Introduction to Git — Fall 2023

Lecture 5: Branches

Slides: https://hackmd.io/@git-fall-2023/L5-branches#/

Repetition

Not starting with “R”

- `checkout` - go/move HEAD to branch or specific commit (hash)
  - Applying to a file discards all unstaged changes made to the file
  - will never move the reference
- `clean` - clear **unstaged** files
  - `--dry-run` to check what will happen
- `diff` - compare
  - lists unstaged changes
  - `--staged` - commit vs staged
  - HEAD - commit vs working tree
- `log` - history of commit tree
  - `--graph` - graphically see the branches
- `show` - shows info for commit
- `stash` - temporarily **stores the staged** changes to the working tree
- `switch` - go to branch only (more clear for this use)
- `tag` - tag (like version number), **naming commits** (not branches)

Repetition

Starting with “R”

- `rebase -i HEAD~3` - interactively rebase last 3 **commits**
  - with `squash` - summarize last 3 **commits**
- `reflog` - log of commits that changes the head
- `reset` - as `checkout` but takes options to also do updates
  - `<filename>` - **unstages** the file
  - may move the reference and thereby update the branch
- `restore` - restore **file** in work dir
  - `--staged` - **unstages**
  - does not update your branch
- `revert` - makes **inverse of** the previous **commit**. The commit tree is not modified, rather two cancelling commits.
- `rev-parse --short` - find short hash for references, like `HEAD~~`
- `rm` - remove from repo

Objectives

- Get some more hands-on working with branches
  - creation
  - switching
  - merging
  - deletion
  - handling uncommitted changes
    - stashing
    - discarding
    - checkout with merge
  - merging and merge conflicts
  - rebasing: combining a sequence of commits to a new base commit.
  - cherry-picking

What is a Git branch?

- A pointer to a commit (ref: named pointer)
- Defined as all points reachable in the commit graph from the named commit (the “tip” of the branch)
- The ref HEAD determines what branch you are on.
- If HEAD is a symbolic ref for an existing branch, then you are “on” that branch.
- If HEAD is a simple ref directly naming a commit by its SHA-1 ID, you are not “on” any branch - you are in “detached HEAD” mode, which happens when you check out some earlier commit to examine.

Why use branches?

There are many uses for branches:

Why use branches?

There are many uses for branches:

- We want to develop new features, but not risk changing the working main code yet

Why use branches?

There are many uses for branches:

- We want to develop new features, but not risk changing the working main code yet
- Test different directions for a project

Why use branches?

There are many uses for branches:

- We want to develop new features, but not risk changing the working main code yet
- Test different directions for a project
- Several projects members would like to work on their own copy of the code

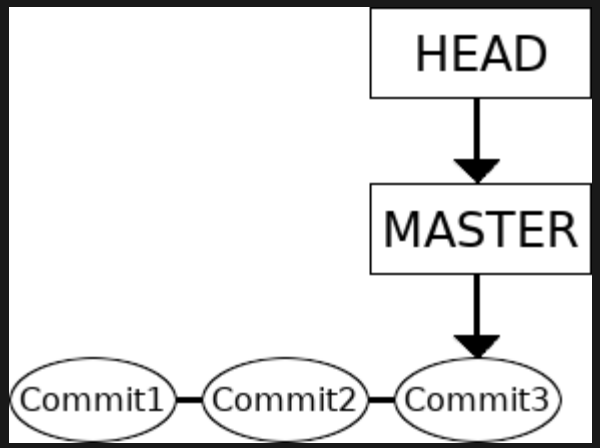
Why use branches?

There are many uses for branches:

- We want to develop new features, but not risk changing the working main code yet
- Test different directions for a project
- Several projects members would like to work on their own copy of the code
- Bug fixes that are not yet tested, but will later be merged into the main version

What is a Git branch?

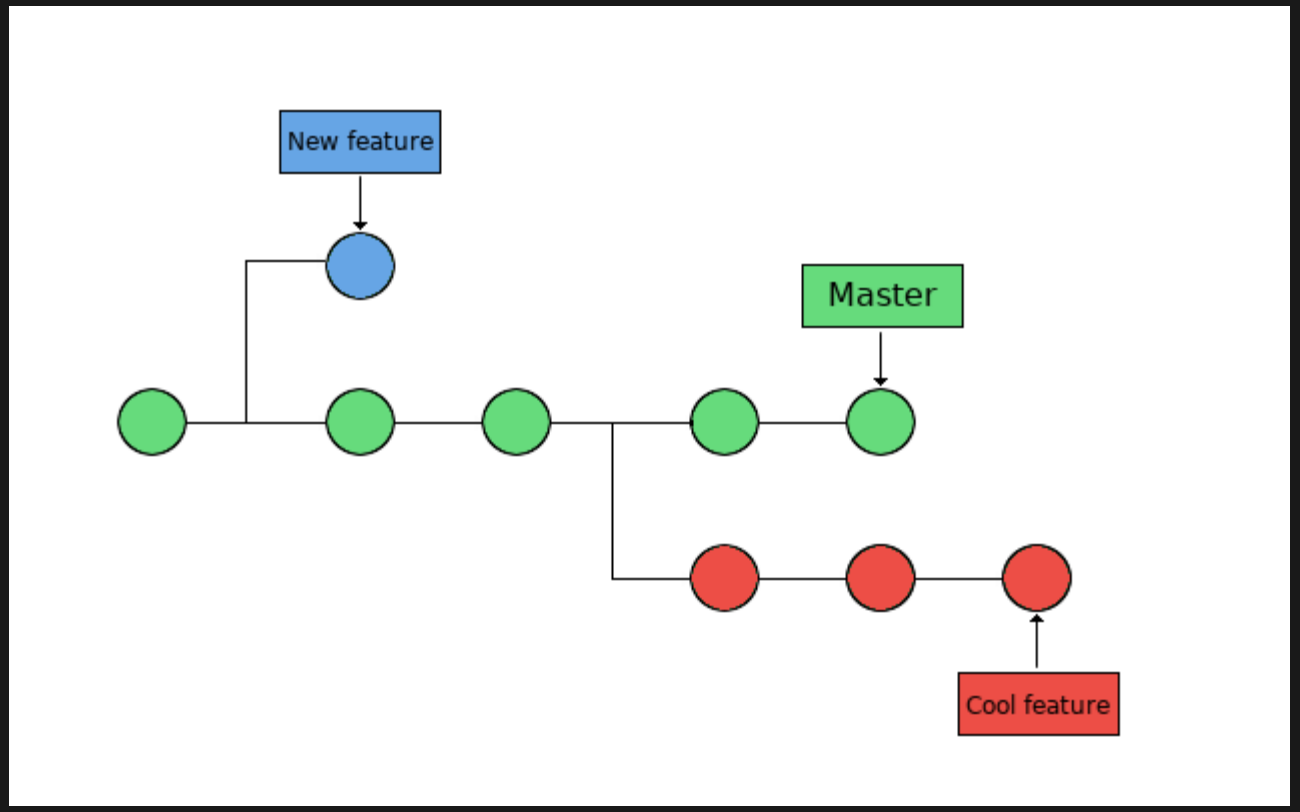
Until now, we have worked with a repository that only have one branch, with the commits done one at a time:



In the above picture, the master branch points to a commit. The current position is HEAD. (Time goes rightwards)

What is a Git branch - basic concepts

Now we want to look at repositories with several branches:



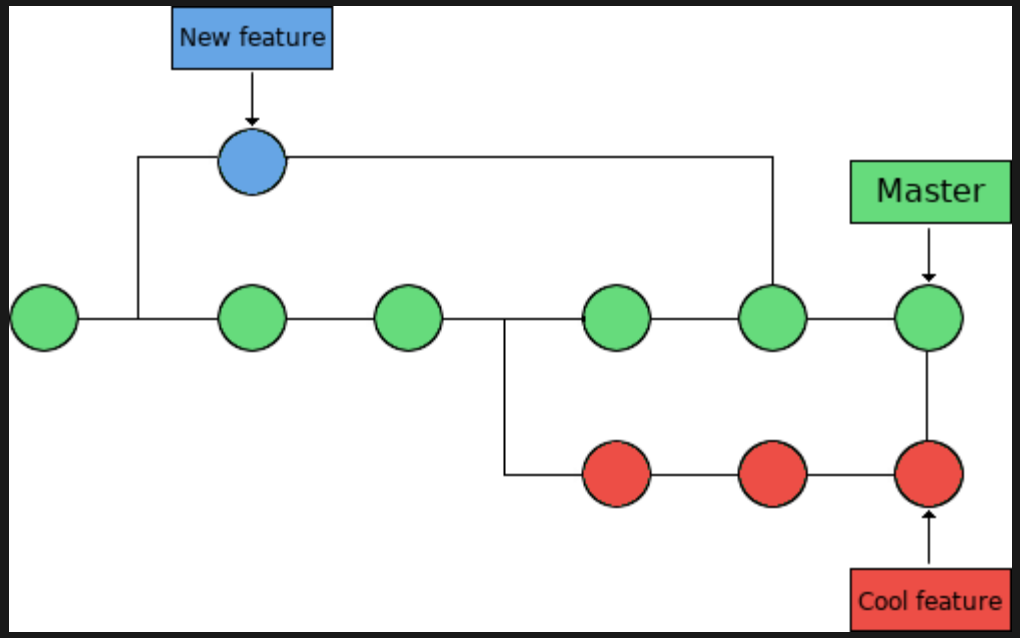
Branches are used to create another line of development. They are “individual projects” within a git repository. (Time goes rightwards)

- The branch is the commit and all its parent commits, not just the one we are currently pointing at.

- The branch is the commit and all its parent commits, not just the one we are currently pointing at.
- The main line of development is usually called the “master” branch.

- The branch is the commit and all its parent commits, not just the one we are currently pointing at.
- The main line of development is usually called the “master” branch.
- Different branches within a repository can have
    - completely different files and folders
    - almost everything the same except for a few lines of code in a file

Usually, a branch is created to work on a new feature. Once the feature is completed, it is merged back with the master branch.



(Time goes right)

Branches: Creation

Creating a new branch does not change the repository, it just points out the commit.

Branches: Creation

Creating a new branch does not change the repository, it just points out the commit.

Note that the branch is created from the current HEAD.

Branches: Creation

Creating a new branch does not change the repository, it just points out the commit.

Note that the branch is created from the current HEAD.

To create a new branch (called cool-feature in the following):

Branches: Creation

Creating a new branch does not change the repository, it just points out the commit.

Note that the branch is created from the current HEAD.

To create a new branch (called cool-feature in the following):

```
$ git branch cool-feature
```

To move to another branch (switch):

```
$ git checkout cool-feature
```

or…

```
$ git switch cool-feature
```

To move to another branch (switch):

```
$ git checkout cool-feature
```

or…

```
$ git switch cool-feature
```

If you wish to switch to a new branch that is not yet created, you can do so by adding the flag `-b` to `git checkout`.

To move to another branch (switch):

```
$ git checkout cool-feature
```

or…

```
$ git switch cool-feature
```

If you wish to switch to a new branch that is not yet created, you can do so by adding the flag `-b` to `git checkout`.

To see which branch you are on:

To move to another branch (switch):

```
$ git checkout cool-feature
```

or…

```
$ git switch cool-feature
```

If you wish to switch to a new branch that is not yet created, you can do so by adding the flag `-b` to `git checkout`.

To see which branch you are on:

```
$ git branch
```

Branches: merging, deletion

- When you have decided you are happy with the changes you made to the new branch, merge it back to the master branch (or another branch)

Branches: merging, deletion

- When you have decided you are happy with the changes you made to the new branch, merge it back to the master branch (or another branch)
- Note: The branch is always merged to the current HEAD.

Branches: merging, deletion

- When you have decided you are happy with the changes you made to the new branch, merge it back to the master branch (or another branch)
- Note: The branch is always merged to the current HEAD.
- First switch to the branch you are merging it to:

Branches: merging, deletion

- When you have decided you are happy with the changes you made to the new branch, merge it back to the master branch (or another branch)
- Note: The branch is always merged to the current HEAD.
- First switch to the branch you are merging it to:

```
$ git checkout master
```

Branches: merging, deletion

- When you have decided you are happy with the changes you made to the new branch, merge it back to the master branch (or another branch)
- Note: The branch is always merged to the current HEAD.
- First switch to the branch you are merging it to:

```
$ git checkout master
```

- Then merge them:

Branches: merging, deletion

- When you have decided you are happy with the changes you made to the new branch, merge it back to the master branch (or another branch)
- Note: The branch is always merged to the current HEAD.
- First switch to the branch you are merging it to:

```
$ git checkout master
```

- Then merge them:

```
$ git merge cool-feature
```

Branches: merging, deletion

- When you have decided you are happy with the changes you made to the new branch, merge it back to the master branch (or another branch)
- Note: The branch is always merged to the current HEAD.
- First switch to the branch you are merging it to:

```
$ git checkout master
```

- Then merge them:

```
$ git merge cool-feature
```

- You can now delete the extra branch:

Branches: merging, deletion

- When you have decided you are happy with the changes you made to the new branch, merge it back to the master branch (or another branch)
- Note: The branch is always merged to the current HEAD.
- First switch to the branch you are merging it to:

```
$ git checkout master
```

- Then merge them:

```
$ git merge cool-feature
```

- You can now delete the extra branch:

```
$ git branch -d cool-feature
```

Example - Type along if you wish

- Create a directory. Initialize a repository

Example - Type along if you wish

- Create a directory. Initialize a repository
- Create a file, stage it, and commit it

Example - Type along if you wish

- Create a directory. Initialize a repository
- Create a file, stage it, and commit it

```
$ mkdir my-project; cd my-project/
$ git init
Initialized empty Git repository in /home/bbrydsoe/my-project/.git/
$ touch file.txt
$ git add file.txt
$ git commit -m "Committing the first file"
[master (root-commit) 1006b51] Committing the first file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file.txt
```

- Create a new branch, then switch to that branch

- Create a new branch, then switch to that branch
- Make some changes - add files and text (> overwrites or are suitable for new file)

- Create a new branch, then switch to that branch
- Make some changes - add files and text (> overwrites or are suitable for new file)
- Stage the file and commit it

- Create a new branch, then switch to that branch
- Make some changes - add files and text (> overwrites or are suitable for new file)
- Stage the file and commit it

```
$ git branch cool-feature
$ git checkout cool-feature
Switched to branch 'cool-feature'
$ echo "This is a text" > file.txt
$ git add file.txt
$ git commit -m "Added text to the first file"
[cool-feature 5bad966] Added text to the first file
 1 file changed, 1 insertion(+)
```

- Switch back to the master branch, make some changes

- Switch back to the master branch, make some changes

```
$ git checkout master
Switched to branch 'master'
$ echo "Text to the second file" > second-file.txt
$ git add second-file.txt
$ git commit -m "Added a second file"
[master bdec2cf] Added a second file
 1 file changed, 1 insertion(+)
 create mode 100644 second-file.txt
```

- Switch back to the master branch, make some changes

```
$ git checkout master
Switched to branch 'master'
$ echo "Text to the second file" > second-file.txt
$ git add second-file.txt
$ git commit -m "Added a second file"
[master bdec2cf] Added a second file
 1 file changed, 1 insertion(+)
 create mode 100644 second-file.txt
```

- We should check the effect of the changes. I will use this command:

- Switch back to the master branch, make some changes

```
$ git checkout master
Switched to branch 'master'
$ echo "Text to the second file" > second-file.txt
$ git add second-file.txt
$ git commit -m "Added a second file"
[master bdec2cf] Added a second file
 1 file changed, 1 insertion(+)
 create mode 100644 second-file.txt
```

- We should check the effect of the changes. I will use this command:

```
$ git log --graph --oneline --decorate --all
```

- Switch back to the master branch, make some changes

```
$ git checkout master
Switched to branch 'master'
$ echo "Text to the second file" > second-file.txt
$ git add second-file.txt
$ git commit -m "Added a second file"
[master bdec2cf] Added a second file
 1 file changed, 1 insertion(+)
 create mode 100644 second-file.txt
```

- We should check the effect of the changes. I will use this command:

```
$ git log --graph --oneline --decorate --all
```

- or, if you made an alias before.

- Switch back to the master branch, make some changes

```
$ git checkout master
Switched to branch 'master'
$ echo "Text to the second file" > second-file.txt
$ git add second-file.txt
$ git commit -m "Added a second file"
[master bdec2cf] Added a second file
 1 file changed, 1 insertion(+)
 create mode 100644 second-file.txt
```

- We should check the effect of the changes. I will use this command:

```
$ git log --graph --oneline --decorate --all
```

- or, if you made an alias before.

```
$ git graph
```

- Switch back to the master branch, make some changes

```
$ git checkout master
Switched to branch 'master'
$ echo "Text to the second file" > second-file.txt
$ git add second-file.txt
$ git commit -m "Added a second file"
[master bdec2cf] Added a second file
 1 file changed, 1 insertion(+)
 create mode 100644 second-file.txt
```

- We should check the effect of the changes. I will use this command:

```
$ git log --graph --oneline --decorate --all
```

- or, if you made an alias before.

```
$ git graph
```

- Otherwise make the alias:

- Switch back to the master branch, make some changes

```
$ git checkout master
Switched to branch 'master'
$ echo "Text to the second file" > second-file.txt
$ git add second-file.txt
$ git commit -m "Added a second file"
[master bdec2cf] Added a second file
 1 file changed, 1 insertion(+)
 create mode 100644 second-file.txt
```

- We should check the effect of the changes. I will use this command:

```
$ git log --graph --oneline --decorate --all
```

- or, if you made an alias before.

```
$ git graph
```

- Otherwise make the alias:

```
$ git config --global alias.graph "log --all --graph --decorate --oneline"
```

This is on the master branch

This is on the master branch

```
$ git graph
* bdec2cf (HEAD -> master) Added a second file
| * 5bad966 (cool-feature) Added text to the first file
|/
* 1006b51 Committing the first file
```

This is on the master branch

```
$ git graph
* bdec2cf (HEAD -> master) Added a second file
| * 5bad966 (cool-feature) Added text to the first file
|/
* 1006b51 Committing the first file
```

We now merge the branches and check again

This is on the master branch

```
$ git graph
* bdec2cf (HEAD -> master) Added a second file
| * 5bad966 (cool-feature) Added text to the first file
|/
* 1006b51 Committing the first file
```

We now merge the branches and check again

```
$ git merge cool-feature
Merge made by the 'recursive' strategy.
 file.txt | 1 +
 1 file changed, 1 insertion(+)
```

This is on the master branch

```
$ git graph
* bdec2cf (HEAD -> master) Added a second file
| * 5bad966 (cool-feature) Added text to the first file
|/
* 1006b51 Committing the first file
```

We now merge the branches and check again

```
$ git merge cool-feature
Merge made by the 'recursive' strategy.
 file.txt | 1 +
 1 file changed, 1 insertion(+)
```

- Note that in recent git versions (>=2.33) the “recursive” strategy is replaced by the “ort” strategy.

This is on the master branch

```
$ git graph
* bdec2cf (HEAD -> master) Added a second file
| * 5bad966 (cool-feature) Added text to the first file
|/
* 1006b51 Committing the first file
```

We now merge the branches and check again

```
$ git merge cool-feature
Merge made by the 'recursive' strategy.
 file.txt | 1 +
 1 file changed, 1 insertion(+)
```

- Note that in recent git versions (>=2.33) the “recursive” strategy is replaced by the “ort” strategy.

```
$ git graph
*   cf3e6b7 (HEAD -> master) Merge branch 'cool-feature'
|\
| * 5bad966 (cool-feature) Added text to the first file
* | bdec2cf Added a second file
|/
* 1006b51 Committing the first file
```

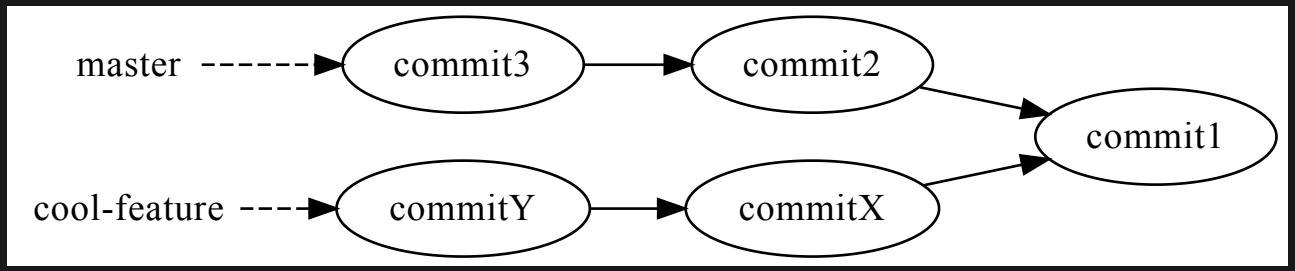
Now we can delete the new branch we had created, since all the content is now in the master branch.

```
$ git branch -d cool-feature
Deleted branch cool-feature (was 5bad966).
```

Comment: It is good practice to keep old branches for understanding of the development. Deletion could however be done for very evident mistakes or insignificant issues.

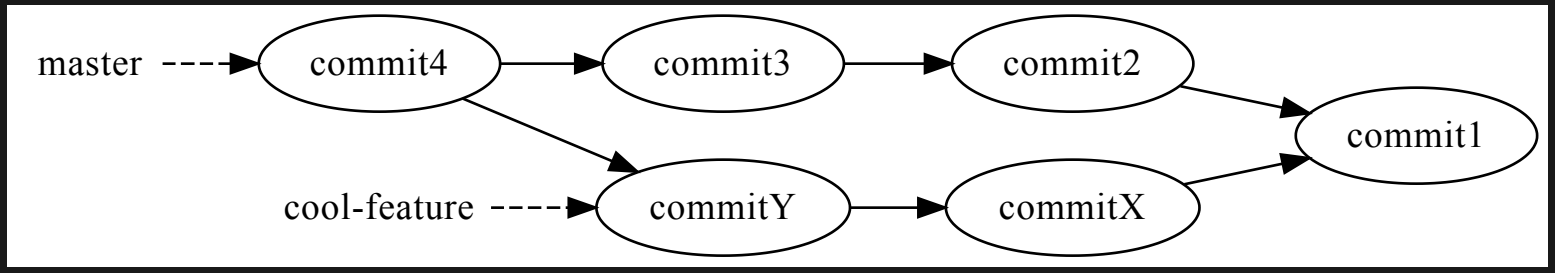
In a somewhat nicer format, it looks like this:

We commit stuff to both branches

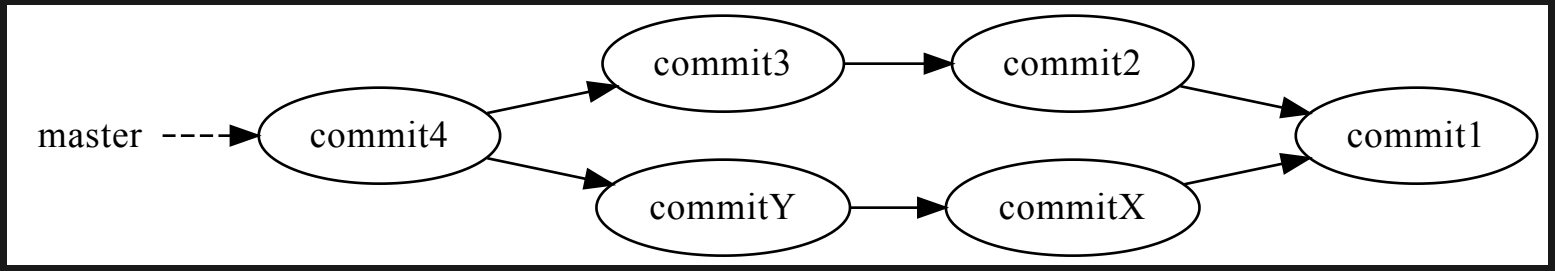


(Time goes leftwards)

Merge ‘cool-feature’ to ‘master’



Delete ‘cool-feature’



(Time goes leftwards)

Switching with uncommitted changes

As mentioned above, you switch between branches with:

Switching with uncommitted changes

As mentioned above, you switch between branches with:

```
$ git checkout <branch>
```

Switching with uncommitted changes

As mentioned above, you switch between branches with:

```
$ git checkout <branch>
```

What happens if you have uncommitted changes (and/or new files added) when you try to switch?

Switching with uncommitted changes

As mentioned above, you switch between branches with:

```
$ git checkout <branch>
```

What happens if you have uncommitted changes (and/or new files added) when you try to switch?

- The uncommitted changes will be carried to the new branch that you switch to, if possible.

Switching with uncommitted changes

As mentioned above, you switch between branches with:

```
$ git checkout <branch>
```

What happens if you have uncommitted changes (and/or new files added) when you try to switch?

- The uncommitted changes will be carried to the new branch that you switch to, if possible.
- Changes that you commit will be committed to the newly switched branch.

Switching with uncommitted changes

As mentioned above, you switch between branches with:

```
$ git checkout <branch>
```

What happens if you have uncommitted changes (and/or new files added) when you try to switch?

- The uncommitted changes will be carried to the new branch that you switch to, if possible.
- Changes that you commit will be committed to the newly switched branch.

What if there is a conflict?

Switching with uncommitted changes

As mentioned above, you switch between branches with:

```
$ git checkout <branch>
```

What happens if you have uncommitted changes (and/or new files added) when you try to switch?

- The uncommitted changes will be carried to the new branch that you switch to, if possible.
- Changes that you commit will be committed to the newly switched branch.

What if there is a conflict?

- You will **not** be allowed to switch to the other branch.

Switching with uncommitted changes

As mentioned above, you switch between branches with:

```
$ git checkout <branch>
```

What happens if you have uncommitted changes (and/or new files added) when you try to switch?

- The uncommitted changes will be carried to the new branch that you switch to, if possible.
- Changes that you commit will be committed to the newly switched branch.

What if there is a conflict?

- You will **not** be allowed to switch to the other branch.
- You must commit or stash any conflicting changes before switching branches.

Example - new file

We continue in the same repository!

Here we create a new branch, switch to it, then add a new file. Then we switch back to the master branch without committing the changes:

```
$ git checkout -b cool-feature
Switched to a new branch 'cool-feature'
$ touch newfile.txt
$ git add newfile.txt
$ git checkout master
A newfile.txt
Switched to branch 'master'
```

Git warns that there is a file added (A) in one branch but not the other, but the switch is allowed.

Example - modified file

We continue in the same repository!

First commit the `newfile.txt` in the cool-feature branch to clean the environment.

If we make changes to the file in one of the branches (go back to `cool-feature`) but not on the other and do not commit it, then git will again warn:

```
$ git switch cool-feature
$ git commit -m "newfile.txt"
$ echo "Adding some text" >> second-file.txt
$ git add second-file.txt
$ git checkout master
M second-file.txt
Switched to branch 'master'
```

Git warns that there is a file that is modified (`M`) in one branch but not the other, but the switch is allowed.

Example - uncommitted, conflicting changes

We continue in the same repository!

Assume two branches, “cool-feature” and “morefeatures”

Create the branch “morefeatures” without switching to it
Switch to branch “cool-feature”, add some text to a file, stage the file and commit it:

```
$ git branch morefeatures
$ git checkout cool-feature
Switched to branch 'cool-feature'
$ git commit -m "second-file.txt"
$ echo "add text" >> morefiles.txt
$ git add morefiles.txt
$ git commit -m "Some text"
[cool-feature 469542b] Some text
 1 file changed, 1 insertion(+)
 create mode 100644 morefiles.txt
```

Switch to branch “morefeatures”. Modify the same file, stage the file and commit it. Then try and switch back to the “cool-features” branch:

```
$ git checkout morefeatures
Switched to branch 'morefeatures'
$ echo "Adding yet some more text" >> morefiles.txt
$ git add morefiles.txt
$ git checkout cool-feature
error: Your local changes to the following files would be overwritten by checkout:
  morefiles.txt
Please commit your changes or stash them before you switch branches.
Aborting
```

Now Git complains and do not allow the switch.

Handling uncommitted changes

So, what can we do if there is a conflict?

Handling uncommitted changes

So, what can we do if there is a conflict?

- Commit the changes before switching branch

Handling uncommitted changes

So, what can we do if there is a conflict?

- Commit the changes before switching branch
- Stash the uncommitted changes

Handling uncommitted changes

So, what can we do if there is a conflict?

- Commit the changes before switching branch
- Stash the uncommitted changes
- Discard the uncommitted changes

Handling uncommitted changes

So, what can we do if there is a conflict?

- Commit the changes before switching branch
- Stash the uncommitted changes
- Discard the uncommitted changes
- Checkout with Merge

Stashing

The command “stash” can be described as a **drawer** where you store uncommitted changes temporarily.

After stashing your uncommitted changes you can continue **working on other things** in a different branch.

The uncommitted changes that are stored in the stash **can be taken out and applied to any branch, including the original branch.**

Stashing, example (no type-along this time)

First do a `git status` in the branch where you may have uncommitted changes:

```
$ git status
On branch morefeatures
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  modified:   file.txt
  new file:   morefiles.txt
```

You can see the dirty status.

To fix it, let us use `git stash`:

```
$ git stash
Saved working directory and index state WIP on morefeatures: 4922606 Some tex
```

To fix it, let us use `git stash`:

```
$ git stash
Saved working directory and index state WIP on morefeatures: 4922606 Some tex
```

Checking again with `git status`:

To fix it, let us use `git stash`:

```
$ git stash
Saved working directory and index state WIP on morefeatures: 4922606 Some tex
```

Checking again with `git status`:

```
$ git status
On branch morefeatures
nothing to commit, working tree clean
```

To fix it, let us use `git stash`:

```
$ git stash
Saved working directory and index state WIP on morefeatures: 4922606 Some tex
```

Checking again with `git status`:

```
$ git status
On branch morefeatures
nothing to commit, working tree clean
```

You can now switch branches and work on something else.

Working with stashes (repetition)

You can have several stashes stored. To see them, use

```
$ git stash list
```

Example:

```
$ git stash list
stash@{0}: WIP on morefeatures: 4922606 Some text
stash@{1}: WIP on morefeatures: 4922606 Some text
stash@{2}: WIP on morefeatures: 4922606 Some text
```

Working with stashes - continued (repetition)

When you have done what you needed before committing the stashed changes you can reapply a stash (select branch first), using

```
$ git stash apply
```

which will apply the most recent stash. If you want to apply a different stash, you can name it.

Example:

```
$ git stash apply stash@{0}
```

Discarding uncommitted changes

If you do not want to stash your changes, but just **get rid** of them, you can use `git clean`.

Discarding uncommitted changes

If you do not want to stash your changes, but just **get rid** of them, you can use `git clean`.

WARNING: This command will remove all non-tracked files in your current directory!

Discarding uncommitted changes

If you do not want to stash your changes, but just **get rid** of them, you can use `git clean`.

WARNING: This command will remove all non-tracked files in your current directory!

You can safely test which files will be removed by running:

Discarding uncommitted changes

If you do not want to stash your changes, but just **get rid** of them, you can use `git clean`.

WARNING: This command will remove all non-tracked files in your current directory!

You can safely test which files will be removed by running:

```
$ git clean --dry-run
```

Handling uncommitted changes - merging

- There is a checkout with merge option. Add the flag `--merge` (or `-m`):

```
$ git checkout --merge <branch>
```

Handling uncommitted changes - merging

- There is a checkout with merge option. Add the flag `--merge` (or `-m`):

```
$ git checkout --merge <branch>
```

- This will perform a **three-way merge between your working tree and the new branch, with the current branch as the base.**

Handling uncommitted changes - merging

- There is a checkout with merge option. Add the flag `--merge` (or `-m`):

```
$ git checkout --merge <branch>
```

- This will perform a **three-way merge between your working tree and the new branch, with the current branch as the base.**
- After the merge, you will be on the new branch and the merged result will be in your working tree.

Handling uncommitted changes - merging

- There is a checkout with merge option. Add the flag `--merge` (or `-m`):

```
$ git checkout --merge <branch>
```

- This will perform a **three-way merge between your working tree and the new branch, with the current branch as the base.**
- After the merge, you will be on the new branch and the merged result will be in your working tree.
- NOTE: As with any merge, **conflicts may result** and you will then have to resolve those.

Merging and merge conflicts

- Merge conflicts will happen now and then when you are working with more than one branch and try to merge them.

Merging and merge conflicts

- Merge conflicts will happen now and then when you are working with more than one branch and try to merge them.
- In many cases, Git is actually able to do a merge without problems. However, merge conflicts can happen.

Merging and merge conflicts

- Merge conflicts will happen now and then when you are working with more than one branch and try to merge them.
- In many cases, Git is actually able to do a merge without problems. However, merge conflicts can happen.
- If Git cannot safely merge something automatically, you will get a message like this:

Merging and merge conflicts

- Merge conflicts will happen now and then when you are working with more than one branch and try to merge them.
- In many cases, Git is actually able to do a merge without problems. However, merge conflicts can happen.
- If Git cannot safely merge something automatically, you will get a message like this:

```
error: Entry '<fileName>' would be overwritten by merge.
Cannot merge. (Changes in staging area)
```

Merging and merge conflicts

- Merge conflicts will happen now and then when you are working with more than one branch and try to merge them.
- In many cases, Git is actually able to do a merge without problems. However, merge conflicts can happen.
- If Git cannot safely merge something automatically, you will get a message like this:

```
error: Entry '<fileName>' would be overwritten by merge.
Cannot merge. (Changes in staging area)
```

- NOTE: Always check that you are on the right branch before merging! You check the branch with `git branch`.

Git can automatically try to merge when you give the command:

```
$ git merge <branch-to-merge-into-present-branch>
```

while standing on the branch you want to merge to.

Merge strategies

The most commonly used

- Fast Forward Merge
  - the commit history is one straight line.
  - You create a branch, you make some commits there, but no changes to the ‘master’. You then just merge onto the ‘master’. This just moves the pointer for the ‘master’ branch forward in a straight line.
- Recursive Merge (until 2.32)
  - make a branch and make some commits there, but also make new commits that are made on another branch, like the ‘master‘.
  - Then, when you want to merge, git will recurse over the branch and create a new merge commit. The merge commit will continue to have two parents.
- ORT (from git-2.33)
  - acronym for “Ostensibly Recursive’s Twin”
  - replacement for the previous default algorithm, recursive.
  - This is the default merge strategy when pulling or merging one branch.

Merge conflicts, example

Here we create a merge conflict:

```
$ mkdir merge-test
$ cd merge-test/
~/merge-test$ git init
Initialized empty Git repository in /home/bbrydsoe/merge-test/.git/
~/merge-test$ echo "Creating a file with some text to play with." >> myfile.txt
~/merge-test$ git add myfile.txt
~/merge-test$ git commit -m "First commit"
[master (root-commit) 9badcc6] First commit
 1 file changed, 1 insertion(+)
 create mode 100644 myfile.txt
~/merge-test$ git checkout -b mergebranch
Switched to a new branch 'mergebranch'
~/merge-test$ echo "Adding text to the file in order to merge." > myfile.txt
```

```
~/merge-test$ git add myfile.txt
~/merge-test$ git commit -m "Changed the content of myfile.txt"
[mergebranch 41b0e36] Changed the content of myfile.txt
 1 file changed, 1 insertion(+), 1 deletion(-)
~/merge-test$ git checkout master
Switched to branch 'master'
~/merge-test$ echo "Put more text to the file" >> myfile.txt
~/merge-test$ git add myfile.txt
bbrydsoe@enterprise-a:~/merge-test$ git commit -m "Added more text"
[master c17e479] Added more text
 1 file changed, 1 insertion(+)
~/merge-test$ git merge mergebranch
Auto-merging myfile.txt
CONFLICT (content): Merge conflict in myfile.txt
Automatic merge failed; fix conflicts and then commit the result.
```

So Git complains

We can get some more information with the `git status` command:

```
~/merge-test$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

  both modified:   myfile.txt

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

Looking inside the file myfile.txt:

```
~/merge-test$ cat myfile.txt
<<<<<<< HEAD
Creating a file with some text to play with.
Put more text to the file
=======
Adding text to the file in order to merge.
>>>>>>> mergebranch
```

Some “conflict dividers” have been added.

Resolving merge conflicts

- The most direct way to resolve the conflict is editing the file yourself.

Resolving merge conflicts

- The most direct way to resolve the conflict is editing the file yourself.
- When this has been done, you can repeat the merge with:

Resolving merge conflicts

- The most direct way to resolve the conflict is editing the file yourself.
- When this has been done, you can repeat the merge with:

```
$ git merge --continue <branch-to-merge>
```

Resolving merge conflicts

- The most direct way to resolve the conflict is editing the file yourself.
- When this has been done, you can repeat the merge with:

```
$ git merge --continue <branch-to-merge>
```

Commands to help:

Resolving merge conflicts

- The most direct way to resolve the conflict is editing the file yourself.
- When this has been done, you can repeat the merge with:

```
$ git merge --continue <branch-to-merge>
```

Commands to help:

- To identify conflicting files: `git status`

Resolving merge conflicts

- The most direct way to resolve the conflict is editing the file yourself.
- When this has been done, you can repeat the merge with:

```
$ git merge --continue <branch-to-merge>
```

Commands to help:

- To identify conflicting files: `git status`
- To get a list of commits that conflict between the branches: `git log --merge`

Resolving merge conflicts

- The most direct way to resolve the conflict is editing the file yourself.
- When this has been done, you can repeat the merge with:

```
$ git merge --continue <branch-to-merge>
```

Commands to help:

- To identify conflicting files: `git status`
- To get a list of commits that conflict between the branches: `git log --merge`
- Find differences between states of a repository/files: `git diff`

Resolving merge conflicts

- The most direct way to resolve the conflict is editing the file yourself.
- When this has been done, you can repeat the merge with:

```
$ git merge --continue <branch-to-merge>
```

Commands to help:

- To identify conflicting files: `git status`
- To get a list of commits that conflict between the branches: `git log --merge`
- Find differences between states of a repository/files: `git diff`
- Reset conflicted files to a known good state: `git reset`

Resolving merge conflicts

- The most direct way to resolve the conflict is editing the file yourself.
- When this has been done, you can repeat the merge with:

```
$ git merge --continue <branch-to-merge>
```

Commands to help:

- To identify conflicting files: `git status`
- To get a list of commits that conflict between the branches: `git log --merge`
- Find differences between states of a repository/files: `git diff`
- Reset conflicted files to a known good state: `git reset`

If you made a mistake when you resolved a conflict and have completed the merge before realizing, you can roll back to the commit before the merge was done with the command `git reset --hard`.

Workflow - merge goes well

Workflow - merge goes well

- Work on files

Workflow - merge goes well

- Work on files
- Stage files

Workflow - merge goes well

- Work on files
- Stage files
- Commit files

Workflow - merge goes well

- Work on files
- Stage files
- Commit files
- Then do

Workflow - merge goes well

- Work on files
- Stage files
- Commit files
- Then do

```
$ git merge <other-branch>
```

Workflow - merge goes well

- Work on files
- Stage files
- Commit files
- Then do

```
$ git merge <other-branch>
```

Success!

Workflow - merge goes badly

- (Commit files)

Workflow - merge goes badly

- Work on files
- (Commit files)

Workflow - merge goes badly

- Work on files
- Stage files

- (Commit files)

Workflow - merge goes badly

- Work on files
- Stage files
- Commit files

- (Commit files)

Workflow - merge goes badly

- Work on files
- Stage files
- Commit files
- Then do

- (Commit files)

Workflow - merge goes badly

- Work on files
- Stage files
- Commit files
- Then do

```
$ git merge <other-branch>`
```

- (Commit files)

Workflow - merge goes badly

- Work on files
- Stage files
- Commit files
- Then do

```
$ git merge <other-branch>`
```

- CONFLICT!

- (Commit files)

Workflow - merge goes badly

- Work on files
- Stage files
- Commit files
- Then do

```
$ git merge <other-branch>`
```

- CONFLICT!
- Fix problems

- (Commit files)

Workflow - merge goes badly

- Work on files
- Stage files
- Commit files
- Then do

```
$ git merge <other-branch>`
```

- CONFLICT!
- Fix problems
- Stage files
- (Commit files)

Workflow - merge goes badly

- Work on files
- Stage files
- Commit files
- Then do

```
$ git merge <other-branch>`
```

- CONFLICT!
- Fix problems
- Stage files
- (Commit files)
    - not necessary because git merge --continue takes care of that

Workflow - merge goes badly

- Work on files
- Stage files
- Commit files
- Then do

```
$ git merge <other-branch>`
```

- CONFLICT!
- Fix problems
- Stage files
- (Commit files)
    - not necessary because git merge --continue takes care of that
- Then do

Workflow - merge goes badly

- Work on files
- Stage files
- Commit files
- Then do

```
$ git merge <other-branch>`
```

- CONFLICT!
- Fix problems
- Stage files
- (Commit files)
    - not necessary because git merge --continue takes care of that
- Then do

```
$ git merge --continue <other-branch>`
```

Workflow - merge goes badly

- Work on files
- Stage files
- Commit files
- Then do

```
$ git merge <other-branch>`
```

- CONFLICT!
- Fix problems
- Stage files
- (Commit files)
    - not necessary because git merge --continue takes care of that
- Then do

```
$ git merge --continue <other-branch>`
```

Success!

Rebasing

- Rebasing is the process of moving or combining a sequence of commits to a new base commit.

Rebasing

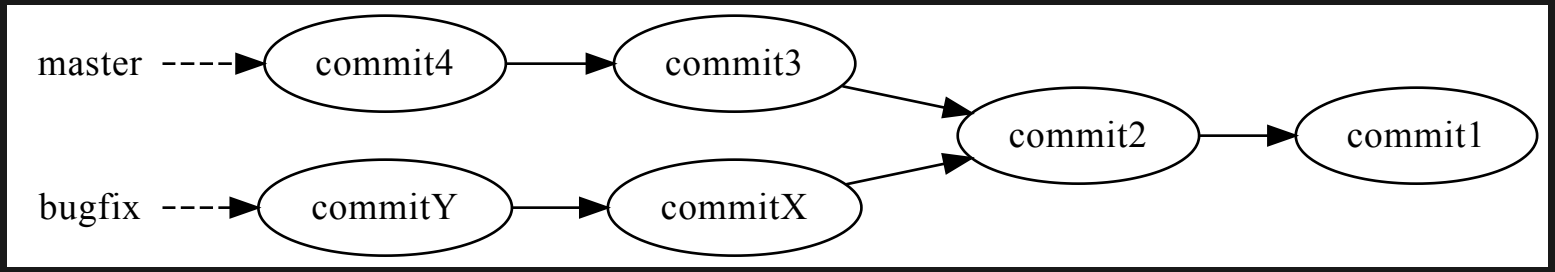
- Rebasing is the process of moving or combining a sequence of commits to a new base commit.
- It solves the same problem as git merge. The commands are both used to integrate changes from one branch into another branch, however the way they do it is very different.

Rebasing

- Rebasing is the process of moving or combining a sequence of commits to a new base commit.
- It solves the same problem as git merge. The commands are both used to integrate changes from one branch into another branch, however the way they do it is very different.
- When you do a rebase, all the changes will be compressed together in a single “patch” which is then “applied” - rebasing creates new commits on the other branch for each commit in the original branch.

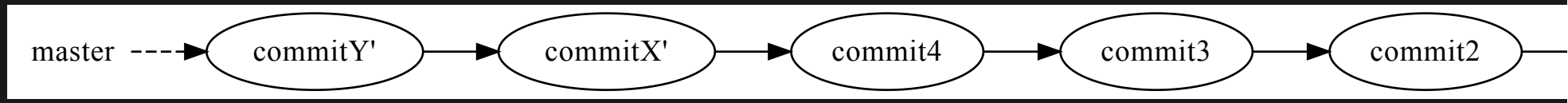
Rebasing - illustration

Branch ‘bugfix’ was branched from ‘master’



(Time goes leftwards)

Rebasing ‘bugfix’ to the ‘master’ branch



(Time goes leftwards)

Rebasing - continued

Assume a master branch and the branch “cool-features” and that you want to rebase the branch “cool-features” onto the master branch:

```
$ git checkout cool-features
$ git rebase master
```

Rebasing - continued

Assume a master branch and the branch “cool-features” and that you want to rebase the branch “cool-features” onto the master branch:

```
$ git checkout cool-features
$ git rebase master
```

This works by

Rebasing - continued

Assume a master branch and the branch “cool-features” and that you want to rebase the branch “cool-features” onto the master branch:

```
$ git checkout cool-features
$ git rebase master
```

This works by

- going to the common ancestor of the two branches

Rebasing - continued

Assume a master branch and the branch “cool-features” and that you want to rebase the branch “cool-features” onto the master branch:

```
$ git checkout cool-features
$ git rebase master
```

This works by

- going to the common ancestor of the two branches
- getting the diff introduced by each commit of the branch you are on

Rebasing - continued

Assume a master branch and the branch “cool-features” and that you want to rebase the branch “cool-features” onto the master branch:

```
$ git checkout cool-features
$ git rebase master
```

This works by

- going to the common ancestor of the two branches
- getting the diff introduced by each commit of the branch you are on
- saving those diffs to temporary files

Rebasing - continued

Assume a master branch and the branch “cool-features” and that you want to rebase the branch “cool-features” onto the master branch:

```
$ git checkout cool-features
$ git rebase master
```

This works by

- going to the common ancestor of the two branches
- getting the diff introduced by each commit of the branch you are on
- saving those diffs to temporary files
- resetting the current branch to the same commit as the branch you are rebasing onto

Rebasing - continued

Assume a master branch and the branch “cool-features” and that you want to rebase the branch “cool-features” onto the master branch:

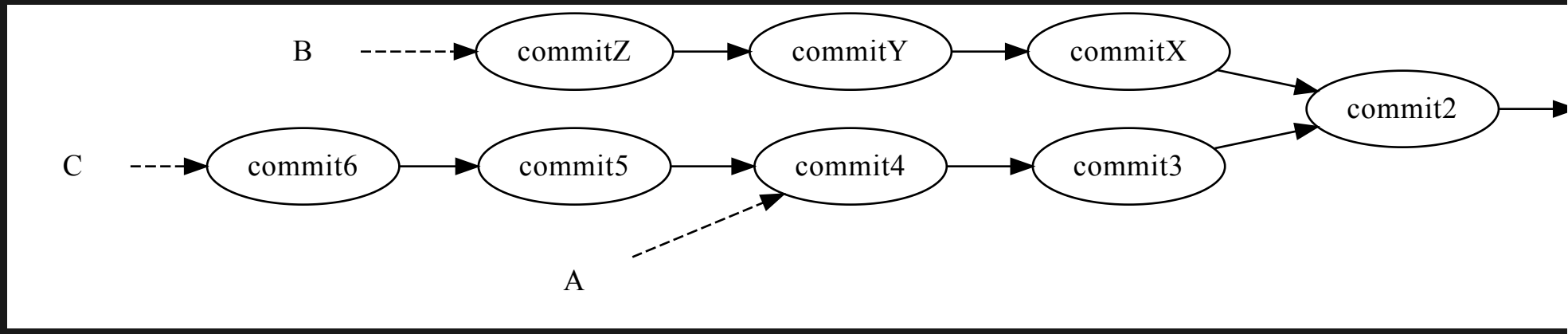
```
$ git checkout cool-features
$ git rebase master
```

This works by

- going to the common ancestor of the two branches
- getting the diff introduced by each commit of the branch you are on
- saving those diffs to temporary files
- resetting the current branch to the same commit as the branch you are rebasing onto
- apply each change in turn

Rebasing vs. Fast-forward merge

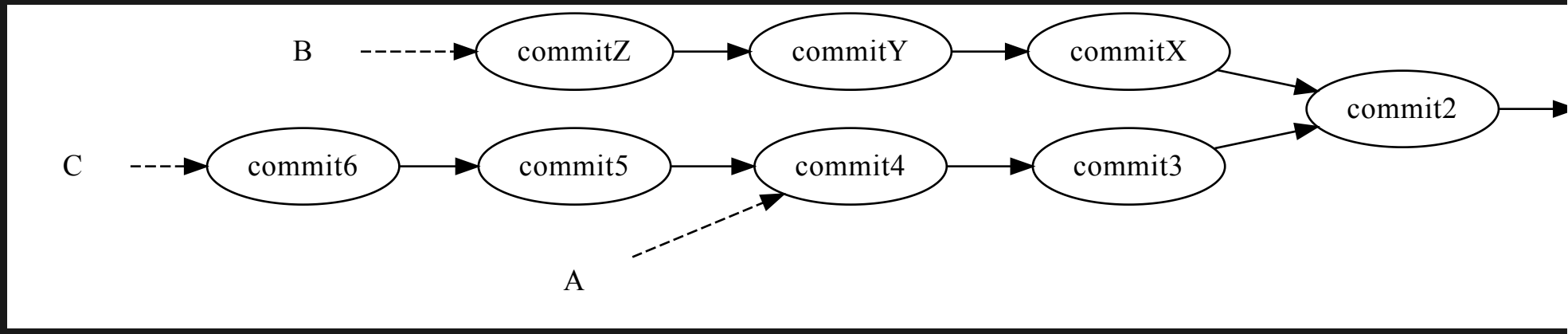
Not the same! A rebase moves a branch from one base to another. A fast-forward merge moves a branch head from the current commit to a commit for a descendant.



Rebasing vs. Fast-forward merge

Not the same! A rebase moves a branch from one base to another. A fast-forward merge moves a branch head from the current commit to a commit for a descendant.

Example:

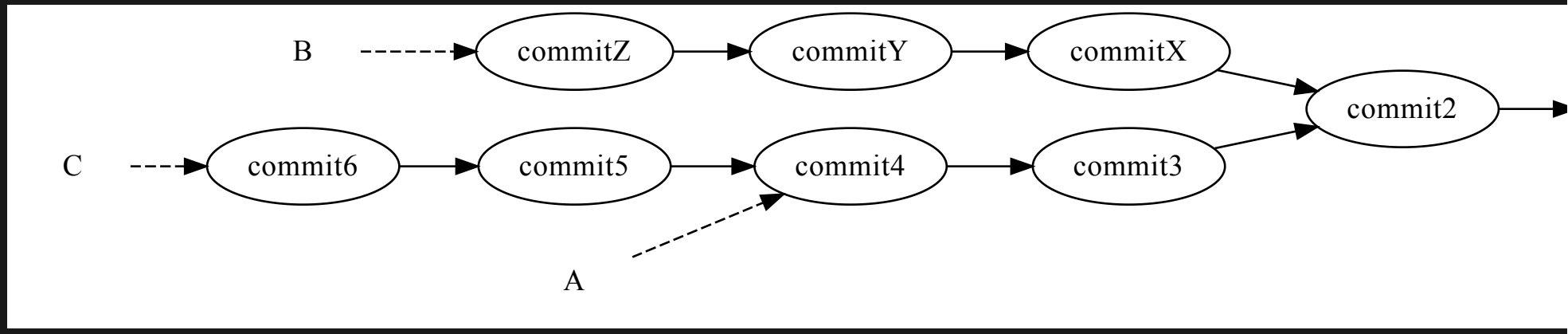


Rebasing vs. Fast-forward merge

Not the same! A rebase moves a branch from one base to another. A fast-forward merge moves a branch head from the current commit to a commit for a descendant.

Example:

Start

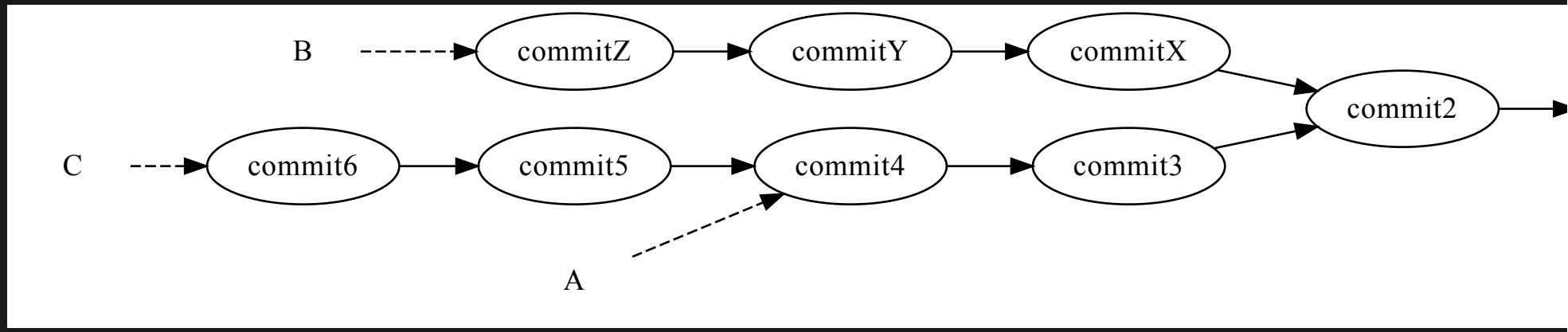


Rebasing vs. Fast-forward merge

Not the same! A rebase moves a branch from one base to another. A fast-forward merge moves a branch head from the current commit to a commit for a descendant.

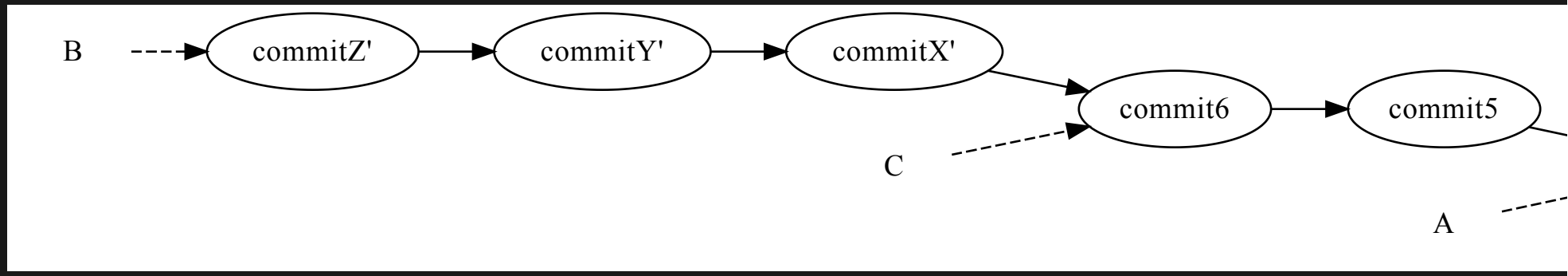
Example:

Start

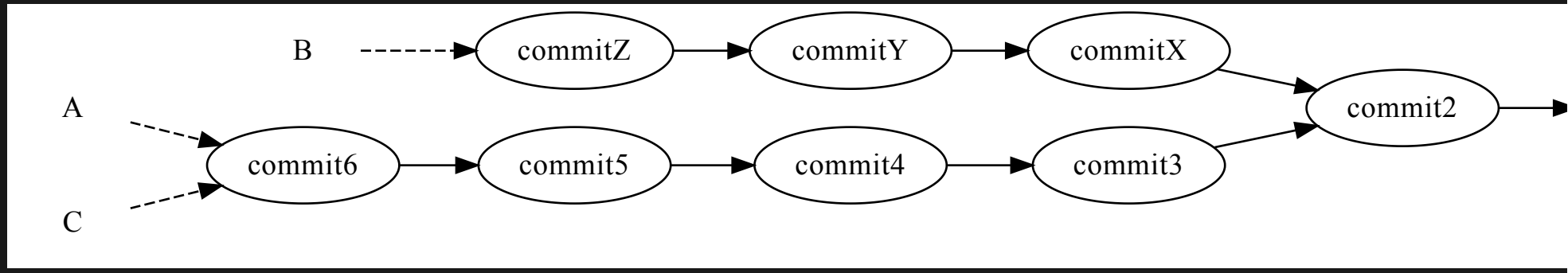


(Time goes leftwards)

Rebase B onto C



FF merge C into A:



(Time goes leftwards)

Cherry-picking (advanced)

Basically, cherry-picking in Git means that you choose a commit from one branch that you apply to another.

Cherry-picking (advanced)

Basically, cherry-picking in Git means that you choose a commit from one branch that you apply to another.

Find the hash for the commit you want to apply, using `git log`.

Cherry-picking (advanced)

Basically, cherry-picking in Git means that you choose a commit from one branch that you apply to another.

Find the hash for the commit you want to apply, using `git log`.

Then make sure you are on the right branch that you want to apply the commit to:

Cherry-picking (advanced)

Basically, cherry-picking in Git means that you choose a commit from one branch that you apply to another.

Find the hash for the commit you want to apply, using `git log`.

Then make sure you are on the right branch that you want to apply the commit to:

```
$ git checkout <branch>
```

Cherry-picking (advanced)

Basically, cherry-picking in Git means that you choose a commit from one branch that you apply to another.

Find the hash for the commit you want to apply, using `git log`.

Then make sure you are on the right branch that you want to apply the commit to:

```
$ git checkout <branch>
```

Now you execute the cherry-picking:

Cherry-picking (advanced)

Basically, cherry-picking in Git means that you choose a commit from one branch that you apply to another.

Find the hash for the commit you want to apply, using `git log`.

Then make sure you are on the right branch that you want to apply the commit to:

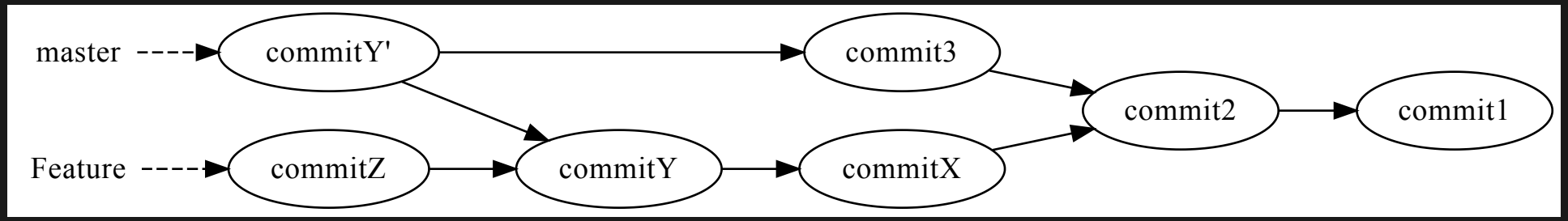
```
$ git checkout <branch>
```

Now you execute the cherry-picking:

```
$ git cherry-pick <hash>
```

Cherry-picking — illustration (advanced)

Apply the commit Y to the master branch (called Y´)



(Time goes leftwards)

Take aways

- create or parse branch - `git branch`
- switch branch - `git checkout` or `git switch`
- merge branches - `git merge`
- rebase branch - `rebase` - like merge but end result is just one branch

Conflict?

- Commit or `stash` or discard (`clean`) the changes before switching branch or do a `checkout --merge` .

Workflow merge

- Work on files
- Stage and commit files
- Then do: `$ git merge <other-branch>`

Conflict?

- Fix problems
- Stage and commit files
- Then do: `$ git merge --continue <other-branch>`

Exercises

Each of the exercises has a README.md file with explanations and descriptions of what to do. You can find all of them in the subdirectory 5.branches. You should do them in the below order:

1. Fast-forward Merge (OK): This exercise will show an example where git can do a fast-forward merge. The exercise is in the subdirectory “1.merge-ok”
2. Recursive/ORT Merge (OK): In this exercise you will see an example where git can automatically merge two branches. This time git will use the recursive merge. The exercise can be found in the subdirectory “2.merge-ok-recursive”

Exercises

3. Merge (BAD): This exercise gives an example of a merge that cannot be done automatically with the merge command. The exercise can be found in the subdirectory “3.merge-bad”
4. Rebasing (OK): In this exercise you will try the command rebase and see that it succeeds. The exercise can be found in the subdirectory “4.rebase-ok”
5. Rebasing (BAD): This exercise again gives an example of rebasing two branches, but in this case the rebase fails. The exercise can be found in the subdirectory “5.rebase-bad”
