Git Local Commands and Workflows

Setup:

1. git clone https://github.com/yanivkrol/git-training.git
2. git config (--global) core.editor "/Applications/IntelliJ\ IDEA.app/Contents/MacOS/idea --wait"
3. Jetbrains commit window setup

Good to know

- `4fh6k3df` - Example of commit hash
- `HEAD` - Reference to the latest commit in the working branch
- `master` / `main` - Reference to the latests commit in master/main
- `<hash>^` - Reference to one commit prior to the commit with the given hash. E.g. `4fh6k3df^`, `HEAD^`. Adding more `^` will go further backwards, for example, 3 commits behind is `<hash>^^^`
- x number of `^` s can be replaced by `~x`. e.g. `HEAD^^^` == `HEAD~3`
- A branch name is also a reference to specific hash (the HEAD of the branch)

Tips and tricks:

- `HEAD^` or `HEAD^^` can be used to quickly reference one of the latest commits in the working branch.

Commands

`git fetch`

Fetch changes of the branch without merging them.

Options:

- `<hash>` specify a different branch for fetch
- `–a` Fetch changes of all branches

`git pull`

Like `git fetch` but also merges/rebase (configurable).

- `<hash>` specify a different branch for pull

`git push`

Push changes of the current branch to remote

Options:

- `<hash>` specify a different branch to push
- `-f` Force push when history changes (amend or rebase)

`git checkout <hash>`

Changes the working branch to the specified commit. Usually used with branch name, but can be used with a hash as well.

Options:

- `-b <name>` - Creates a new branch with the specified name

Tips and tricks:

- like the `cd` command, `git checkout -` checks out the previous branch

```
git status              > branch1
git checkout branch2    > branch2
git checkout -          > branch1
```

`git status`

Displays the state of the working directory and the staging area. Also shows statuses of merges, rebases, cherry-picks, etc.

`git log`

Displays history of the repository. Can be used to quickly see how our branch looks and look for specific commits to use `fixup` with.

Options:

- `-x` - Limits the number of commits shows, counting from HEAD inclusive (`git log -5` - shows last 5 commits)
- `--oneline` - One line per commit. Displays only the commit hash, branch name, and commit message.

Tips and tricks:

- `git log --oneline master..HEAD` - to fully show current branch

`git branch`

Shows branches (local only by default)

Options:

- `-m <new-name>` - Rename branch
- `-d <branches...>` - Deletes merges branches (merged into current branch)
- `-D <branches...>` - Like `-d` but "forced", deletes unmerged branches too

`git add <files> <directories>`

Stages specified files nad directories.

Tips and tricks:

- Staged changes using the Jetbrains commit window will be actually staged (as opposed to the regular commit window). This lets us easily stage **specific lines**

`git commit`

Captures a snapshot of the project's currently staged changes.

Options:

- `-m` - Specify a message (`git commit -m "my message"`)
- `-a` - Commit all modified files (automatic staging)
- `--amend` - Adds the changes to the last commit, also editing the commit message. If used without changes, only edits the message. `--no-edit` Can be added.
  After an amend, the commit hash changes, which means you'll have to `push -f`
  (Assuming original branch is on remote)
- `--squash <hash>` - Creates a commit that will later be "squashed" together with the commit specified, merging their changes and their messages
- `--fixup <hash>` - Same as `squash` but using the specified commit's message instead of merging them. This will be in greater use than `squash`

`git commit`

Tips and tricks:

- `git commit -a --amend --no-edit` - quickly add a missing semi colon, or any other changes for which there's no reason to create a new commit
- `git commit -a --amend --no-edit && git push -f` - Same as above but when you've already pushed (for example build failed because of missing colon)

`git reset <hash>`

Change the HEAD of the branch to be the specified commit, all "lost" changes will be unstaged

Options:

- `--soft` - All "lost" changes remain staged
- `--hard` - All "lost" changes are **permanently removed** (be careful)

`git stash` and `git stash pop`

Temporarily shelves (or stashes) changes you've made to your working copy so you can work on something else, and then come back and re-apply them later on

Options:

- `-u` - Also stash new files

Tips and tricks:

- Don't use unless you are going to pop in the next 2 minutes (It's black hole).

  Alternative:

  ```
  git add . && git commit -m "stash"
  do stuff...
  git reset HEAD^
  ```

- Can be used across branches

`git rebase <hash>`

Moves current branch to be based of the specified commit. "Rewriting history"

Options:

- `-i` `--interactive` - Enables controlling what happens with each commit above the base commit
- `--autosquash` - When using `-i`, automatically moves squash or fixup commits to their correct places in the interactive editor

`git rebase <hash>`

Tips and tricks:

- `git rebase master` - To stay updated
- `git rebase master -i (--autosquash)` - Interactively edit the entire branch
- After a rebase, the commit hashes change, which means you'll have to `push -f` (Assuming original branch is on remote)
- When rebasing a remote branch and the history was changed, use `-i` and drop all local "duplicate" commits (otherwise conflicts will be horrible)
- If at any stage during a rebase you are not 100% sure how to resolve a conflict or got lost, `git rebase --abort`

`git cherry-pick <hash>`

Adds the specified commit to the top of the current branch

Tips and tricks:

- Can be used to apply temporary changes that we store in dedicated branches

```
git checkout -b stg-settings
git commit -m "stg kafka settings"
git commit -m "stg db password"
...
git checkout feature-branch          > Assume a few commits already
git cherry-pick stg-settings         > We now have the staging settings in our branch
...
git reset <HEAD original hash>       > discards the staging settings
```