Recap

Last lecture: Version Control Systems ⇒ git

⇒ 3 areas

⇒ creating commits, checking out old commits

⇒ working with remotes

⇒ pull/push

⇒ working with branches

⇒ merging branches via git merge
Recap quiz

Fill out the following graphic:
Recap quiz

Fill out the following graphic:

Working directory -> Staging Area

unmodified/modified files | stage files | committed files

Staging Area -> .git directory (Repository)

commit files
More on Git
Master and feature branches

⇒ Typically, there's a master branch in the repo
  → don't use it for development, rather store "releasable" version of your code/assets on it
  → content on the master branch should work, i.e. no errors.
  ⇒ DON'T BREAK THE MASTER!

⇒ In a couple slides: Typical git workflows.
Rebasing
15.01 git merge vs. git rebase

⇒ To join branches, **git rebase** is an alternative to **git merge**

⇒ If you don't know how to rebase properly, things can go wrong badly

Recap:
⇒ to rebase a branch on another, run `git rebase`. Assuming you're on branch feature, then you can rebase onto the master via:

```
.git rebase master
```

branch on which to base on

rebasing feature on master general syntax:

```
.git rebase master feature
```

(will checkout feature)
15.03 Golden rule of rebasing

⇒ NEVER, NEVER, NEVER rebase the master onto a feature branch.

→ Only rebase feature on master OR

→ featureA on featureB

I.e., don't run the following commands:

```
  git checkout master && git rebase feature

  git rebase feature master
```
15.03 Golden rule of rebasing

⇒ If you rebased the master on your feature, you would create a confusing history.
15.04 Rebase example

**setup**

```bash
git init &&
echo -e "# README
" > README.md &&
git add . &&
git commit -m "initial commit" &&
echo "This is a readme file." >> README.md &&
git commit -a -m "updated readme" &&
git checkout HEAD~1 &&
git checkout -b feature &&
echo "feature branch. " >> README.md &&
git commit -a -m "feature update."
```

First, rewinding head to replay your work on top of it...
Applying: feature update.
Using index info to reconstruct a base tree...
M README.md
.git/rebase-apply/patch:8: trailing whitespace.
feature branch.
warning: 1 line adds whitespace errors.
Falling back to patching base and 3-way merge...
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
error: Failed to merge in the changes.
The copy of the patch that failed is found in:
.git/rebase-apply/patch

Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --abort".
15.05 Resolving conflicts in rebase

⇒ resolve conflicts on individual files using the 3 options:
   1.) `git checkout --ours`
   2.) `git checkout --theirs`
   3.) manual merge

⇒ add resolved files via `git add`. (you can also edit the commit message via `git commit --amend`)

⇒ after conflict resolution run `git rebase --continue`

⇒ `git rebase --abort` stops the rebase!
15.06 Completing the rebase of feature on master

⇒ After rebasing on the master, feature branch can be "cleanly" merged to master (i.e. a fast-forward merge)

⇒ `git checkout master && git merge feature`

git merge places the commits of feature on top of the commits of the master
More on rebasing
15.07 interactive rebase & squashing commits

⇒ often you work on a separate branch but don't want to put all commits on the master or only a subset

⇒ `git rebase -i` starts rebase in interactive mode, which allows for more efficient history manipulation.

⇒ interactive mode allows to squash multiple commits into one (first commit must be pick / p though)

Tip:
Use `git config rebase.abbreviateCommands true` to force git to use shortcuts only instead of pick, squash, ...
More on conflict resolution
15.08 Visual merge tools

⇒ manual conflict resolution can be done in the console or via an IDE (most IDEs provide built-in support for merging)

⇒ there are many visual merge tools available, e.g.
  - vimdiff
  - meld
  - GitKraken

⇒ to start merging via a tool, run `git mergetool`
  → configure it via `git config merge.tool meld`
  → per default, git creates .orig backup files. Disable via `git config mergetool.keepBackup false`
Stashing
15.09 Stashing

⇒ sometimes you work on a branch and have to switch to another one, but you don't feel like committing yet.

→ *git stash* saves changes away onto a temporary stack and reverts your local working copy

⇒ use *git stash* to save local changes

⇒ *git stash pop* to apply previously stashed changes
15.09 Stashing

⇒ to keep changes in stash and to apply them, use `git stash apply`
  → Can be used to apply changes to multiple branches

⇒ `git stash list` displays overview of stashed changes
  → pop n-th stash via `git stash pop stash@{n}`
  → remove n-th stash via `git stash drop stash@{n}`
  → clear stash via `git stash clear`

⇒ you can add a note to a stash, by using `git stash save "note"`
15.10 Cleaning the repo

⇒ sometimes you accumulate a lot of temporary / ignored files in your repository.

→ `git clean -n` to list what files would be removed

→ `git clean -f` to remove untracked files

→ `git clean -xf` to remove untracked and ignored files
15.11 Discarding local changes

⇒ to discard ALL local changes (no undo for this), you can use
  `git reset --hard`

⇒ You can also use `git reset` to reset the HEAD to a specific commit, DO THIS ONLY if you haven't pushed to a remote yet.

  → don't screw up the remote
  → if you use `git reset` frequently, there's something wrong.
Tags
Last lecture: checkout specific commits via their SHA-1 hash
- creates detached head

Often you want to release your software to the public at specific commits
- tags provide an option to "tag" or mark a commit

List available tags via `git tag`
- you can search for tags using a regex via `git tag -l "<regex>"`
There are two types of tags:

1) lightweight
2) annotated

lightweight tags are just a reference to a commit (i.e., the checksum)

→ use `git tag <tagname>` to create a lightweight tag
→ you need to explicitly push a tag to a remote via `git push origin <tagname>`
→ checkout a tag via `git checkout <tagname>`
15.12 Creating tags

⇒ to create an annotated tag (with a message) use

    git tag -a <tagname> -m "tag message"

⇒ to retrieve tag info, use git show <tagname>

⇒ push tag via git push origin <tagname>

Commit messages
15.13 How to write good commit messages

⇒ writing good commit messages is an art for itself

⇒ Try to make them informative and to keep track of changes

⇒ If you make a pull request or push onto a public branch, have clean & clear commit messages
## 15.13 How to write good commit messages

<table>
<thead>
<tr>
<th>bad examples</th>
<th>good examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>add cli new</td>
<td>Fix error when the URL is not reachable</td>
</tr>
<tr>
<td>fixes</td>
<td>Add error message for file not found</td>
</tr>
<tr>
<td>fix code review comments</td>
<td>Add server fingerprint check</td>
</tr>
<tr>
<td>no message</td>
<td>Fix shadow box closing problem</td>
</tr>
<tr>
<td>description</td>
<td></td>
</tr>
<tr>
<td>wip</td>
<td></td>
</tr>
<tr>
<td>hackz</td>
<td></td>
</tr>
<tr>
<td>little edit</td>
<td></td>
</tr>
</tbody>
</table>
15.13 How to write good commit messages

⇒ write in imperative mode: If commit is applied, <your message>

⇒ write a short subject line of a maximum of 50-72 chars and capitalize first word e.g.
   Fix float casting bug in compiler

⇒ add whitespace line, then details of your commit.

⇒ Don't explain how it was done, but instead what and why

15.13 How to write good commit messages

# 50-character subject line
#
# 72-character wrapped longer description. This should answer:
#
# * Why was this change necessary?
# * How does it address the problem?
# * Are there any side effects?
#
# Include a link to the ticket, if any.
#
# Add co-authors if you worked on this code with others:
#
# Co-authored-by: Full Name <email@example.com>
# Co-authored-by: Full Name <email@example.com>

template from https://thoughtbot.com/blog/write-good-commit-messages-by-blaming-others
Git workflows
There are several collaboration models or workflows used in (software) engineering teams:

- central part of them is a pull request
- most repository management systems like github/bitbucket/gitlab/… provide support for pull requests/reviews/…

Following slides are based on: https://www.slideshare.net/psguy/git-collaboration
15.14 Pull request

(1)  Create feature on dedicated branch in local repo
(2)  Push branch to public repository/remote
(3)  File pull request to official repository
(4)  Other developers review code, discuss it, update it
(5)  Project maintainer merges feature into official repository and closes the pull request
15.15 Four standard git workflows

Feature Branch Workflow

Centralized Workflow

Gitflow Workflow

Forking Workflow
15.16 Centralized workflow

⇒ one master branch on which everybody works

<table>
<thead>
<tr>
<th>Pro</th>
<th>Con</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ simple flow</td>
<td>- more conflicts when many developers work together</td>
</tr>
<tr>
<td>+ good for not-frequently updated/changed projects</td>
<td>- no review or feature pull requests allowed</td>
</tr>
<tr>
<td></td>
<td>- no branching</td>
</tr>
<tr>
<td></td>
<td>- everybody works on the same branch</td>
</tr>
<tr>
<td></td>
<td>- high chance for dirty master/problems</td>
</tr>
</tbody>
</table>

Diagram: Centralized Workflow
15.17 Forking workflow

⇒ everybody forks the official repo, changes are added using pull request to the official repo

<table>
<thead>
<tr>
<th>Pro</th>
<th>Con</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ standard used for open-source projects</td>
<td>- slower, because they require maintainer to incorporate changes</td>
</tr>
<tr>
<td>+ allows to incorporate changes into &quot;read&quot; only repos, i.e. not everybody needs push access</td>
<td></td>
</tr>
<tr>
<td>+ less &quot;code conflict&quot; friction</td>
<td></td>
</tr>
</tbody>
</table>

Note: a forked repo is a "server-side" cloned repo
15.18 Feature branch workflow

⇒ best for small teams. Have 1-2 senior engineers who merge in pull requests

⇒ Each developer creates for a feature a separate branch and makes a pull request

<table>
<thead>
<tr>
<th>Pro</th>
<th>Con</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ master branch not disturbed by development</td>
<td>- develop vs. production?</td>
</tr>
<tr>
<td>+ pull requests/reviews</td>
<td>- feature vs. hotfix?</td>
</tr>
<tr>
<td>+ easy to manage</td>
<td>- release tracking?</td>
</tr>
<tr>
<td>+ good for internal projects</td>
<td>- dirty master branch?</td>
</tr>
</tbody>
</table>
15.19 Gitflow workflow

One master branch

One develop branch

One temporary branch for each release

One feature branch for each feature

One temporary hotfix branch for each hotfix

Gitflow Workflow
15.19 Gitflow workflow

semantic versioning

v0.1 => v0.2 => v1.0

Development

Release

Feature Branch Workflow
15.19 Gitflow workflow

⇒ naming conventions:
  feature branch     feature/<name>
  hotfix branch      hotfix/<name>
  release branch     release/v1.0

⇒ for practical management, there are plugins to support this workflow explicitly in git
https://github.com/nvie/gitflow

<table>
<thead>
<tr>
<th>Pro</th>
<th>Con</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ separate release and dev</td>
<td>- need to follow conventions to work smoothly</td>
</tr>
<tr>
<td>+ no dirty branch history</td>
<td>- many branches, overkill for small projects</td>
</tr>
<tr>
<td>+ good for product with release base</td>
<td></td>
</tr>
</tbody>
</table>
Merging or rebasing?
Some persons argue you should always rebase on the master before you file a pull request.

- this is more about faith than arguments, both solve the same problem

- squash your commits when you make a pull request!

<table>
<thead>
<tr>
<th>Rebase</th>
<th>Merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ clean, linear history</td>
<td>+ clear history, shows exactly what happened</td>
</tr>
<tr>
<td>+ scales well with many developers/branches</td>
<td>+ non destructive</td>
</tr>
<tr>
<td>+ no extra merge commit</td>
<td>- leads to polluted and difficult to understand history</td>
</tr>
<tr>
<td>- more difficult, many developers make mistakes</td>
<td>when many branches/developers are involved in a project</td>
</tr>
<tr>
<td>- reverting commits is difficult</td>
<td>- extra merge commit</td>
</tr>
<tr>
<td>- destructive operation</td>
<td></td>
</tr>
</tbody>
</table>
Final words

⇒ don't push blindly to a remote, always examine first what you did.

→ fixing branches on a remote is difficult and may screw up your team member's working copies.
End of lecture.

Next class: Tue, 4pm-5:20pm @ CIT 477