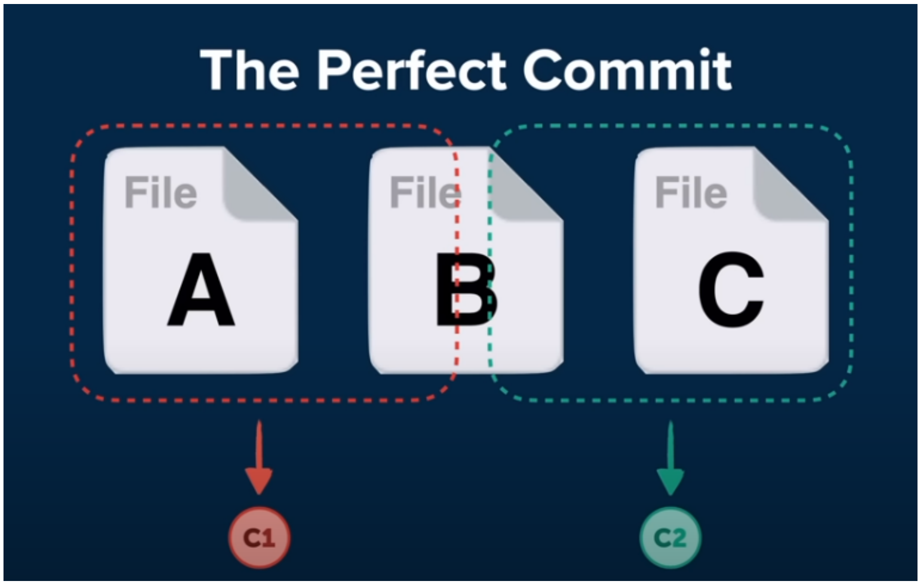
The *git add -p* command is a powerful feature of Git that allows you to interactively stage changes in your working directory for a commit. It stands for "patch" mode and provides you with the ability to review and selectively stage portions of your changes instead of adding the entire file. This is particularly useful when you have made multiple changes within a single file and want to commit them separately or want to ensure you only commit specific portions of the changes.

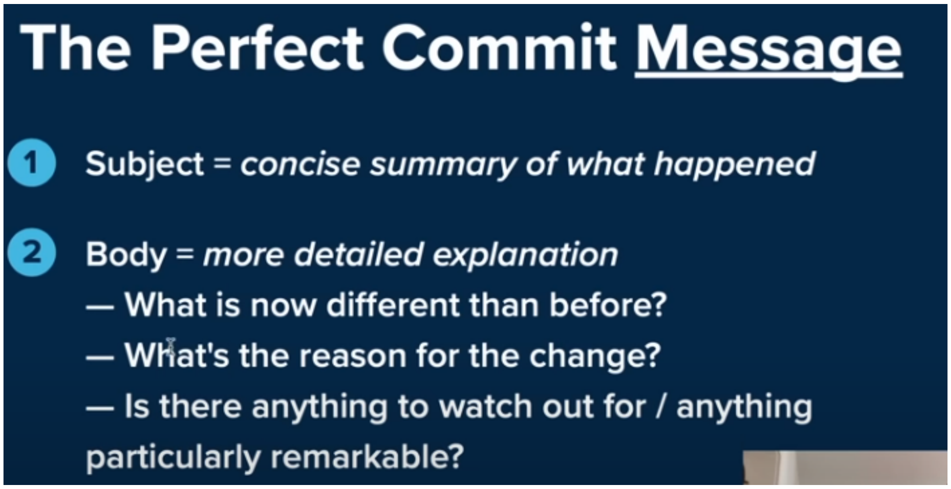


<u>Commit Messages:</u>

git commit

-> this lets you add a new commit message

-> you can write a short message, then leave a blank line and write the commit message in detail thereafter

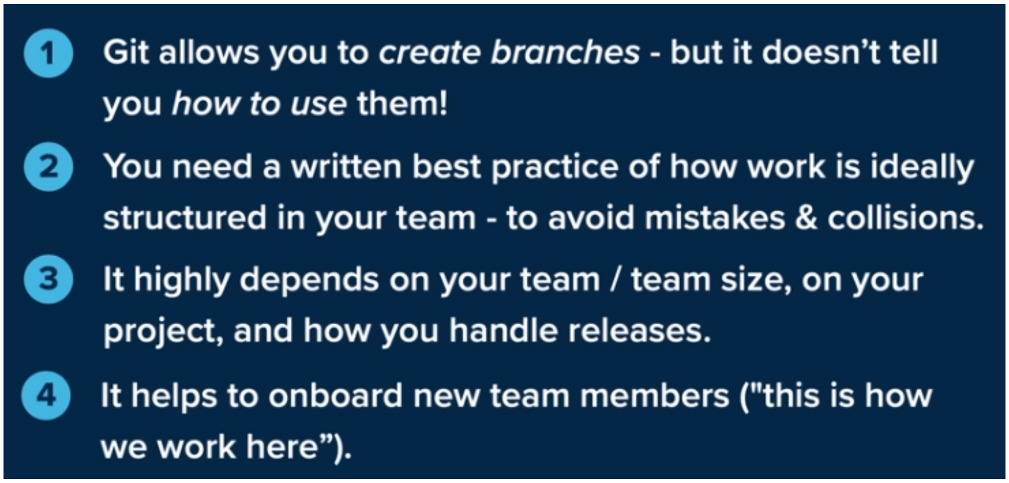


The Perfect Commit <u>Message</u>

1. Subject = *concise summary of what happened*
2. Body = *more detailed explanation*
   — What is now different than before?
   — What's the reason for the change?
   — Is there anything to watch out for / anything particularly remarkable?

-> can also use *git commit -m "your message here"*

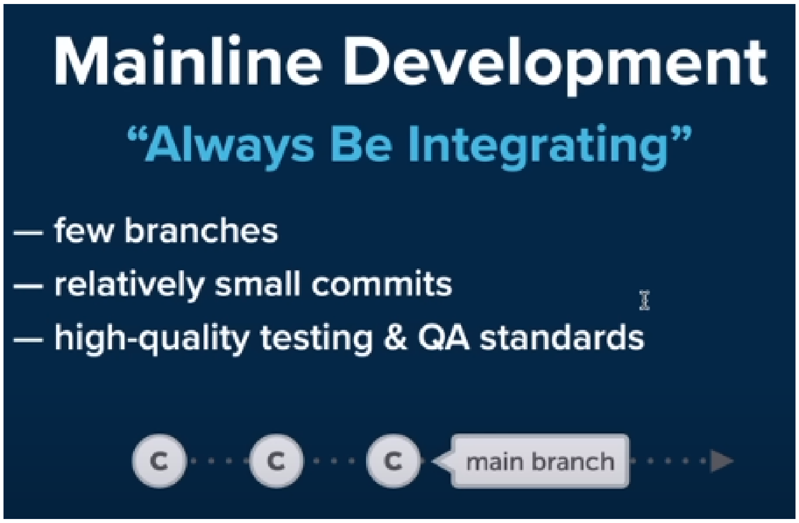
Branching:



1. Git allows you to *create branches* - but it doesn't tell you *how to use* them!
2. You need a written best practice of how work is ideally structured in your team - to avoid mistakes & collisions.
3. It highly depends on your team / team size, on your project, and how you handle releases.
4. It helps to onboard new team members ("this is how we work here").

Two strategies:

1. **Mainline Development ("Always Be Integrating")**



Mainline Development

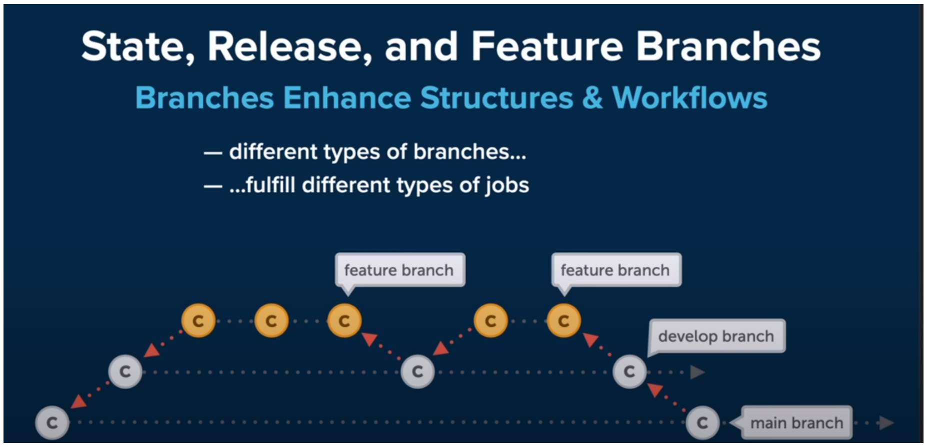
"Always Be Integrating"

— few branches

— relatively small commits

— high-quality testing & QA standards

2. **State, Release, and Feature Branches**



State, Release, and Feature Branches

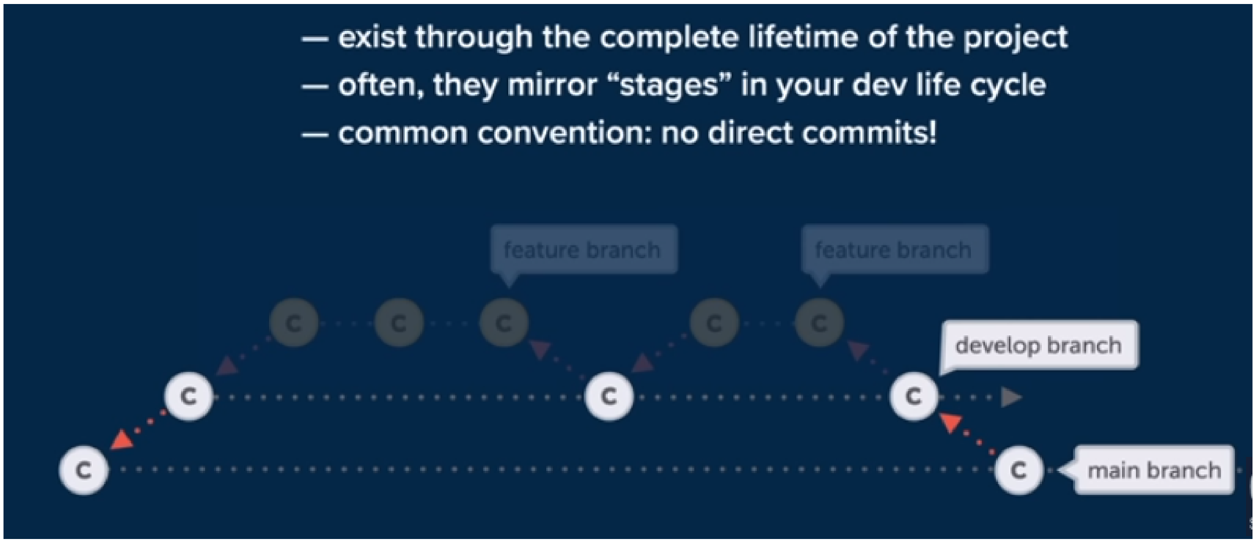
Branches Enhance Structures & Workflows

— different types of branches...

— ...fulfill different types of jobs

Two types of branches:

1. **Long Running**

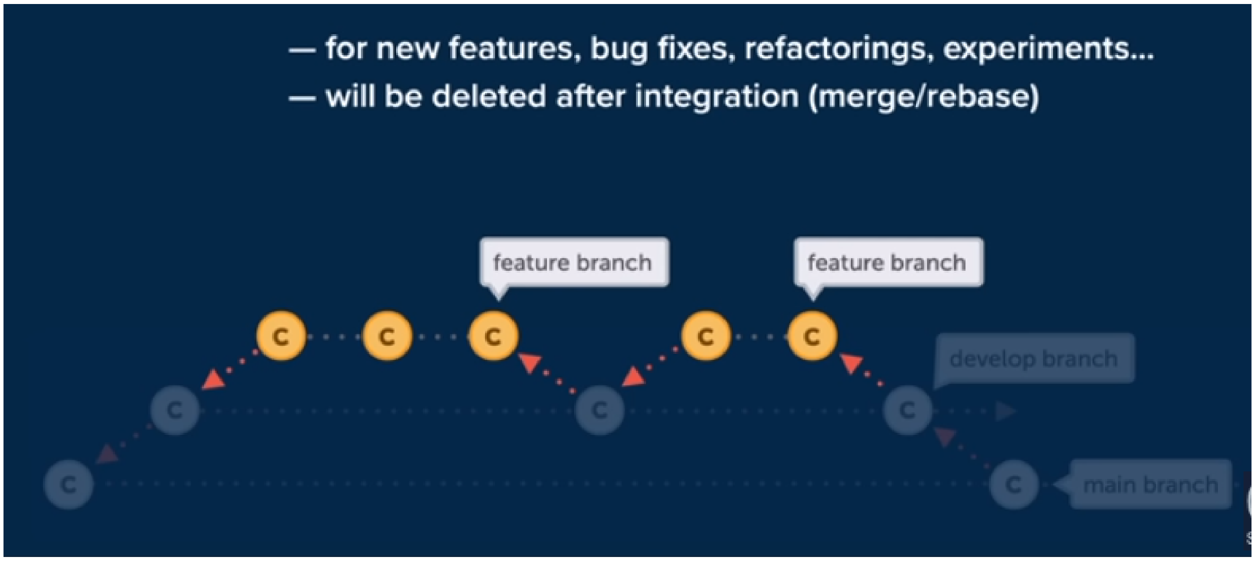


— exist through the complete lifetime of the project

— often, they mirror "stages" in your dev life cycle

— common convention: no direct commits!

2. **Short Lived**

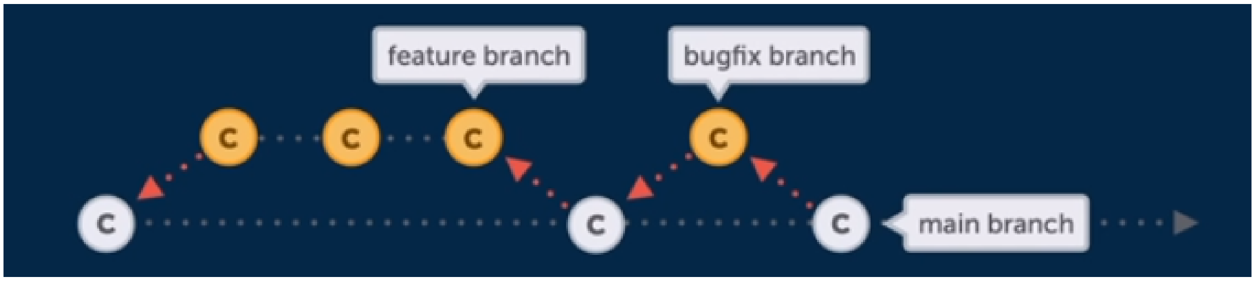


— for new features, bug fixes, refactorings, experiments...

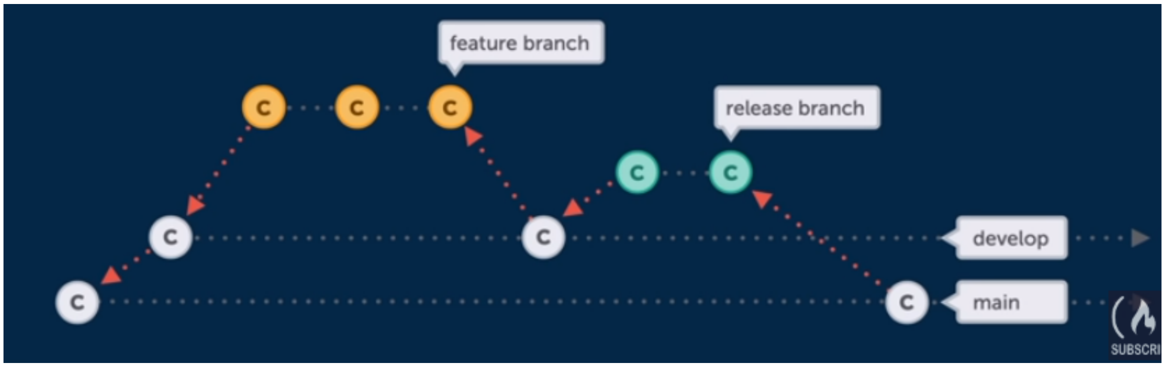
— will be deleted after integration (merge/rebase)

Two example branching strategies:

1. **GitHub Flow:**



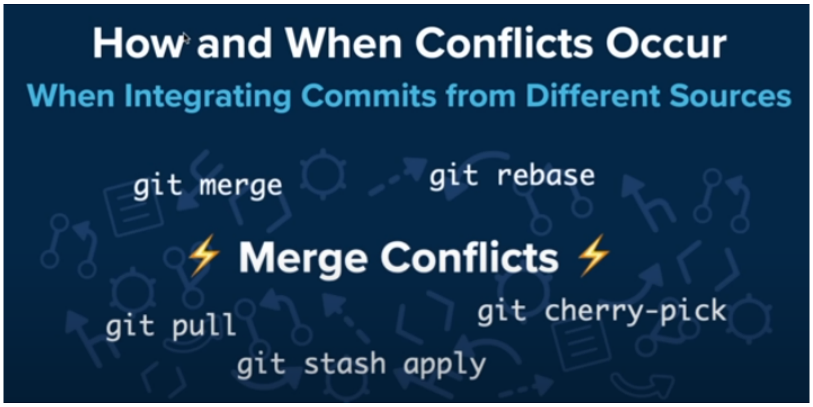
2. GitFlow:



Pull Requests and Fork:

Feature	**Pull Request**	**Fork**
Purpose	Propose changes to a repository	Create an independent copy
Relationship	Requires an existing repository	Creates a new repository
Workflow	Collaborative	Individual or Collaborative
Ownership	Original repository maintains control	Forked repository has control
Changes Visibility	Visible in the original repository	Separate from original repository
Collaboration	Contributors collaborate on changes	Independent changes can be made
Workflow Step	Comes after creating a branch	Begins with repository creation
Example	Contributor submits a PR to merge changes into original repo	Developer forks a repo to work on a feature

Merge Conflicts:



How and When Conflicts Occur

When Integrating Commits from Different Sources

git merge

git rebase

⚡ Merge Conflicts ⚡

git pull

git cherry-pick

git stash apply

How to undo a conflict and start over?

$ *git merge --abort*

$ *git rebase --abort*