for shared commits
- `git rebase -i <after-this-commit>`
  - Commits in the current branch after `<after-this-commit>` are listed in an editor and can be modified





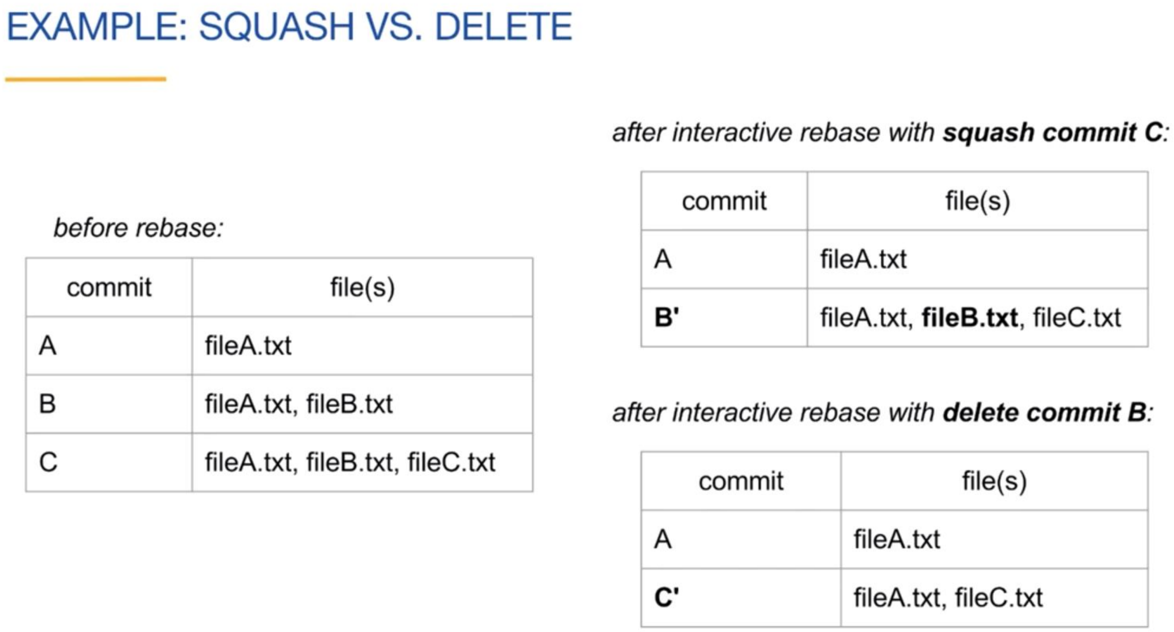
EDIT A COMMIT

before rebase:

commit	file(s)	commit message
A	fileA.txt	add fileA.txt
B	fileA.txt, **fileBB.txt**	add fileB.txt with typo
C	fileA.txt, **fileBB.txt**, fileC.txt	add fileC.txt

after rebase:

commit	file(s)	commit message
A	fileA.txt	add fileA.txt
B'	fileA.txt, **fileB.txt**	add fileB.txt
C'	fileA.txt, **fileB.txt**, fileC.txt	add fileC.txt



EXAMPLE: SQUASH VS. DELETE

before rebase:

commit	file(s)
A	fileA.txt
B	fileA.txt, fileB.txt
C	fileA.txt, fileB.txt, fileC.txt

after interactive rebase with ***squash commit C****:*

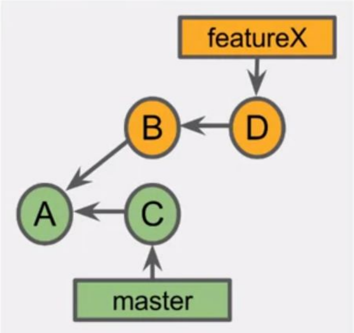
commit	file(s)
A	fileA.txt
B'	fileA.txt, **fileB.txt**, fileC.txt

after interactive rebase with ***delete commit B****:*

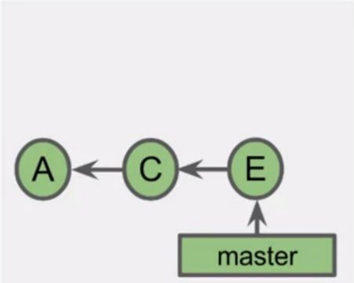
commit	file(s)
A	fileA.txt
C'	fileA.txt, fileC.txt

SQUASH MERGE

1. Merges the tip of the feature branch (D) onto the tip of the base branch (C)
   - There is a chance of a merge conflict
2. Places the result in the **staging area**
3. The result can then be committed (E)



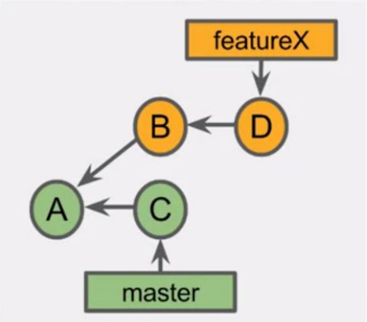
before merge



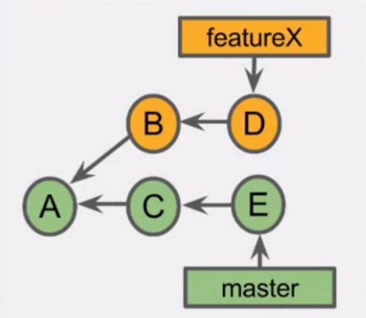
after squash merge
(and delete feature)

WHAT HAPPENS TO THE FEATURE COMMITS?

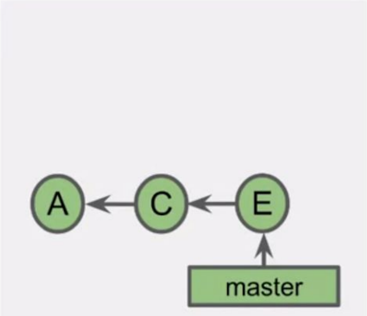
- After the **featureX** label is deleted, commits B and D are no longer part of any named branch
  - Commits B and D will eventually be garbage collected
- A squash merge *rewrites the commit history*



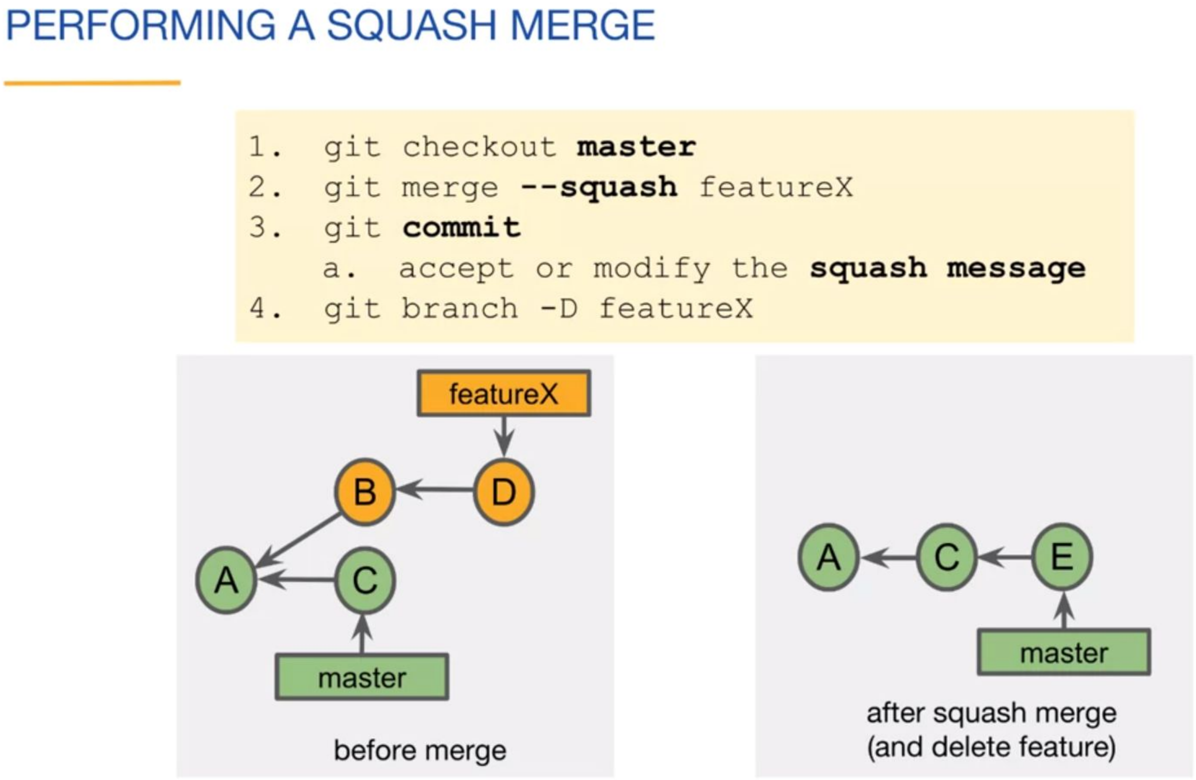
before squash
merge



after squash
merge



after squash merge
(and delete feature)



PERFORMING A SQUASH MERGE

```
1.  git checkout master
2.  git merge --squash featureX
3.  git commit
    a.  accept or modify the squash message
4.  git branch -D featureX
```

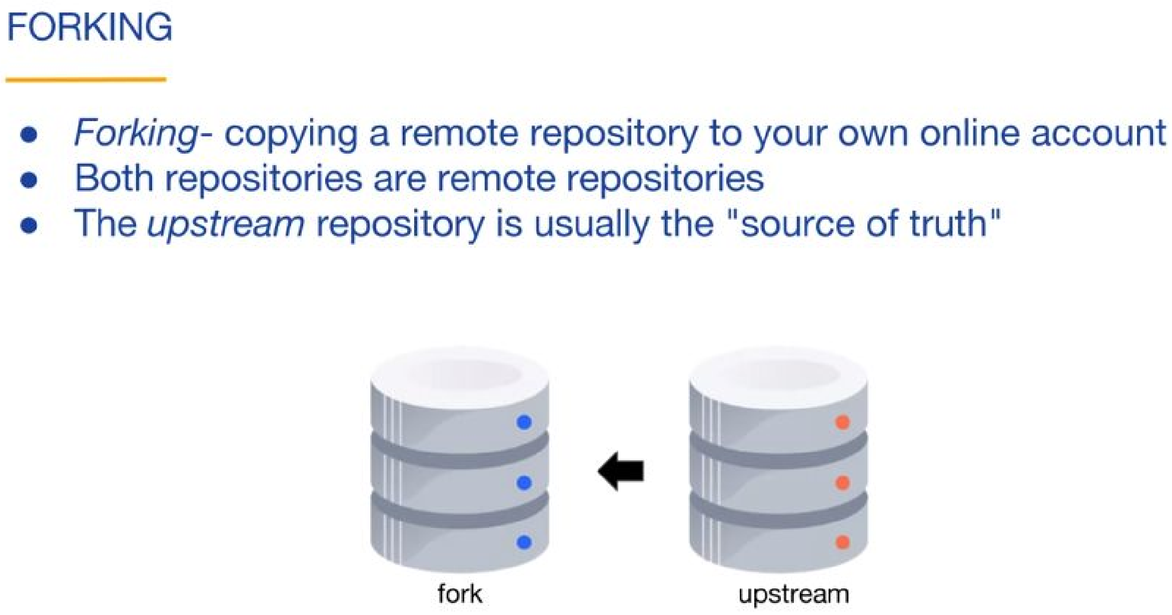
Pull Requests

Pull requests are a feature of Git hosting sites such as GitHub and Bitbucket. The ultimate goal of a pull request is to merge a branch into the project.
Pull requests enable team communication related to the work of the branch. Notifications, related to the pull request, can be sent to team members. Those team members can provide feedback or comments, and ultimately can a have a say in approving the content, this can act as a form of code review.
There are two basic repository configurations related to pull requests. The first one is a single remote repository, a pull request in a single repository configuration is a request to merge a branch of the repository. The second configuration involves two remote repositories, in this configuration, a pull request is a request to merge a branch from a forked repository into the upstream repository. The fork approach is common if the submitter doesn't have write access to the upstream repository.
You can create a pull request, which is also called opening a pull request, anytime during the life of the branch.

Before making a pull request, you need to prepare to make the request. To prepare to make the pull request, first you create a feature branch. This is the branch that we hope will eventually be merged into a longer running branch. You can work on the future branch before opening a pull request, but to start the team discussion, you can also open a pull request immediately after creating the branch. You then push the branch to the remote repository.

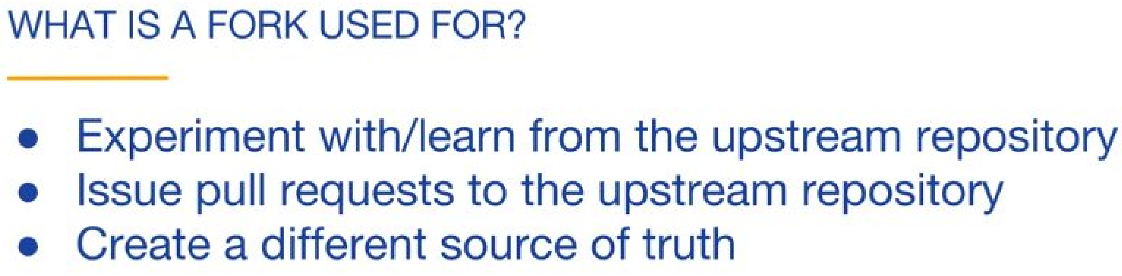
Forking

Forking generally means copying a remote repository to your own online account.
It is a feature of get hosting services such a Bitbucket and GitHub.



FORKING

- *Forking*- copying a remote repository to your own online account
- Both repositories are remote repositories
- The *upstream* repository is usually the "source of truth"



WHAT IS A FORK USED FOR?

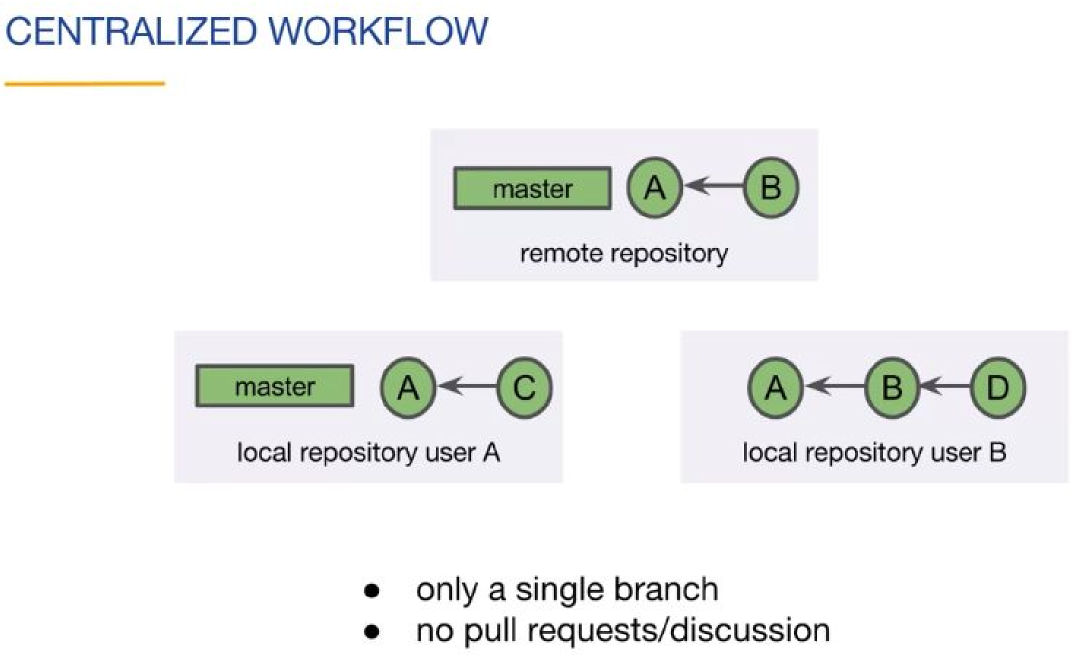
- Experiment with/learn from the upstream repository
- Issue pull requests to the upstream repository
- Create a different source of truth

A fork can be used for a number of reasons.
It can be used to experiment with or learn from the upstream repository. You have a separate remote repository that can be synchronized with the upstream repository as the upstream repository changes.
A fork can also be used to create branches and issue pull requests to the upstream repository.
This is how you can contribute to the upstream project.

A fork can also be used to create a different source of truth. In other words, it can be used to start a new line of development of the project that remains independent from the upstream repository. You could use the upstream project as a starting point for a different type project or you could create a source of truth that competes with the upstream project. This may or may not be a desirable thing to do. It maybe better to just help improve the upstream project.

If commits are made on the upstream repository, by default the fork will not contain those commits.

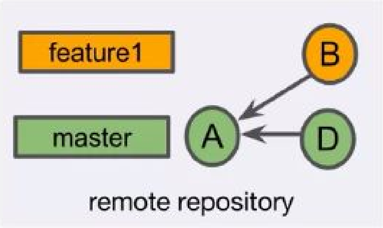
Git Workflow



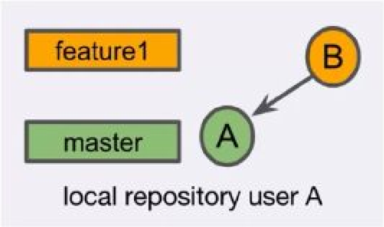
CENTRALIZED WORKFLOW

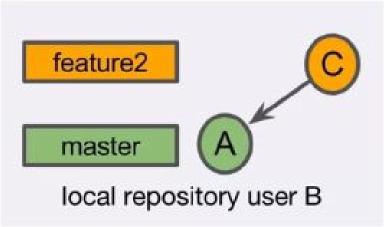
- only a single branch
- no pull requests/discussion

FEATURE BRANCH WORKFLOW

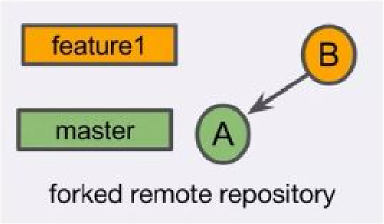


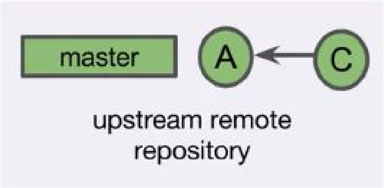
- work done on feature/topic branches
- single remote repository
- pull requests/discussion

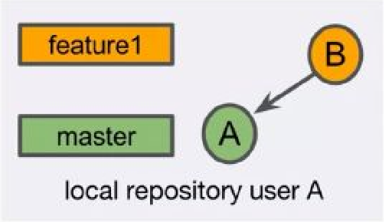




FORKING WORKFLOW

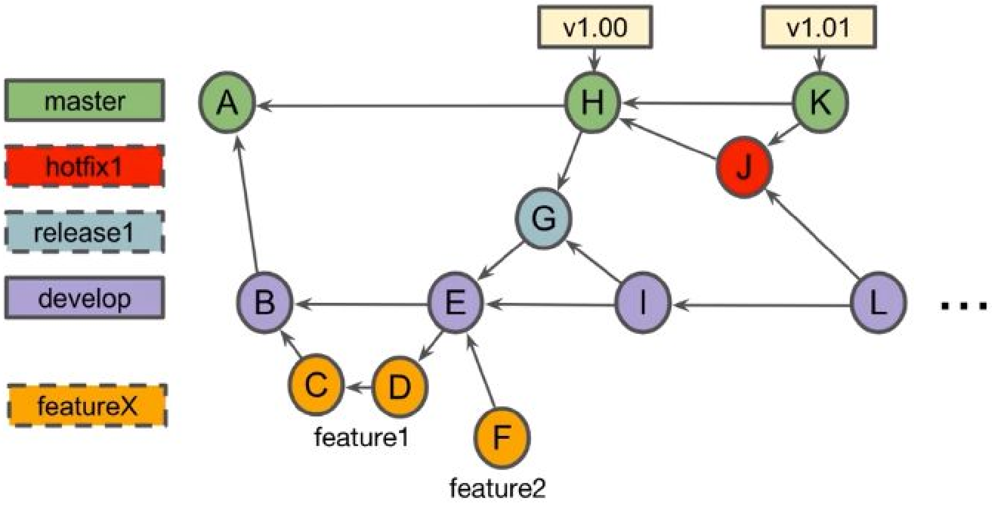




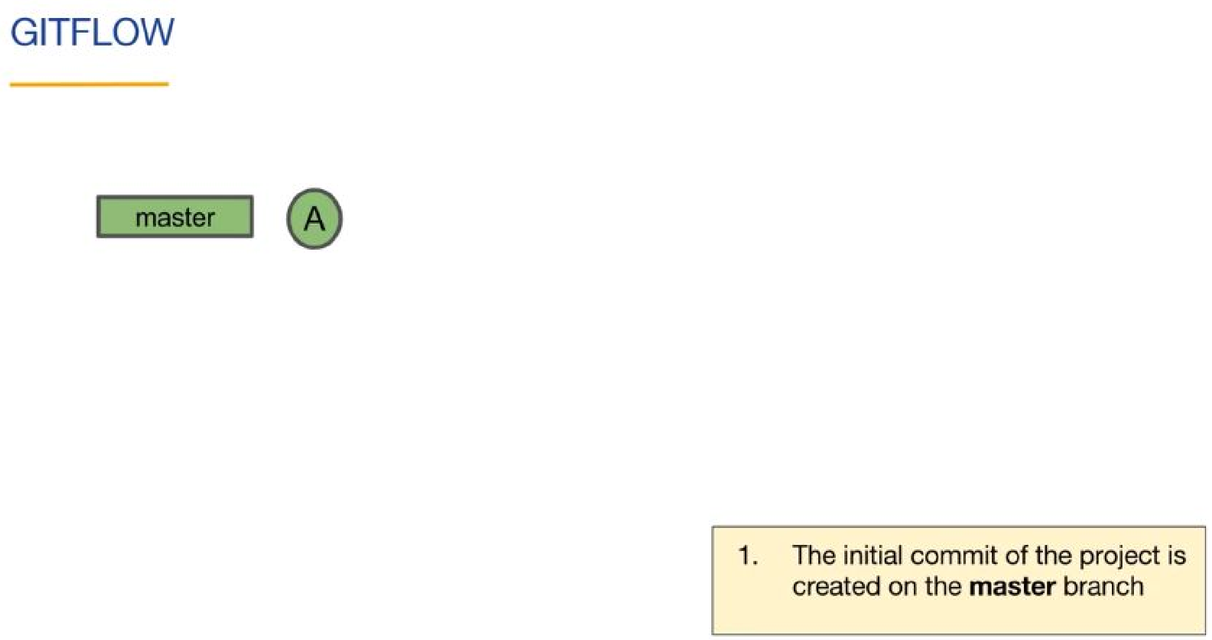


- multiple remote repositories
- pull requests/discussion
- don't need write access on upstream
- backs up your work in progress
- can rebase your forked branch
- must synchronize with upstream

GITFLOW



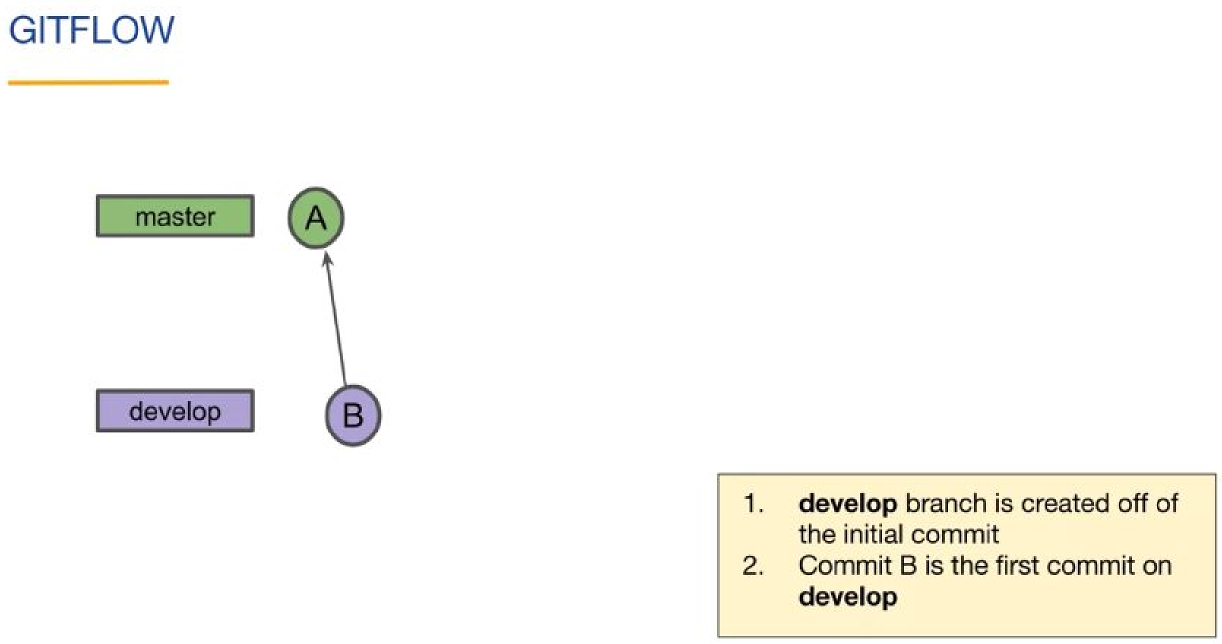
GitFlow is a workflow that allows safe continuous releases of the project. It allows work to continue even through releases and hotfixes. This commit graph is an example in the spirit of a GitFlow workflow. The general ideas here are used in specific ways that depend on the team and the type of project. We will break this commit graph down step by step shortly, but for now you can see that GitFlow projects involve a number of branches. In this diagram, the long running branch labels have solid borders and the short lived branch labels have dashed borders. You can see that the master and developed branches are the only long running branches. If you look at the master branch, you can see that there are three commits. Commit A is the initial commit in the repository. We will assume that any commits on the master branch after commit A represent a version of our project that customers can use. These releases have been tagged with version labels. This commit graph includes a release of version1 of the project, then shortly follows with a minor update. We are going to keep things simple and release version 1 of the project with a single feature named feature1.



GITFLOW

1. The initial commit of the project is created on the **master** branch

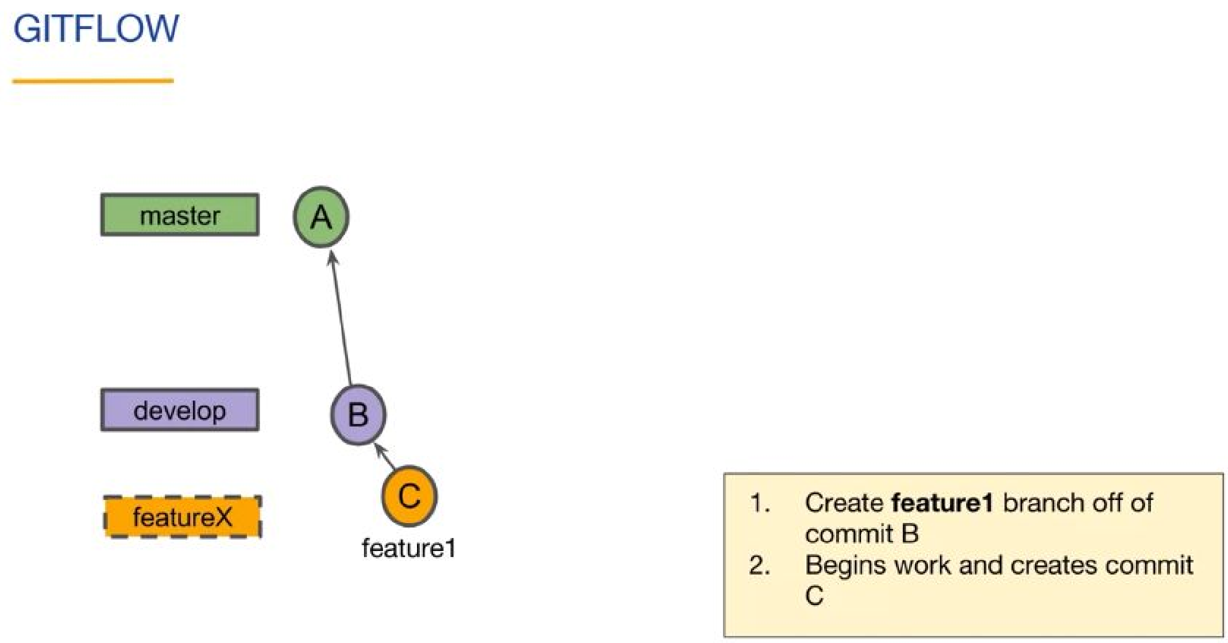
The initial commit of the project is created on the master branch. It can be very simple, like a read me file.



GITFLOW

1. **develop** branch is created off of the initial commit
2. Commit B is the first commit on **develop**

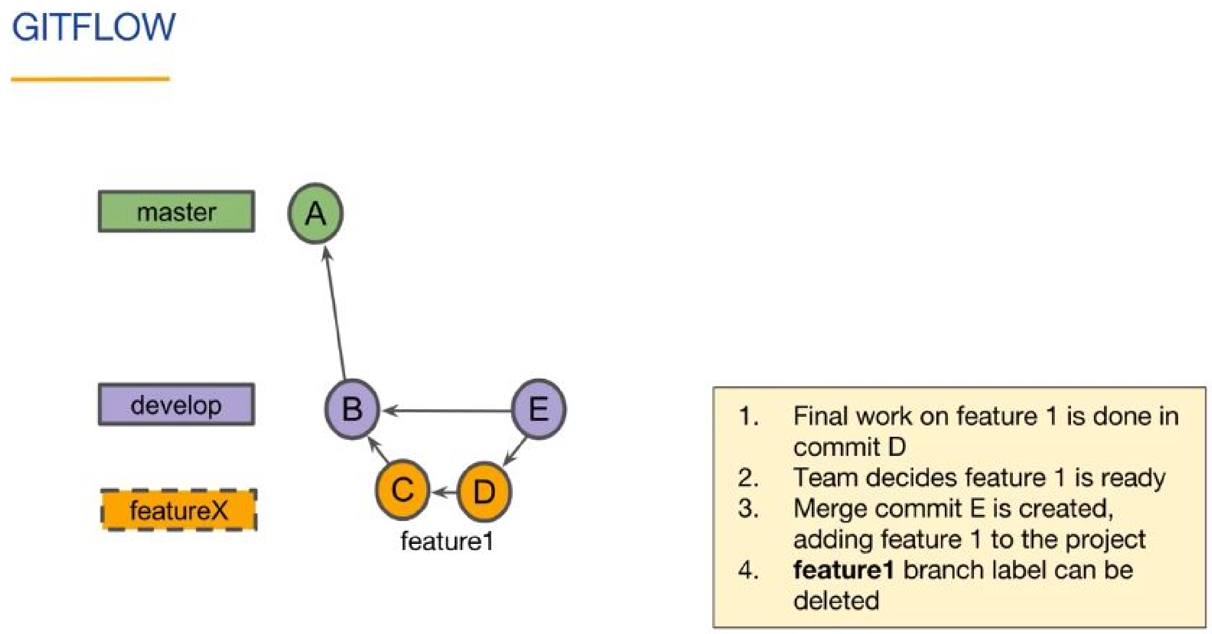
The develop branch is created off of the master branch. Commit B is the first commit on the develop branch. It too can be very simple. We have now created our two long running branches.



GITFLOW

1. Create **feature1** branch off of commit B
2. Begins work and creates commit C

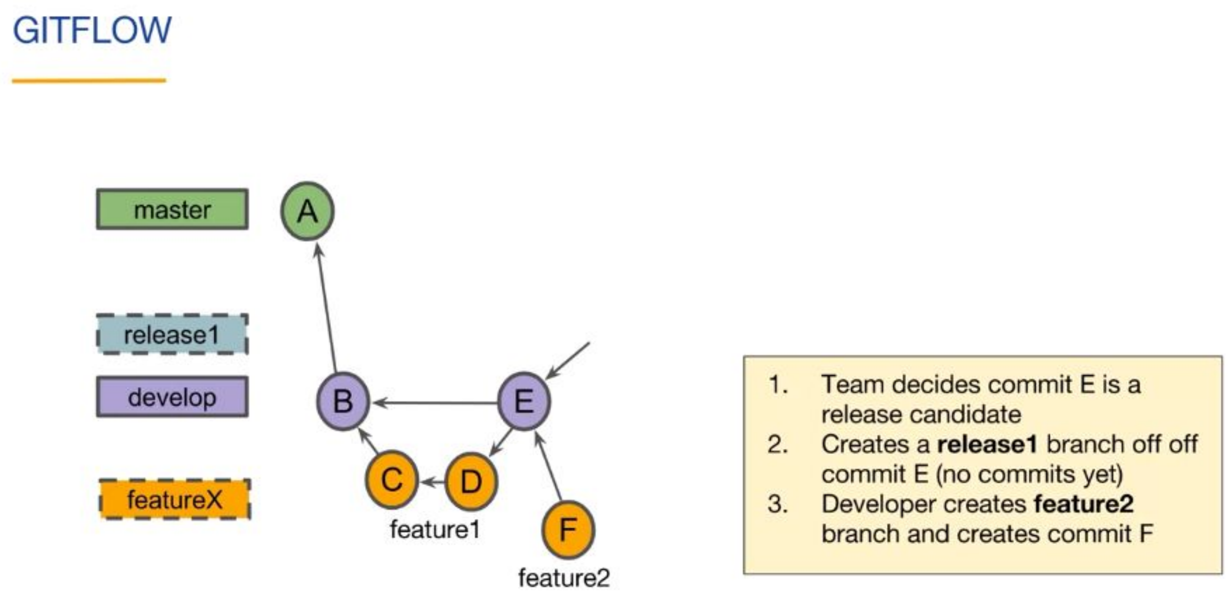
Our first release of the project is planned to have only a single feature named feature1. We create a feature1 branch, then get to work. Commit C begins the work of feature1.



GITFLOW

1. Final work on feature 1 is done in commit D
2. Team decides feature 1 is ready
3. Merge commit E is created, adding feature 1 to the project
4. **feature1** branch label can be deleted

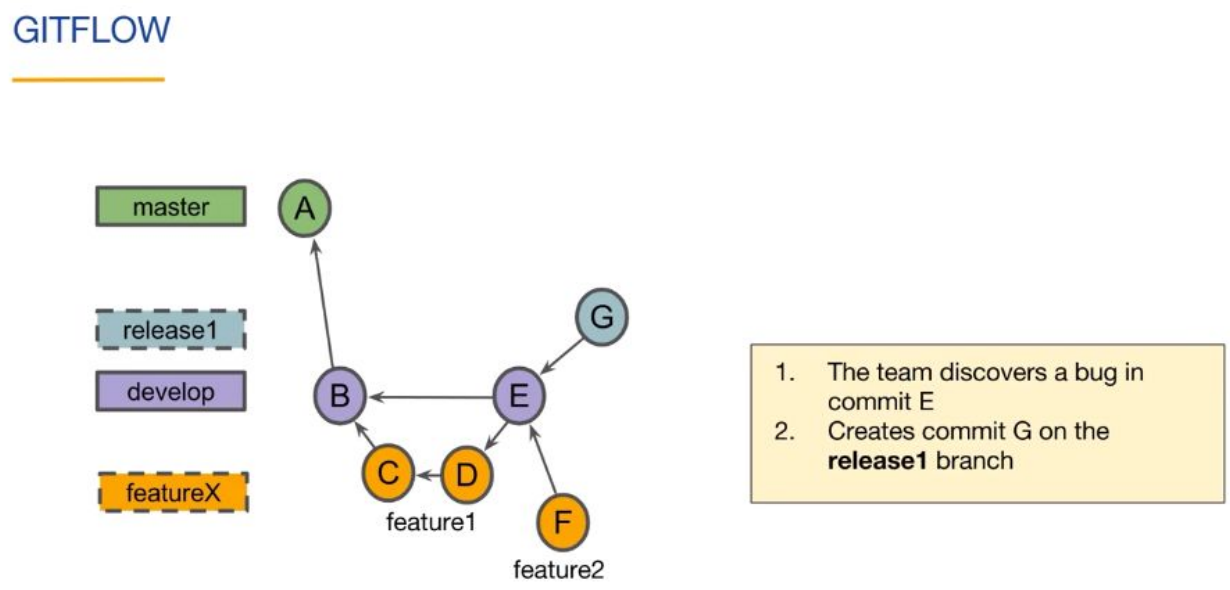
We then finish the work of feature1 in commit D. Let's assume that the team decides that feature1 is ready to be merged into the developed branch. This may have been done through a pull request. Merge commit E is created on the develop branch.
At this point, the develop branch contains feature1. Since the feature1 branch is merged, we can delete the feature1 branch label.



GITFLOW

1. Team decides commit E is a release candidate
2. Creates a **release1** branch off off commit E (no commits yet)
3. Developer creates **feature2** branch and creates commit F

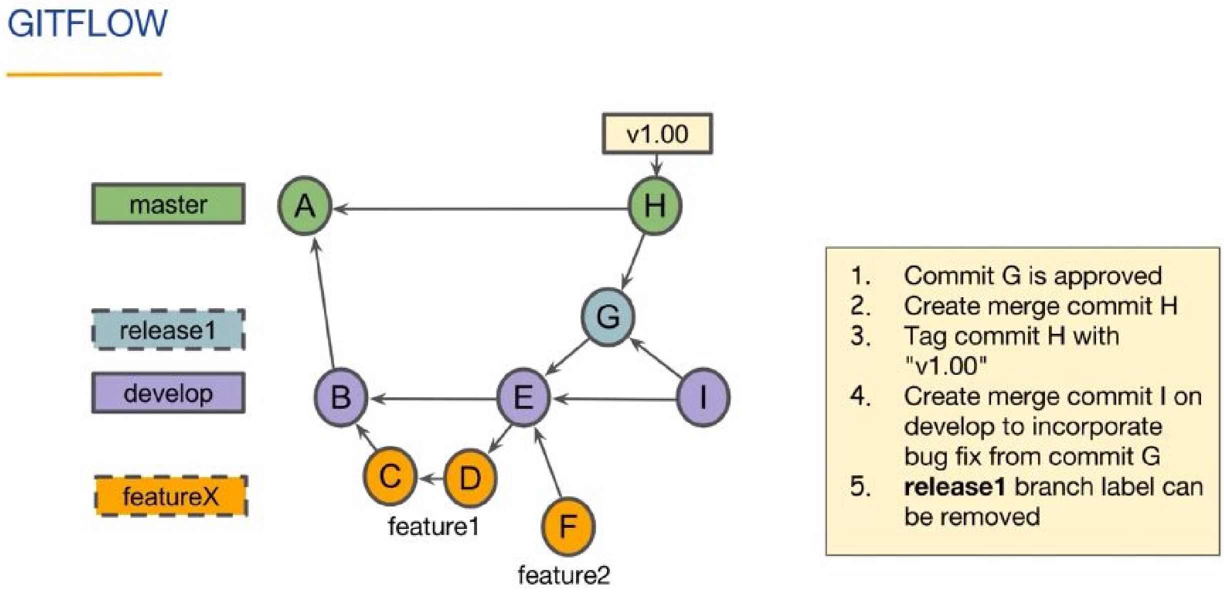
The team has decided that the first release will contain only a single feature, and that commit E represents a release candidate. A release1 branch is created off of commit E. At this point, commits intended for release1 can only be made on the release1 branch. Any commits to the develop branch will be in a future release. The commits on the release1 branch should mainly be for fixing issues with release1, not adding new features. Since the developer is done with feature1 and knows of no issues with the release, they create a feature2 branch and begin working on feature2. This work is for a later release of the project. Commit F contains some of the work of feature2.



GITFLOW

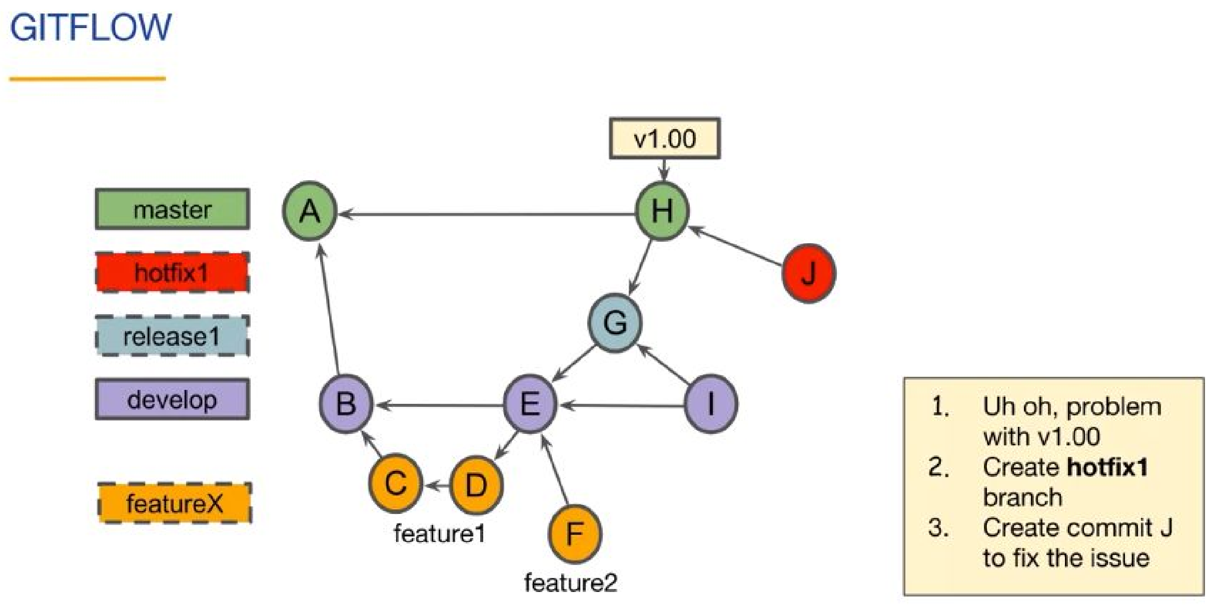
1. The team discovers a bug in commit E
2. Creates commit G on the **release1** branch

The team has discovered a bug in Commit E. Commit G is added to the release1 branch. This commit fixes the bug.



1. Commit G is approved
2. Create merge commit H
3. Tag commit H with "v1.00"
4. Create merge commit I on develop to incorporate bug fix from commit G
5. **release1** branch label can be removed

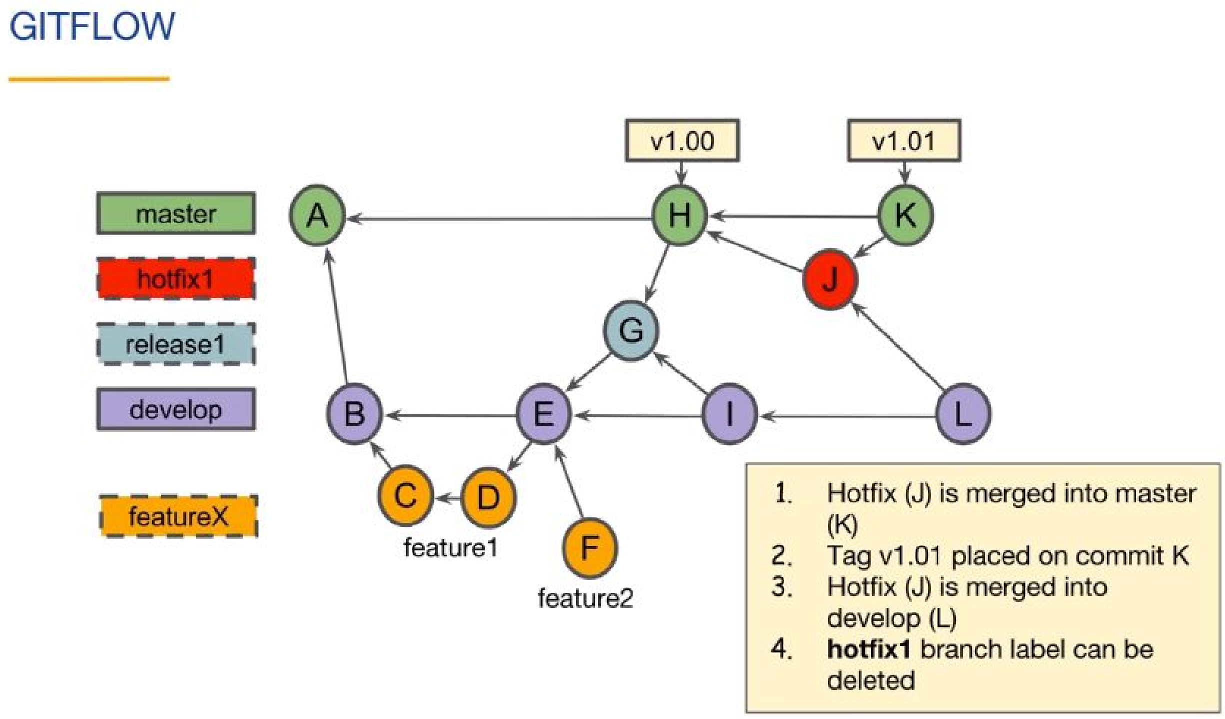
As far as the team is concerned, commit G is ready for release to the master branch. Merge commit H is created. This contains feature1. Commit H is tagged with a version 1.0 label so that team members can easily find released versions of the project. When you commit to the master branch in a GitFlow workflow, it's important to consider how that commit impacts other branches. Commit G should be merged into the develop branch, which we do here with commit I. If we didn't do this, the bug fix from commit G would not be in the develop branch. This means that the bug would come back in the next release. Because we have merged the release1 branch to both master and develop, we can now delete the branch1 label. Release branches are only valid for a single release.



GITFLOW

1. Uh oh, problem with v1.00
2. Create **hotfix1** branch
3. Create commit J to fix the issue

Let's say that we are working away on feature2, and then then someone reports a bug with our version 1 release. We create a hotfix1 branch to deal with this problem. And create commit J which fixes the bug. We do not branch off of the develop branch because new commits could have been made to the develop branch for the next release. We don't want those features in our hotfix. We want to make the change to the release as small as we can to limit our risk while still fixing the bug.

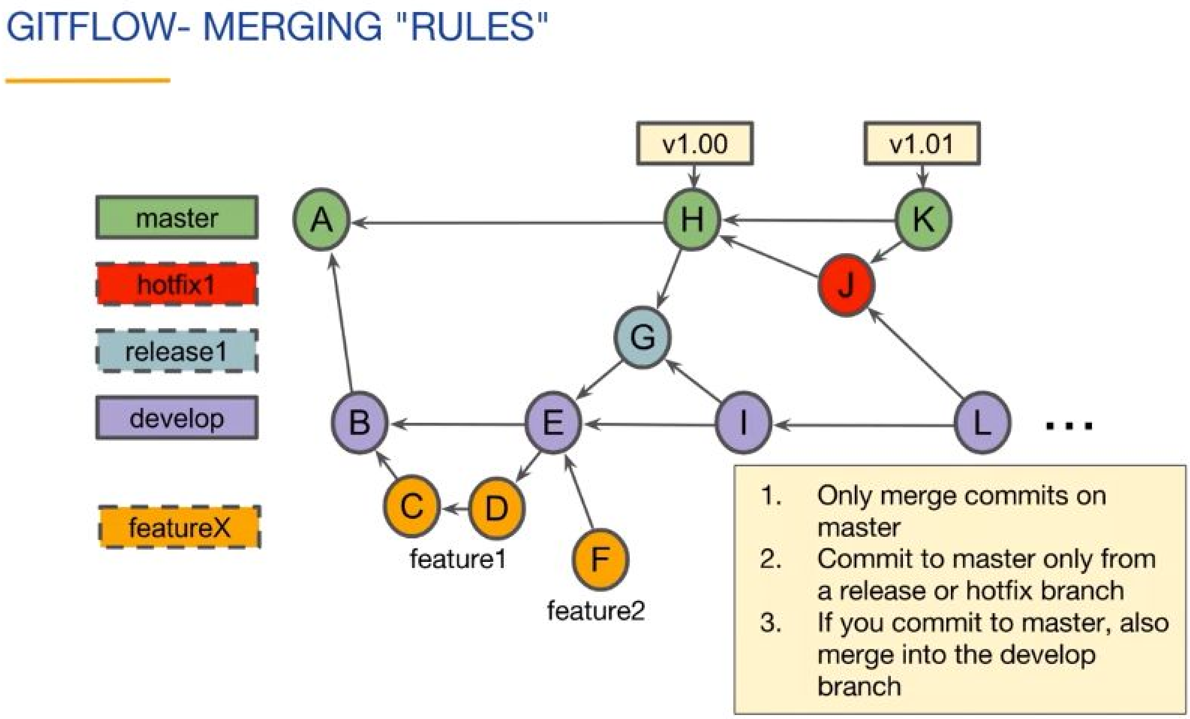


GITFLOW

1. Hotfix (J) is merged into master (K)
2. Tag v1.01 placed on commit K
3. Hotfix (J) is merged into develop (L)
4. **hotfix1** branch label can be deleted

Hotfix commit J is then merged into master with commit K. This fixes the bug. We tag commit K with version 1.01. This represents the latest release of the project. Because we have committed to master, we need to also commit to the develop branch. Merge commit L is then created and sharing that the hotfix is also included in the develop branch. A hotfix branch is created for a single hotfix and once changes are merged to the master and developed branches, the hotfix1 branch label can be deleted.

At this point, we can continue work on feature2 and any other work planned for the next release. This process continues indefinitely, allowing for a continuous improvement of the project.



GITFLOW- MERGING "RULES"

1. Only merge commits on master
2. Commit to master only from a release or hotfix branch
3. If you commit to master, also merge into the develop branch

We have used some rules with our GitFlow workflow. Teams may have different rules. One rule is to only use merge commits on master. You can see that, with the exception of Commit A, no work is done directly on master. Commits H and K are merge commits. You can tell that they are merge commits, because they have have multiple parents. Another rule is that the commits to master can only come from release or hotfix branches.
You should not directly merge from the develop or feature branches. This helps ensure proper testing and quality for the release. You can see here that commit H merges in the release1 branch and commit K merges in the hotfix1 branch. The third rule is, if you commit to master, also merge into the develop branch.
If you don't do this, issues that were fixed will reappear in future releases. You can see here that when we merged commit G into master, we also merged it into the develop branch with merge commit I. If we didn't do that, the bug fix in commit G, will not be in our future

versions. Similarly when we merged hotfix J into master, we also merged it into the develop branch in merge commit L.