

# How Most People use git

(<https://xkcd.com/1597/>)



If that doesn't fix it, `git.txt` contains the phone number of a friend of mine who understands git. Just wait through a few minutes of 'It's really pretty simple, just think of branches as...' and eventually you'll learn the commands that will fix everything.



# Xoriant Git Course

How to use Git Effectively

"I decided I could write something better than everything out there in two weeks. And I was right." –Linus Torvalds

# Objectives

---

- ▶ How to use Git effectively for source control
- ▶ How to use basic Git commands **from the command line**
- ▶ How to use different workflows with Git (If time permits)
- ▶ Resources to learn more about Git



# Not Objectives

---

- ▶ Cover all Git commands
- ▶ Cover all options of some commands
- ▶ Make you a Git expert (though we will point you to the right resources)



# Introduction

---

- ▶ What is Git?
  - Git is a Distributed Version Control System(DVCS)
- ▶ What does a Version Control System do?
- ▶ What are other VCS's?
  - Visual Source Safe
  - CVS
  - SVN
  - Mercurial
  - Bazaar

Why distributed?



# How we start with version control

---

C:\work\my\_project

C:\work\my\_project\_old

C:\work\my\_project\_old\_old

C:\work\my\_project\_old\_8Jan2016

C:\work\my\_project\_old\_10Jan2016

C:\work\my\_project\_old\_12Jan2016

Using Git will save us a lot of time AND disk space

# 1. Setup

---

## ► Setup Name and Email

```
git config --global user.name "Your Name"  
git config --global user.email your\_email@xoriant.com
```

## ► Make a directory called **work**

## 2. Start a Project

---

- Initialize a Git repository

In your working directory, make an empty directory called hello.

```
cd hello
git init
git status
```

Make a file called hello.html using notepad++ and save it in the hello directory. Add this as the content of hello.html

```
<!DOCTYPE html>
```

```
<html>
```

```
  <body>
```

```
    Hello World
```

```
  </body>
```

```
</html>
```

```
git status
```

```
git add hello.html
```

```
git commit -m "my first commit"    (These are regular double quotes. PPT
mangles them)
```

```
git status
```

```
git branch
```





# Terminology

---

- ▶ The Git Directory (This directory might be hidden. You will need to do `ls -a` to see it or in Windows Explorer you will need to turn on the option to see hidden files and folders)
  - The Git directory is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.
- ▶ The Working Directory
  - The working directory is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.
- ▶ The Staging Area
  - The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit. It's sometimes referred to as the "index", but it's also common to refer to it as the staging area.



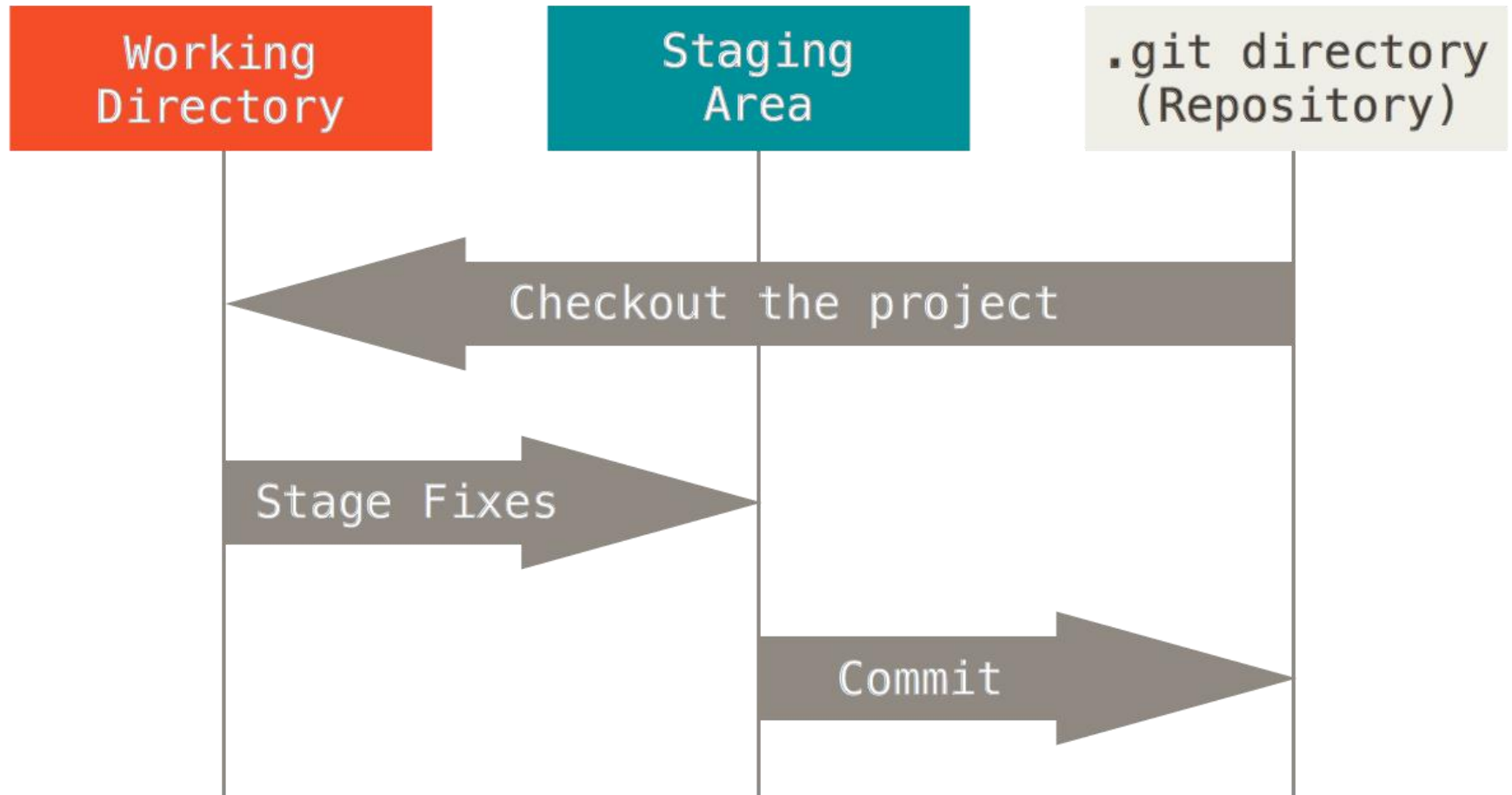
# Basic Git Workflow

---

- 1) You modify files in your working directory.
- 2) You stage the files, adding snapshots of them to your staging area.
- 3) You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory



# Basic Git Workflow



### 3. Making changes

---

Make changes in the `hello.html` file and save the changes.

```
git status
```

Make a file called `hello_again.html` using notepad++ and save it

```
git status
```

```
git add hello_again.html
```

```
git commit -m "my second commit"
```

```
git status
```

You will notice that the changes to `hello.html` are not committed because it wasn't added to the staging area after changes.

```
git log
```



## 4. Staging and Committing

---

Make a new file called hello3.html

```
git add .  
git status
```

The `.` is a shortcut that says add all files in this directory and below. This particular command will add the unstaged hello.html (from earlier) as well as hello3.html to the staging area. Now make a change in the file hello3.html. Add a line and save it for example.

```
git status
```

Now there will be the same hello3.html file visible as staged as well as unstaged. If you commit, only the staged version will be committed

```
git add .  
git commit
```



## 4a. Ignoring files (.gitignore)

---

The purpose of gitignore files is to ensure that certain files not tracked.

A .gitignore file has one pattern per line.

Patterns can have filepaths and wildcards

Examples

```
# Ignore configuration files that may contain  
sensitive information.
```

```
sites/*/settings*.php
```

```
# Ignore paths that contain user-generated content.
```

```
sites/*/files
```

```
sites/*/private
```

```
Files created by Git remain untracked.
```



## 5. History

---

To see history, you can see use the following command

```
git log
```

The command comes with many options

```
git log --pretty=oneline
```

```
git log --pretty=oneline --max-count=2
```

```
git log --pretty=oneline --since="5 minutes ago"
```

```
git log --pretty=oneline --until='5 minutes ago'
```

```
git log --pretty=oneline --author=<your name>
```

```
git log --pretty=oneline --all
```

## 6. Aliases

Sometimes its useful to have shortcuts for frequently used commands or long command line options.

This is done by adding aliases to the .gitconfig file in your home directory. In Windows it is usually C:\users\username

Try adding the following in the .gitconfig file. Again this might be a hidden file. Find it and change this . The last line hist is useful and is used later in the tutorial. So add it exactly as given

```
[alias]
```

```
co = checkout
```

```
ci = commit
```

```
st = status
```

```
br = branch
```

```
hist = log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short
```





## 6. Aliases

---

```
hist = log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short
```



## 7. Changing a commit

---

Once you make a commit and you realize you left out something. You forgot something in the file or you realized that the commit message wasn't correct. Git allows you to edit your previous commit – either change or add files that you left out and change the commit message also .

Change a file, stage it and commit it.

```
git hist
```

Make another change and then

```
git add hello.html
```

```
git commit --amend -m "oops forgot to add this line  
in the commit"
```

```
git status
```

```
git hist
```



# Undoing changes

---

There are methods to undo your changes

- Before staging by using  
git checkout
  - ▶ touch hello.txt
  - ▶ git commit -am "add hello world"
  - ▶ echo "hello world" > hello.txt
  - ▶ git status
  - ▶ git checkout
  - ▶ git status

Undo the changes on a file from the last commit



# Undoing changes

---

- After staging by using  
git reset
  - ▶ Make some changes to 'hello.txt'
  - ▶ git add hello.txt
  - ▶ git status
  - ▶ git reset
  - ▶ git status

Unstage the staged files



# Undoing changes

---

- After committing by using  
git revert
  - ▶ git commit -am 'make some changes'
  - ▶ git log
  - ▶ git revert head
  - ▶ git log

Creates another commit that is an inverse of the commit that you reverted. Does not change history but adds a commit on top.



# Undoing changes

---

- After committing by using  
git reset --hard <commit hash>

Moves the head to the mentioned commit. **Do not do this on a shared remote repository.**

# Branching

---

What is a branch?

Why do we need branches?

A branch represents an independent line of development.

You can think of them as a way to request a brand new working directory, staging area, and project history.

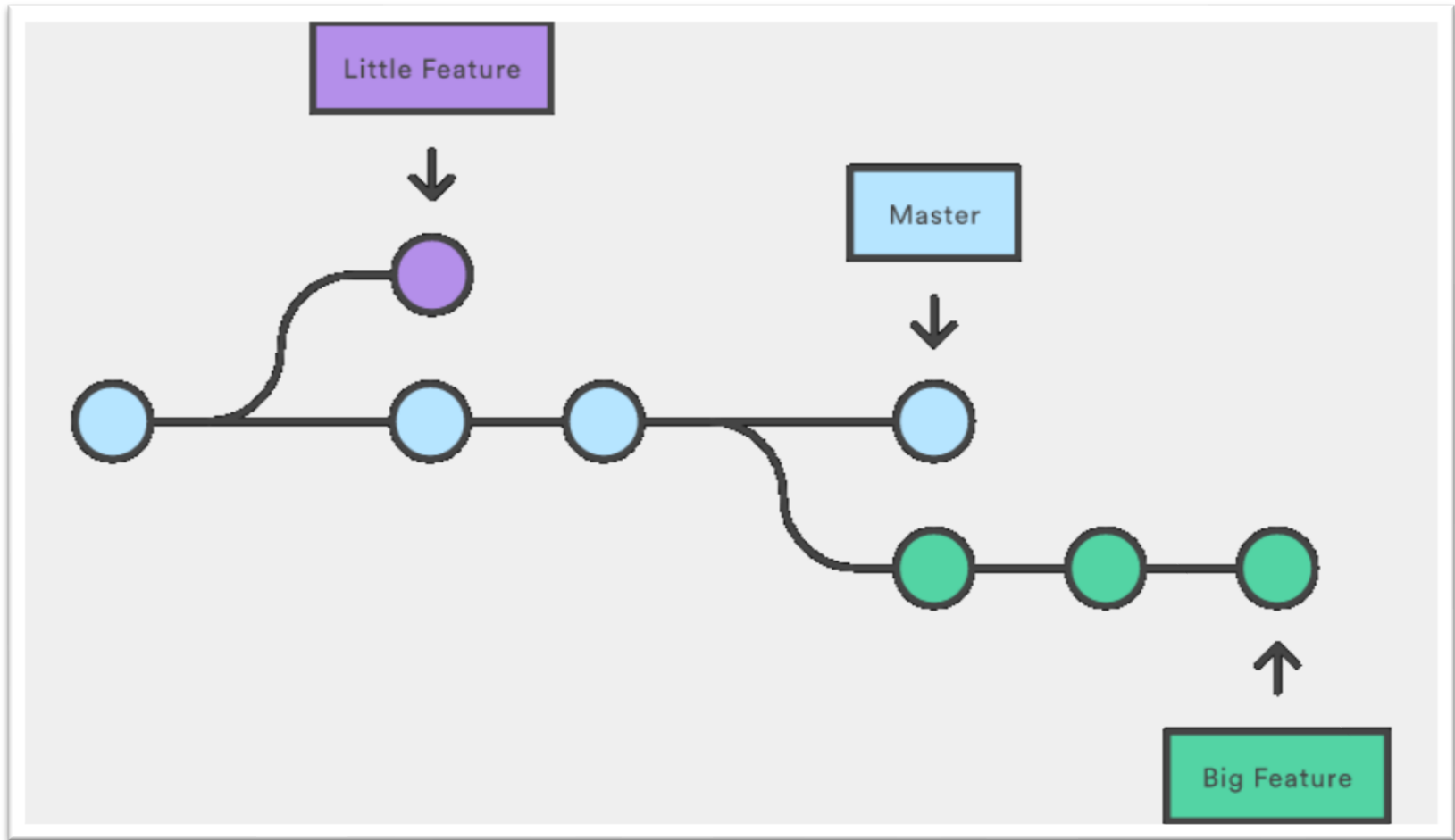
You can make changes, discard changes, make commits, record history , all without losing your original work .

A typical scenario is keeping your production code on master branch and each feature on a separate, new branch

Another scenario can be production code on master, staging code on a different branch and development branch on a different one



# Branching Diagram





## 8. Branching commands

---

To create a new branch

```
git branch my-new-branch
```

To list all branches

```
git branch
```

To move to a branch

```
git checkout my-new-branch
```

To create a new branch and move to it (short-cut for the two steps above)

```
git checkout -b my-new-branch2
```

To move back to master

```
git checkout master
```



# Branching Exercise

---

Create a new branch called dev-branch

```
git checkout -b dev_branch  
git branch
```

A new branch has been created and you are now checked out on the new branch.

Open the existing file hello3.html and make a change. Create a new file hello4.html. Add both to the staging area and commit both.

```
git status  
git add .  
git commit -m "checking how branches work"
```

Now navigate back to the master branch. What happens to the file? Navigate back to the dev-branch. What happens now? If you notice, the working tree changes to reflect the branch

# 10. Merging

---

You have a development branch. This got tested and deployed to production. How do you now bring your master branch up to date?

This can be done with a merge or a rebase. We will discuss merging now and rebasing later.

Remember the steps for merging

- 1) Move to the main branch
- 2) Merge the side branch into the main branch.
- 3) After the side branch has been merged, you can delete it .

## 10. Merging (contd.)

---

```
git checkout master  
git merge dev-branch
```

```
git branch -d dev-branch
```

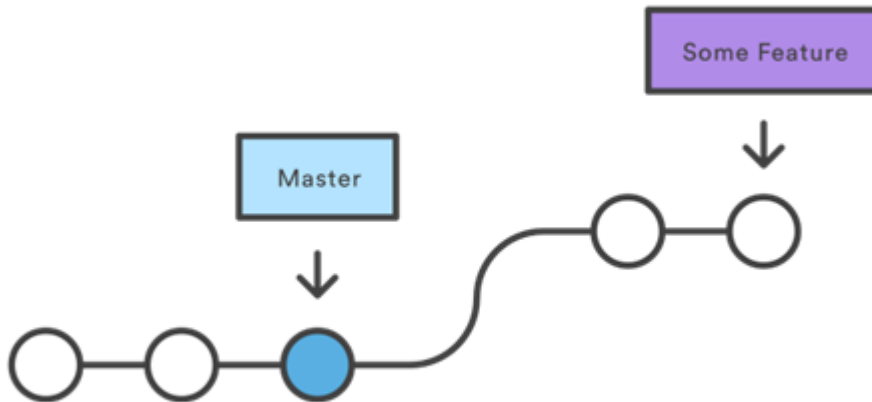
If you try to delete an un-merged branch, Git won't let you...unless you force it with a -D

# 10. Types of Merging

## (Fast Forward Merging )

When the parent branch hasn't moved ahead from when you branched off.

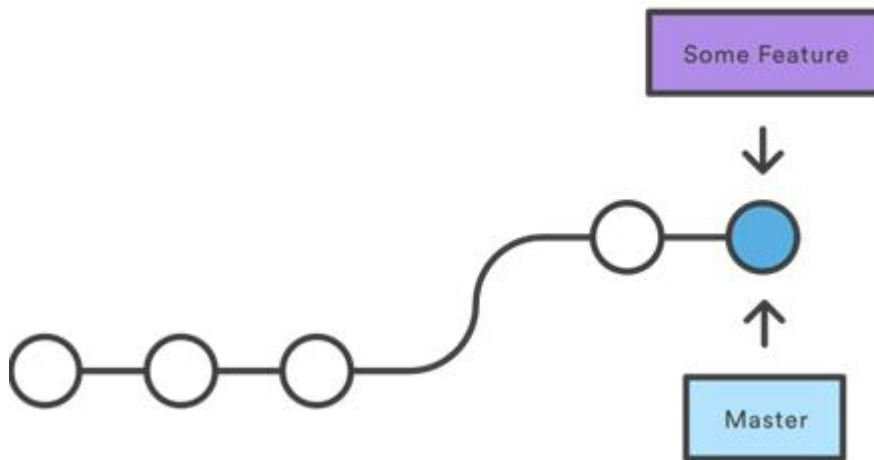
Before Merging



## 10. Types of Merging (Fast Forward Merging )

# Fast Forward Merging

### After a Fast-Forward Merge



# 10. Types of Merging

## (True Merging or 3-way merging)

True Merging – When the parent branch has moved ahead.

### Case 1. Different Files changed

Make a new branch called new-branch3 and navigate to it. How?

```
git checkout -b new-branch3
```

Add a new file hello-newbranch.html in the new branch. Stage and commit it.

In the meanwhile there needs to be an urgent commit on the production branch. So go back to the master branch and make a new file called urgent\_fix.html and add it to the staging and commit it. How?

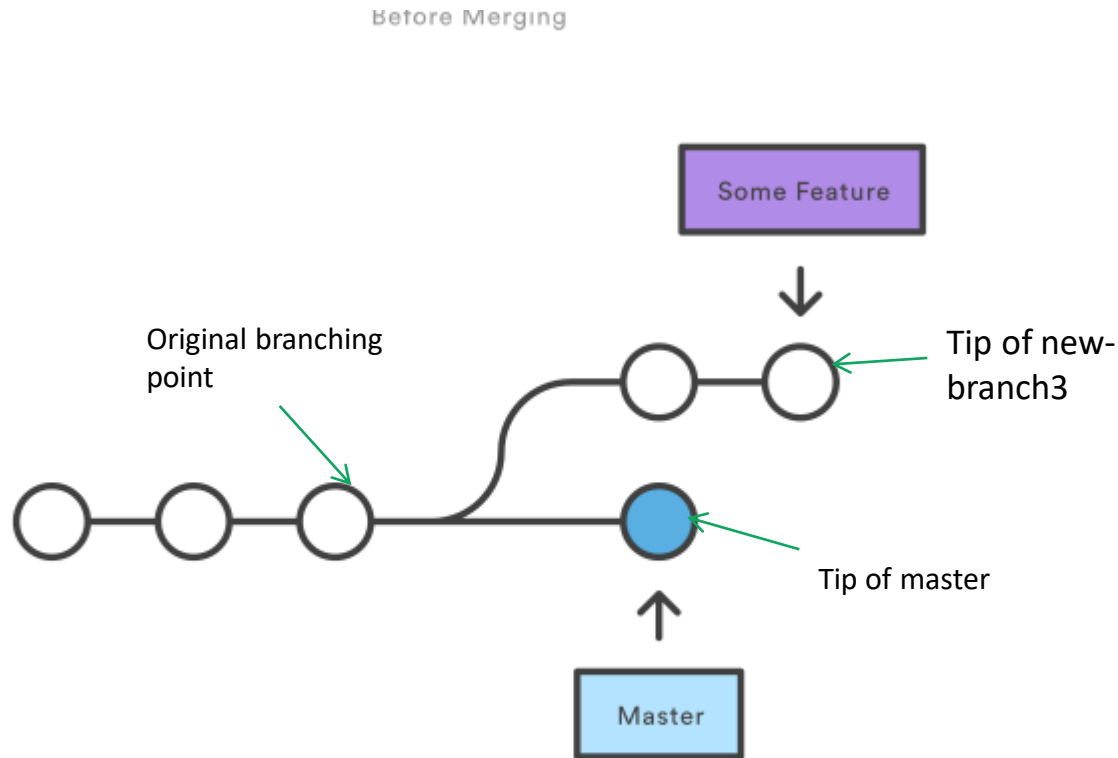
```
git checkout master
git add urgent_fix.html
git commit -m "made an urgent fix"
```

Now navigate back to the new-branch3 make another change in hello-newbranch.html. Stage and commit. Now the situation looks like this next slide



# 10. Types of Merging (True Merging or 3-way merging)

## True Merging





## 10. Types of Merging (True Merging or 3-way merging)

Now that you have completed development on your new feature on new-branch3, you want to merge it back to the master branch. How will you do this?

Remember the steps for merging

- 1) Move to the main branch
- 2) Merge the side branch into the main branch.
- 3) After the side branch has been merged, you can delete it .

```
git checkout master  
git merge new-branch3
```

And optionally

```
git branch -d new-branch3
```

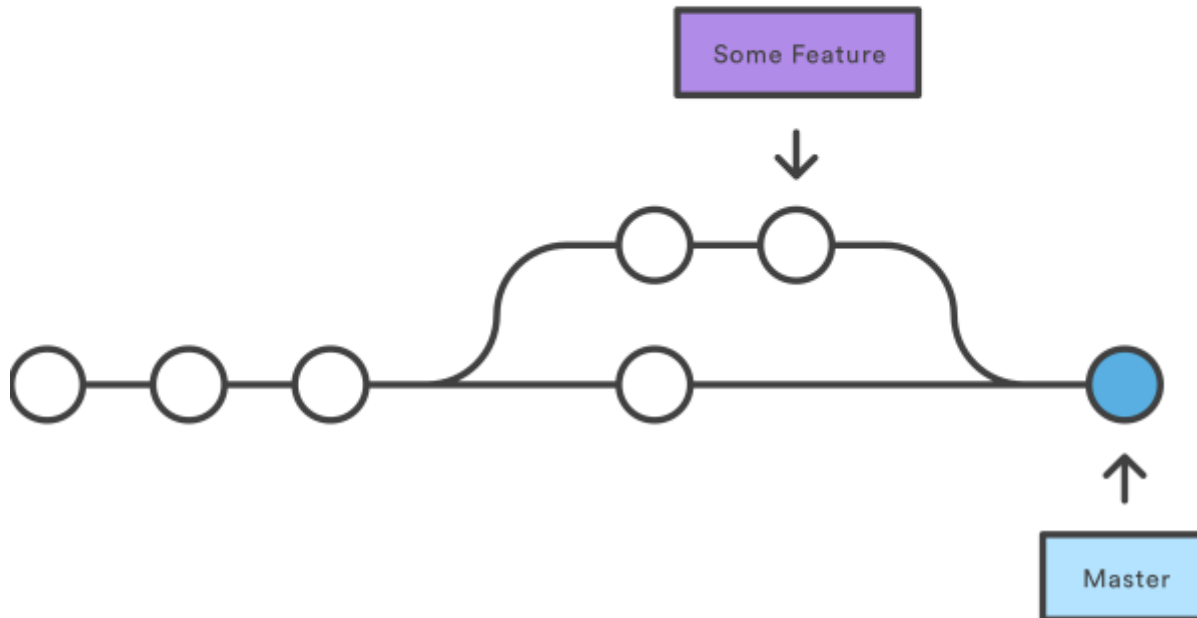
What actually happened is graphically displayed in the new slide



# 10. Types of Merging (True Merging or 3-way merging)

## True Merging

After a 3-way Merge



# 10. Types of Merging

## (True Merging or 3-way merging)

True Merging – When the parent branch has moved ahead.

Case 2. Same File has changed but in different lines

Make a new branch called new-branch4 and navigate to it. How?

```
git checkout -b new-branch4  
git branch
```

Change our original hello.html in the new branch in a part in the beginning of the file. Stage and commit it.

```
git add hello.html  
git commit -m "added a new feature"
```

Now checkout to the master branch

```
git checkout master
```

and change hello.html again but at the end of the file.

```
git add hello.html  
git commit -m "made another urgent fix"
```

Now navigate back and forth between the two branches and see how the file hello.html changes in your editor. Now try to merge the two branches. Do you remember the steps for a merge?



## 10. Types of Merging (True Merging or 3-way merging)

---

- 1) Move to the main branch
- 2) Merge the side branch into the main branch.

```
git checkout master
```

```
git merge new-branch4 -m "merging both changes in the  
same file"
```

Now examine the file in your editor. Changes from both branches should be automatically merged.

# 11. Merge Conflicts

Case 3. Same File has changed but in the same line

Make a new branch called new-branch5 and navigate to it. How?

```
git checkout -b new-branch5
git branch
```

Change our original hello.html on the 10<sup>th</sup> line. Stage and commit it.

```
git add hello.html
git commit -m "added a new feature on line 10"
```

Now checkout to the master branch

```
git checkout master
and change hello.html again but again on the 10th line.
git add hello.html
git commit -m "refactored on line 10"
```



# 11. Merge Conflicts (contd.)

Now do a merge the same old way.

```
git checkout master
git merge new-branch5
```

Git doesn't know which version to keep. So it will throw up a conflict like this.

```
Auto-merging hello.html
CONFLICT (content): Merge conflict in hello.html
Automatic merge failed; fix conflicts and then commit the result.
```

Open the file in your editor. You will see conflict markers. Which look like this

```
<<<<<< HEAD
Lines added in master
=====
Lines added in branch
>>>>>> new-branch5
```

One line marks what the line in the HEAD (or tip of the master is) and the other line shows you what is the version on the branch.

You edit the file manually. Keep the line that should be there. Remove the conflict markers from the file.



# 11. Merge Conflicts(contd.)

To resolve the conflict

- 1) Manually edit the file and keep the line that should be there.
- 2) Remove the conflict markers  

```
<<<<HEAD  
=====  
>>>>new-branch5
```
- 3) Do this for every conflicting part marked by the conflict markers.
- 4) Once this is complete. Stage the file again  

```
git add hello.html
```
- 5) Commit the file  

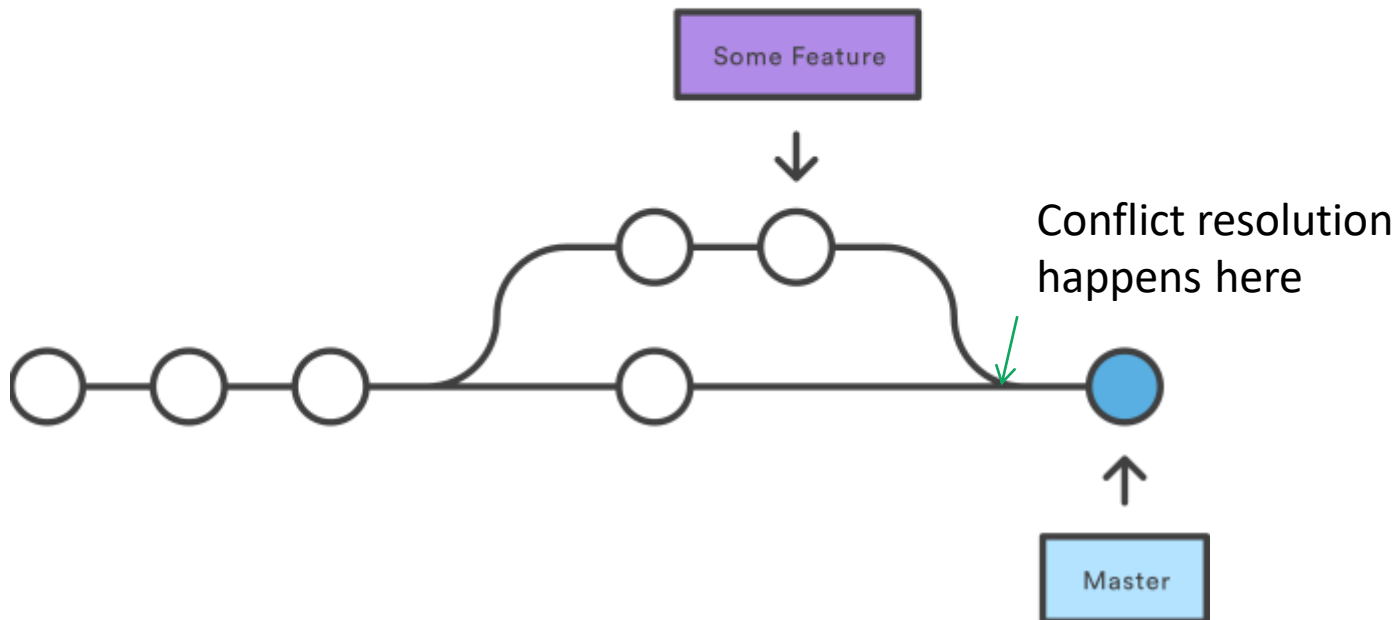
```
git commit -m "new feature successfully merged"
```

For the purpose of this demonstration we are using one file only but there could be conflicts in multiple files. You will have to fix each one before you are able to merge successfully. What you are doing in each case is still as shown in the diagram below

# 11. Merge Conflicts (contd.)

## Merging after resolving conflicts

After a 3-way Merge





## 12. Rolling back a merge

---

When things go very wrong.

You attempt a merge and Git throws up a conflict. You see that there is confusion as to what part of the file to keep and what to discard. Maybe you need to discuss this with the developer whose branch you are trying to merge. In such a scenario, you can abort the merge with the following command.

```
git merge --abort
```

This will roll back the attempted merge and you can retry after resolving whatever issue you had.

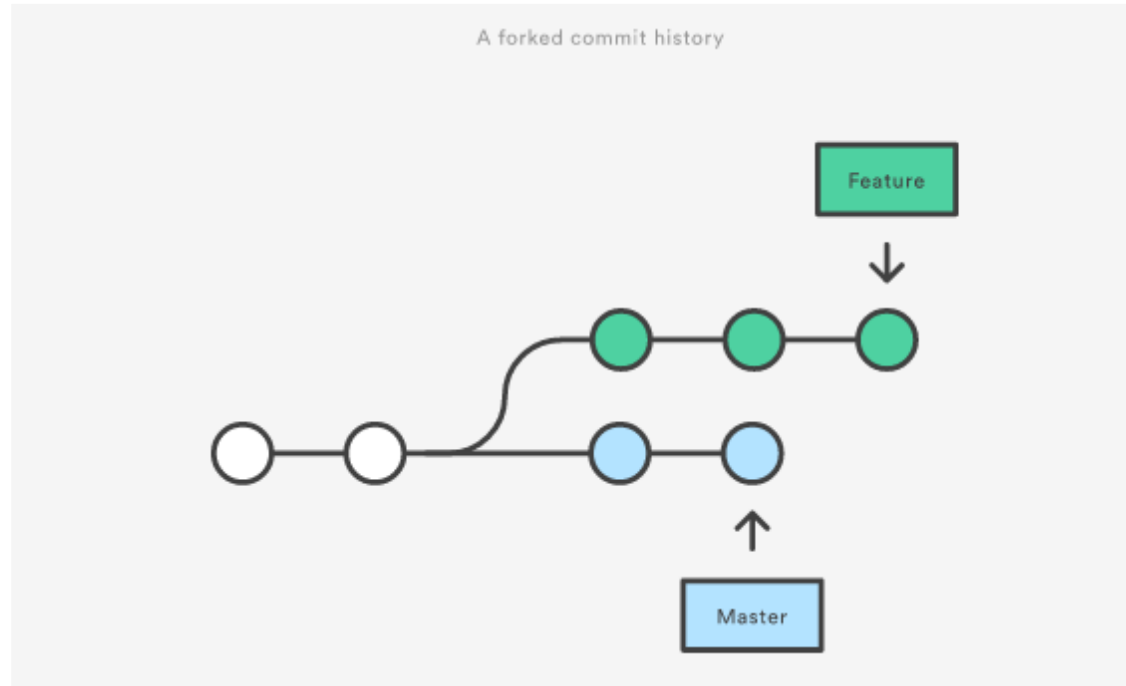
# 13. Rebasing

---

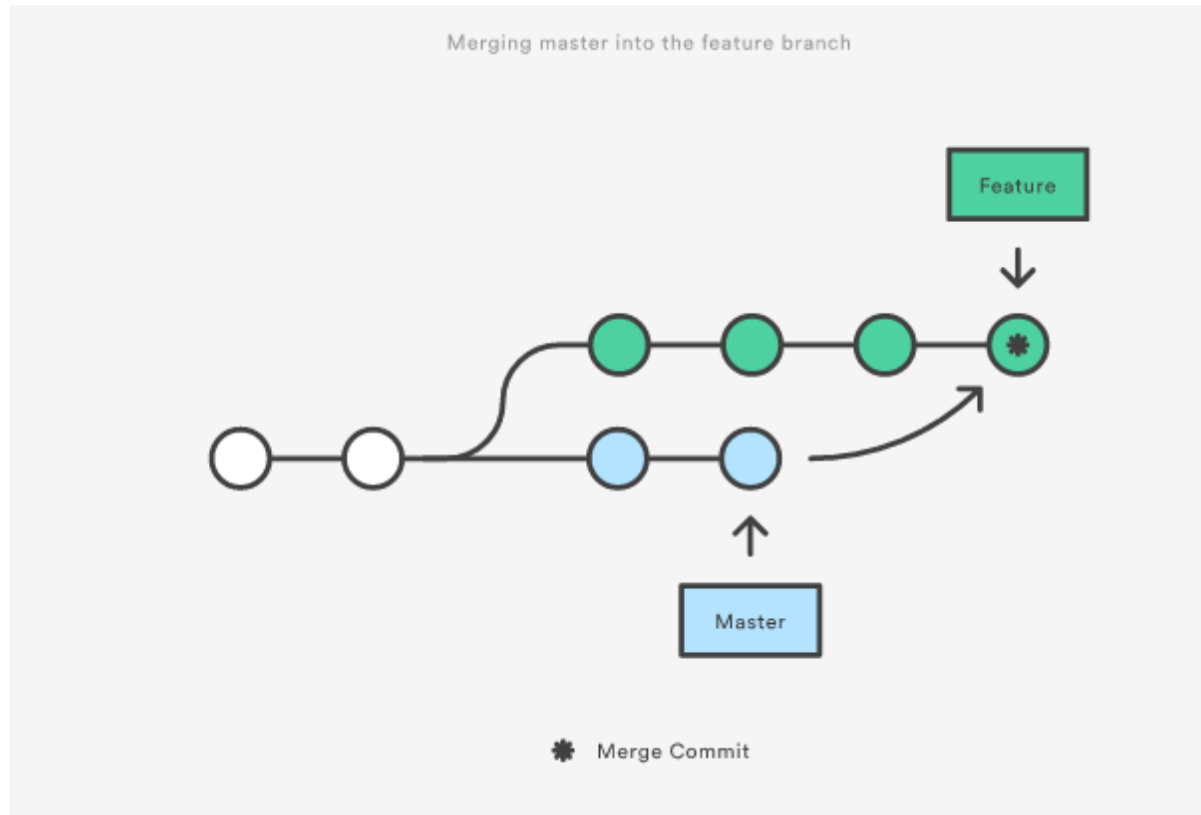
Rebasing is another way to combine your branch with the master branch.



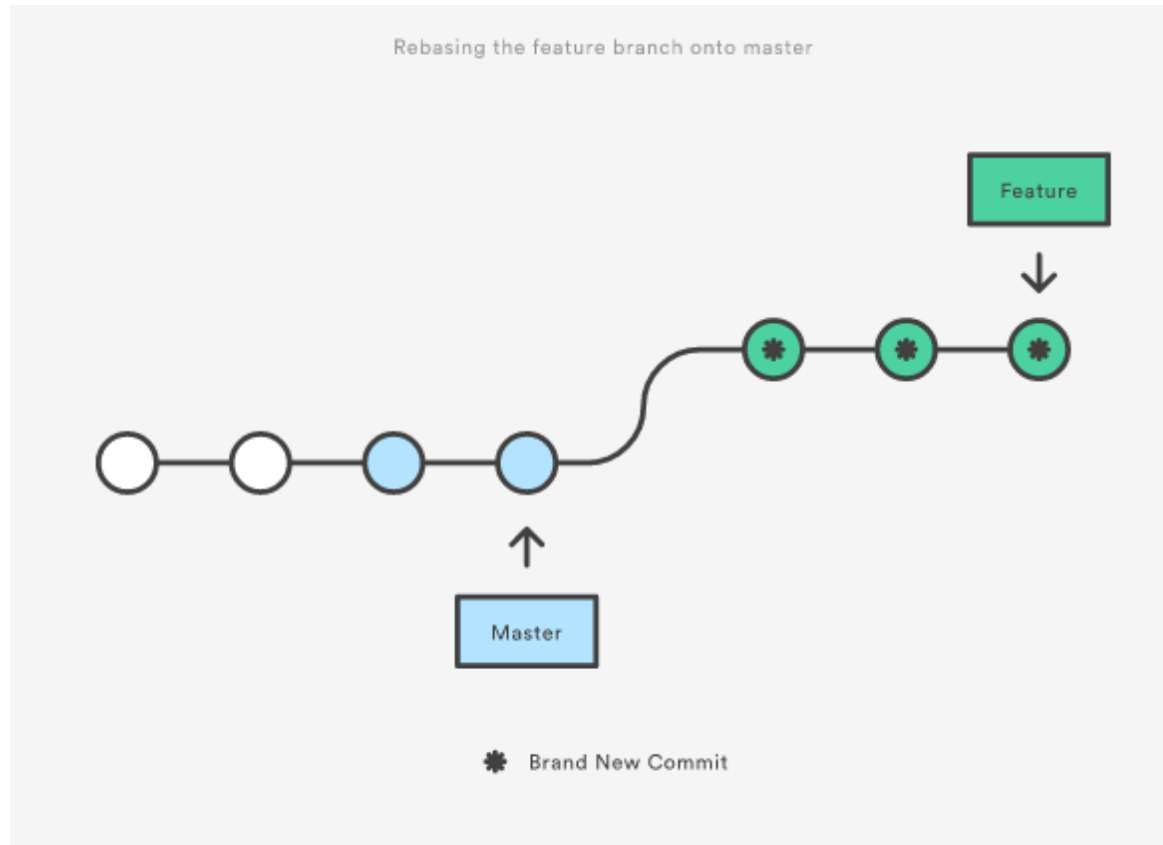
# 13. Forked Commit History



# 13. Merging



# 13. Rebasing



## 14. What happens in a rebase

---

- 1) Your current branch is rewound till the last branching point from the master and all your commits are stored in temp storage.
- 2) All the commits on master since you branched off are added in the right order
- 3) All your old commits are added after this in the right order



# 15. How to rebase

---

This is how you rebase your feature branch onto the master branch. The order of this is important. You are rebasing YOUR branch onto the MASTER branch.

This is done in the reverse way as merge.

1) Checkout your feature branch

2) Rebase your branch on master

At this point feature is like a branch from the latest commit on master. In this situation we can do a fast-forward merge.

3) Checkout master

4) Do a merge which will be a fast forward merge.

```
git checkout my-branch
```

```
git rebase master
```

```
git checkout master
```

```
git merge my-branch
```

# 16. Conflicts during rebase

---

In case a conflict occurs at any step, you can

- a) Abort the rebase `git rebase --abort`
- b) Skip the commit (very rare and dangerous to use)  
`git rebase --skip`
- c) Fix the conflict like you did during a merge conflict and then continue the rebase  
`git rebase --continue`



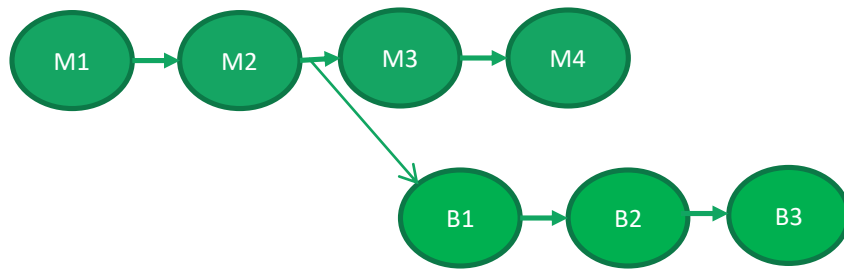


# 17. Rebase Example

Make a tree like this.

Commit your master branch with commit messages M1, M2

Then make a branch called my-branch. Make commits with messages B1, B2, B3. Checkout the master branch and make two more commits M3, M4.



Now checkout the new-branch and rebase your branch onto master

```
git checkout my-branch
```

```
git rebase master
```

```
git hist
```

```
git checkout master
```

```
git merge my-branch
```

```
git hist
```

## 18. Difference Between merge and rebase

- 1) Merging introduces a commit every time there is a merge while rebase doesn't
- 2) Rebasing allows you to have a perfectly linear history
- 3) Rebasing allows you to rewrite the history
- 4) With rebasing you can lose history while merge is safe.
- 5) Rebasing should NEVER be done on a public branch

## When do you use a rebase usually.

- ▶ To avoid many merge commits  
If the master branch is very active, and you have to keep merging, you will get a new commit for every merge. In such a case rebase would make sense.
- ▶ Rebase is also used to do a rewrite of history.  
If you had a lot of frequent and small commits on your branch and you want to combine them together before merging with master then rebase allows you to do that

## 19. Other useful git commands to know

---

- ▶ `git rebase -i master`

Interactive Rebase . Allows you to interactively change commit messages and squash commits together

- ▶ The `.gitignore` file

Tells Git which files not to track

- ▶ `git stash`

Allows you to temporarily store unstaged files before doing a checkout

- ▶ `git fixup`

Allows you to combine two or more commits

- ▶ `git reflog`

Allows you to look at how the HEAD has changed over time and various commits



## 20. Remotes

---

A remote git server is useful for collaborating between multiple developers or even syncing your own work between machines. Github and Bitbucket are two popular Git server providers.

**WARNING :-** The code you share on a Server may be public. Please be very mindful of this when using github or bitbucket. You can create private repositories but these are usually paid accounts.



## 20. Remote (Exercise)

---

Make a directory called work2

```
cd work2
```

```
git init
```

```
echo "It was a bright cold day in April" > 1984.txt
```

```
git add .
```

```
git commit -m "First draft"
```

```
create a repository on github.com
```



# Create a new repository

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

Owner

 alivenotions ▾


Repository name

/ new-repository ✓

Great repository names are short and memorable. Need inspiration? How about **didactic-engine**.

Description (optional)

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

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

☐ **Initialize this repository with a README**

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None ▾

Add a license: None ▾



**Create repository**



## 20. Remote (Exercise)

---

Push the code to the remote repo

```
git remote add origin https://github.com/username/repo.git  
git push -u origin master
```



## 20. Remote (Adding collaborators)

### Push the code to the remote repo

```
git remote add origin https://github.com/username/repo.git
git push -u origin master
```

<> Code    ! Issues 0    Pull requests 0    Projects 0    Wiki    Insights    **Settings**

Options  
**Collaborators**  
Branches  
Webhooks  
Integrations & services  
Deploy keys

Moderation  
Interaction limits

### Collaborators

Push access to the repository

This repository doesn't have any collaborators yet. Use the form below to add a collaborator.

**Search by username, full name or email address**

You'll only be able to find a GitHub user by their email address if they've chosen to list it publicly. Otherwise, use their username instead.

Add collaborator





## 20. Remote (Cloning the main repo)

Clone the main repository in which you are added as a collaborator

Branch: master ▾


New pull request

Create new file

Upload files

Find file

Clone or download ▾

 alivenotions

FIX: update license link

Latest commit 7fddba2 on Jun 18

public	Initial Setup	a month ago
src	Initial Setup	a month ago
.gitattributes	Initial Setup	a month ago
.gitignore	Initial Setup	a month ago
LICENSE	Create LICENSE	a month ago
README.md	FIX: update license link	a month ago
package-lock.json	Initial Setup	a month ago
package.json	Initial Setup	a month ago

```
git clone https://github.com/alivenotions/pidpod.git
```



## 20. Remote (Pushing Changes)

---

Move to the clone repo

```
cd work
```

Make changes in your working directory, stage and commit.

```
git add .
```

```
git commit -m "my commit"
```

One person push their changes.

```
git push
```

This will push the current branch to origin.

To push a specific branch to a specific remote

```
git push <remote-name> <branch-name>
```



## 20. Remote (Pushing Changes after the remote has moved ahead)

Everyone else try pushing their changes. You will get an error

```
! [rejected]          master -> master (fetch first)
error: failed to push some refs to
'https://github.com/mockingbirdz/gitcourse.t'
hint: Updates were rejected because the remote
contains work that you do
hint: not have locally. This is usually caused by
another repository pushing
hint: to the same ref. You may want to first
integrate the remote changes
```



## 20. Syncing with the remote (rebasing)

---

To get what changes have happened since your last commit

```
git fetch
```

Now the remote data has been fetched.

```
git branch --all
```

Shows all the branches including the remote branches.

```
git diff origin/master
```

Will tell you the difference between current and remote master

Now how do we sync it to our master?

The preferred way is to rebase

```
git rebase origin/master
```



## 20. Syncing with the remote(merging)

---

You can also merge after fetching. That will cause an additional merge commit.

```
git merge origin/master
```

A pull is a combination of a fetch and a merge.

```
git pull
```

To pull a specific branch

```
git pull <remote-name> <branch-name>
```



# 21. Workflows

---

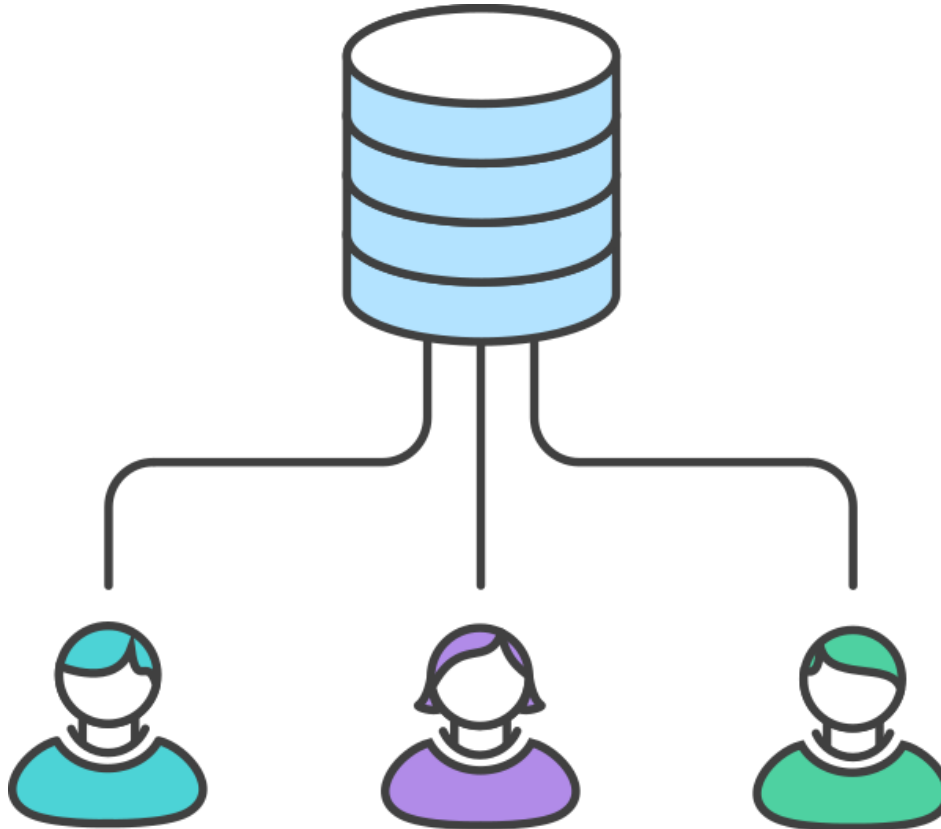
Git allows developer teams to collaborate with each other using remote servers.

Git also provides the flexibility to set certain workflows for collaboration.

This next section will explore the various possible workflows.



## 22. Centralized Workflow (diagram)



## 22. Centralized Workflow

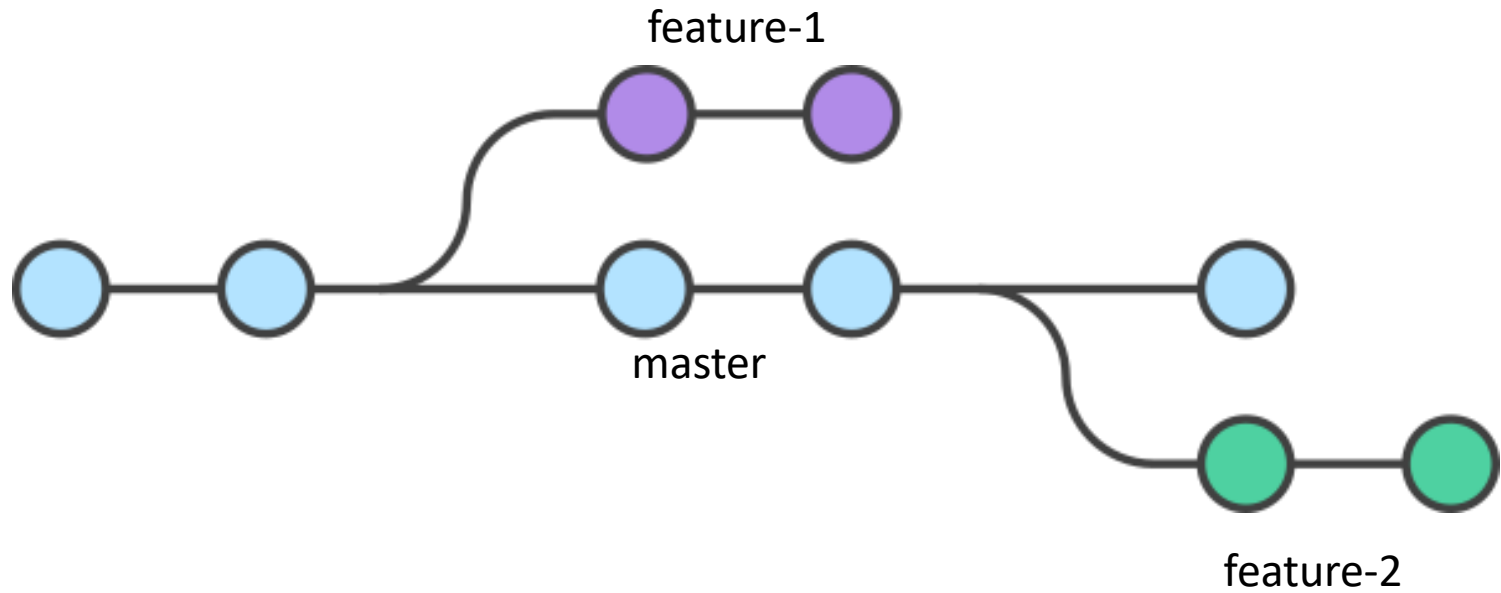
---

- 1) Each developer clones from the central repository
- 2) Each developer develops and commits locally
- 3) Each developer pushes to server
- 4) In case the central repository has moved ahead, the developer will pull or rebase and then push
- 5) Similar to SVN
- 6) Everyone has equal rights on the master repository





## 23. Feature Branch Workflow



## 23. Feature Branch Workflow

---

- 1) The master branch is the official branch
- 2) Every feature is developed on its own separate branch
- 3) Once a branch development is complete and the code is stable, it becomes merged into master
- 4) Merging is usually done with a pull request.

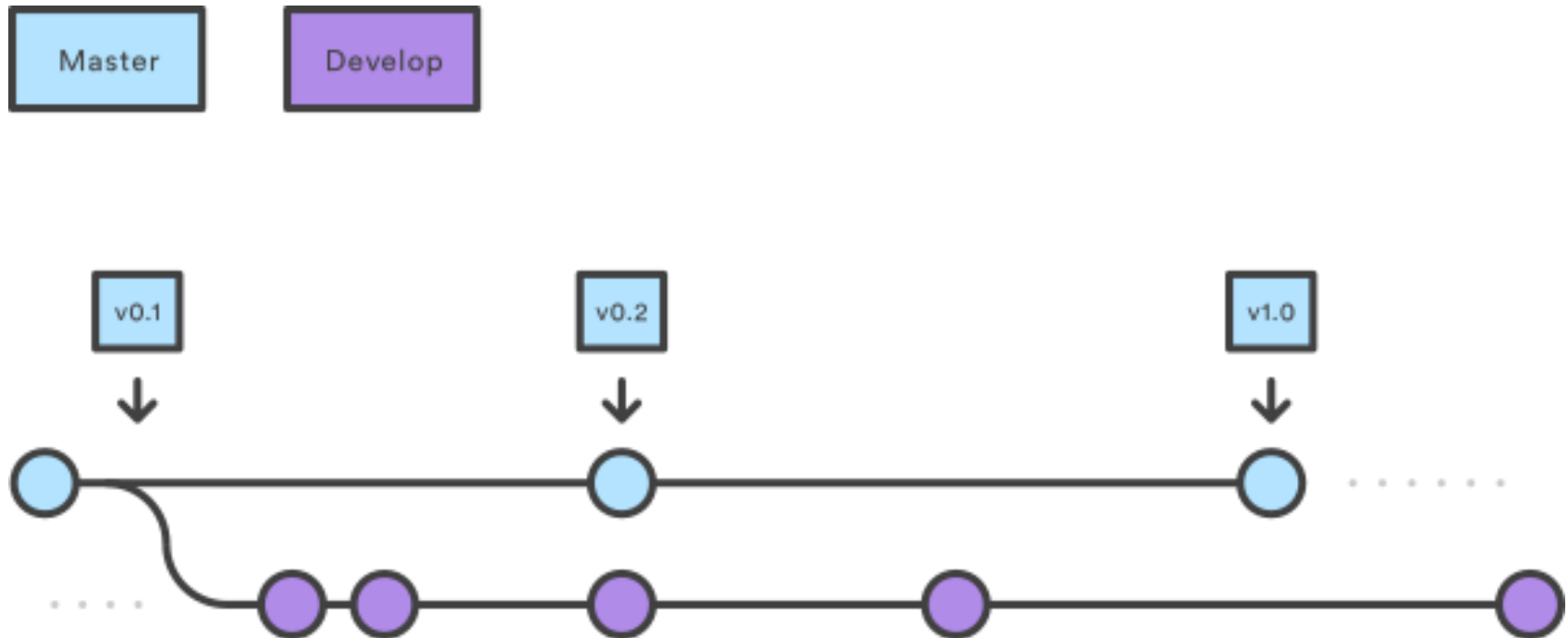
Try out a pull request.

Make a branch in your code. Make changes, stage and commit on the branch and push the branch.

Then make a pull request on Github

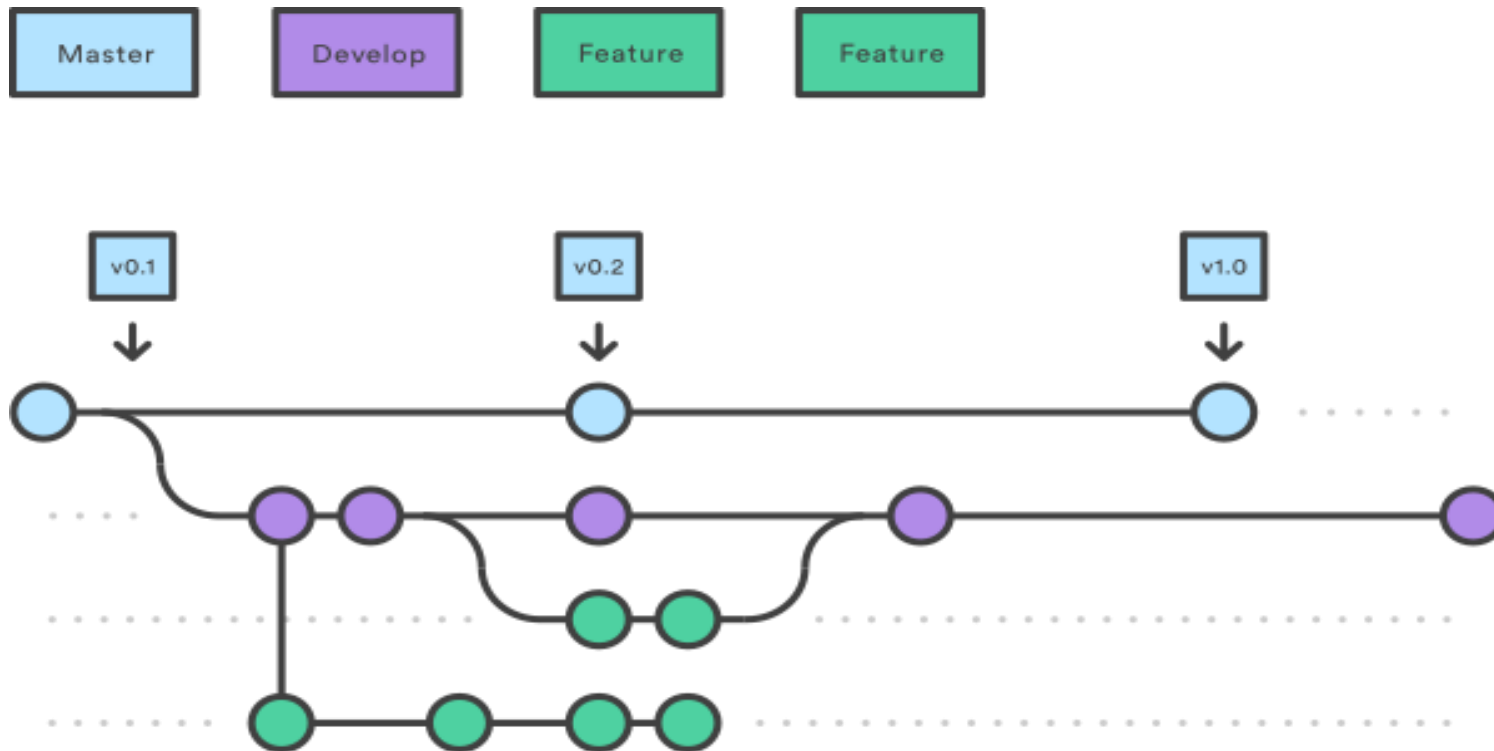


## 24. Gitflow Workflow



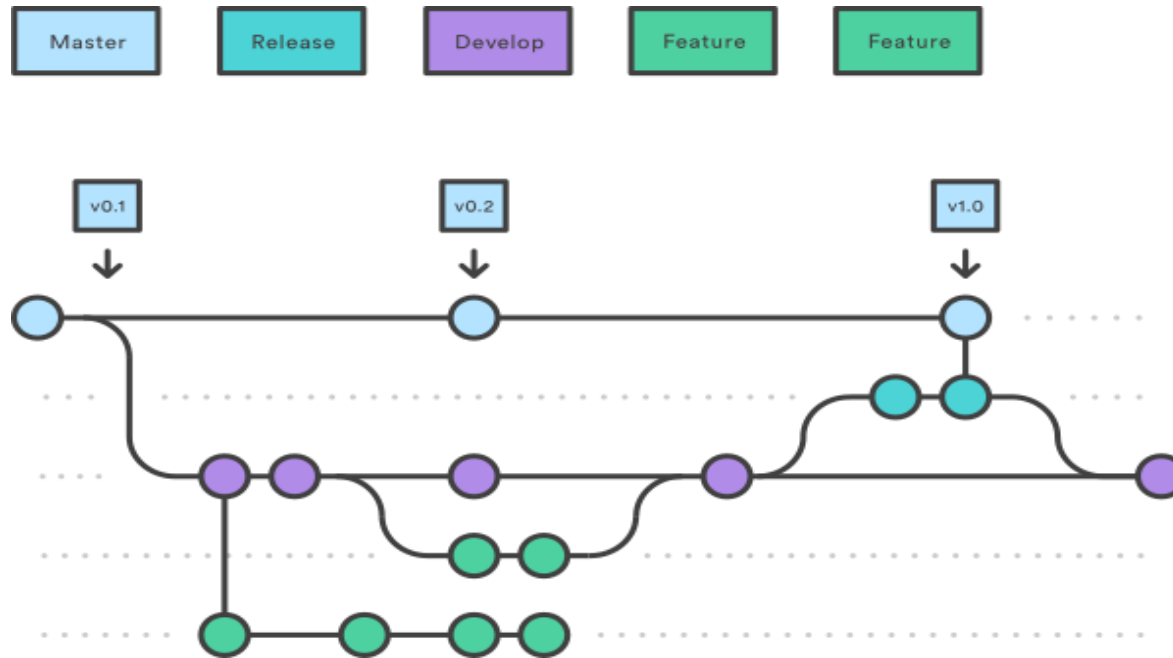
The prod branch is the master. Development happens on a release branch

## 24. Gitflow Workflow



All features are developed by branching off the feature branch and merging back once development is complete

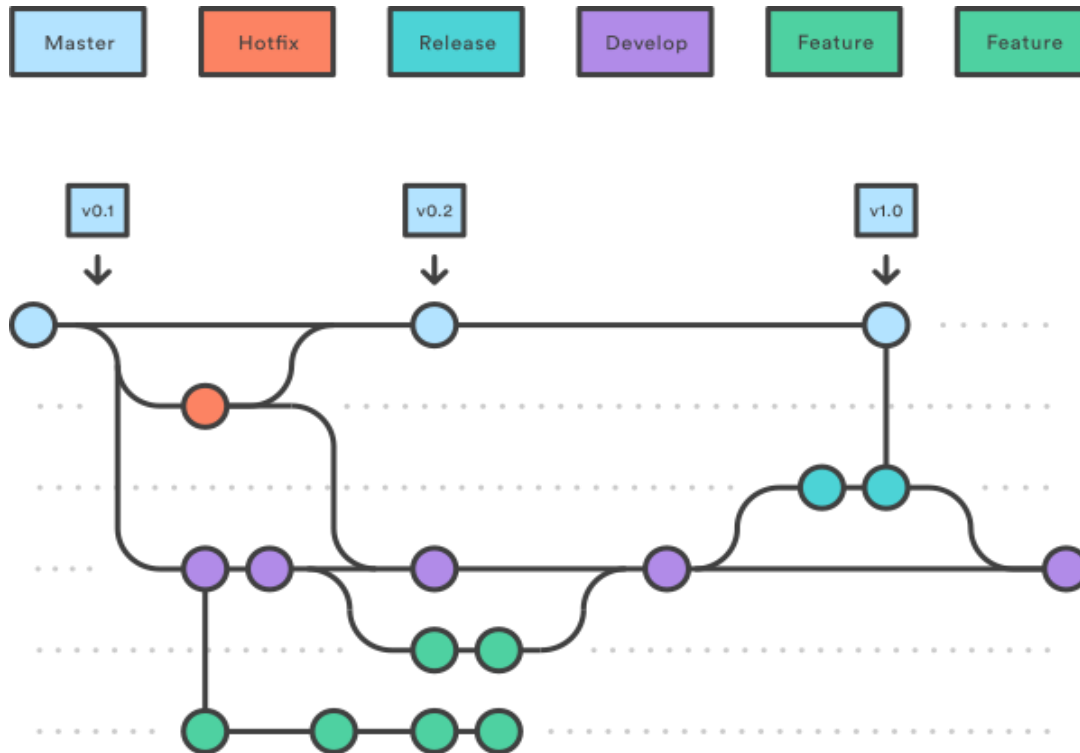
## 24. Gitflow Workflow



Once the development for the release is complete (either development is complete or date is reached), then release branch is merged back into master.

For the next release, a new branch is again created from master.

## 24. Gitflow Workflow



Any hotfixes (urgent patches) are developed by branching off master and then merging back. These are also merged into the release branch.

## 25. Useful Resources

---

Pro-Git Book

<https://git-scm.com/book/en/v2>

Atlassian's Git tutorial

<https://www.atlassian.com/git/tutorials/>

Github's Interactive Tutorial

<https://try.github.io/>

A guide for the perplexed

<http://think-like-a-git.net/>

Git Immersion

<http://gitimmersion.com/>

Git from the bottom up (a look at exactly how Git works)

<http://ftp.newartisans.com/pub/git.from.bottom.up.pdf>

Linus Torvald's TechTalk at Google on Git (purely for entertainment)

<https://www.youtube.com/watch?v=4XpnKHJAok8>

