Leveraging Git

for your development workflow

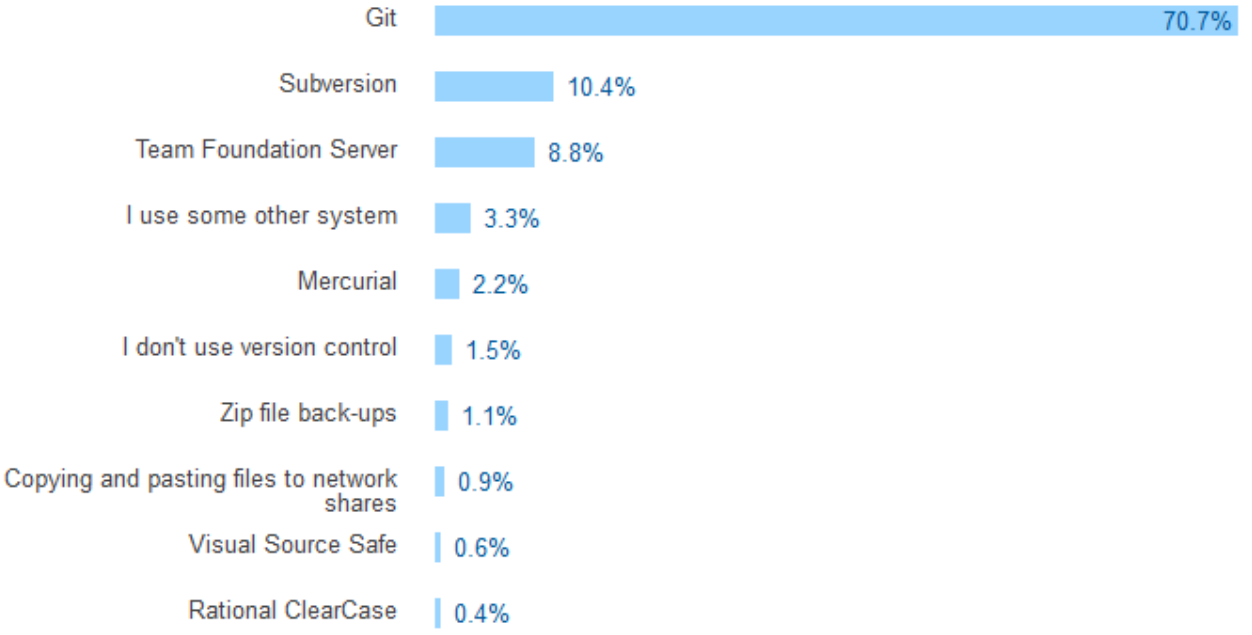
Agenda

1. **Introduction**
2. **Preparing your Commits**
3. **Extending and Adapting**
4. **Rewriting History**
5. **Don't Panic**
6. **Further pointers**

Source control is *essential* for, but not limited to, multi-developer software projects.

What professional Developers use in 2017

Git is used by the Linux Kernel project, Google, facebook, Microsoft, Twitter, LinkedIn, Netflix, Eclipse Foundation, Android, etc...



Source: Stack Overflow Developer Survey 2017

Key points

Git is

- distributed
- fast
- powerful
- widespread
- well-proven
- free

So you should probably use it!

Tooling
Basic Setup

My personal setup for Windows:

- **Git for Windows**
- **cmder**
  - Unix-like shell based on ConEmu
  - comes with Git for Windows
- **VS Code**

Tooling

Graphical User Interfaces

- Nice to visualize History
- Can help with some operations
- Never as powerful or precise as the CL
- May hide or rename operations
- Examples:
  - Tortoise Git
  - Sourcetree
- Note: git comes with gitk

Tooling

Hook your tools of choice into Git

You can define the default editor via the config too:

```
$ git config --global core.editor code --wait
```

`--wait` flag is needed with VS Code, otherwise the window will close right again

The same goes for individual diff/merge tools:

```
$ git config --global diff.tool bc
$ git config --global difftool.bc.path "C:\Program Files\Beyond Compare 4\BComp.exe"
```

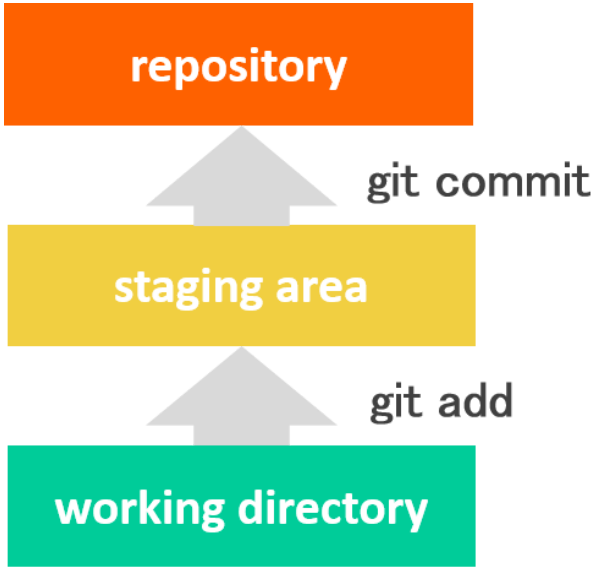
Tip: Beyond Compare lets you compare things like images as well

Preparing your commits

Preparing your commits

Of course there's also a layer diagram

Git knows three areas where your changes can be:



The stash can be viewed as a fourth area

Preparing your commits

Setting up the stage

```
$ git add <file or directory>
```

To add a specific file or directory.

```
$ git add -A
```

Adds all unstaged files, including untracked (new) ones.

Preparing your commits

Have full control

```
$ git add -p
```

Patch mode lets you stage *parts* of a changed file

- Git splits your changes into 'hunks'
- Steps through those hunks
- You decide for every single change if you want to stage it
  - You can even edit them!

Use it to **split** larger changes and **review** your work.

Btw. you can also `git reset -p`

Extending and Adapting

Extending and Adapting

Alias: Not the one with Jennifer Garner

Aliases allow you to save complex commands under a simpler name.

There are *two* Git configs (*.gitconfig*):

- The repository (local) config
- The global config

To add something to the config:

```
$ git config [--global] <type> <value>
```

To add an alias

```
$ git config [--global] alias.<name> <command>
```

Aliases

Be creative!

They can save you time:

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
$ git config --global alias.ap add -p
```

Or nerves:

```
$ git config --global alias.gerp grep
```

Use aliases to adapt the default behaviour in a way that suits you:

```
$ git config --global alias.stash stash --include-untracked
```

Adapting Git to your needs

Aliases for chained commands

The `!` prefix lets git execute the command in the shell. This allows you to

- chain multiple git commands
- include shell commands
- use parameters

```
$ git config alias.cma "!git add -A; git commit -m"

$ git config alias.findBranch "!git branch | grep -i"
$ git findBranch JIRA-123
```

This helps you find the correct feature branch for an issue.

Tip: Wrap more complex commands into a shell function:

```
[alias]
bclean = "!f() {
    git branch --merged ${1-master}
    | grep -v \" ${1-master}$\"
    | xargs -r git branch -d; }; f"
```

This cleans up all merged branches.

Adapting Git to your needs

Aliases for providing structure

Aliases for [semantic commit messages](#) including Issue ID:

```
[alias]
feat     = "!f() { git commit -m \"$1 - feat: $2\" }; f"
docs     = "!f() { git commit -m \"$1 - docs: $2\" }; f"
chore    = "!f() { git commit -m \"$1 - chore: $2\" }; f"
fix      = "!f() { git commit -m \"$1 - fix: $2\" }; f"
refactor = "!f() { git commit -m \"$1 - refactor: $2\" }; f"
```

```
$ git chore JRA-123 "updated build script"
[master 2ff195d] JRA-123 - chore: updated build script
```

Adapting Git to your needs

Aliases for recurring tasks

For your daily standup:

```
[alias]
standup = !git log --all --author=$USER --since="9am yesterday" --format=%s
```

Source: Tim Pettersen from BitBucket

Adapting Git to your needs

Aliases for recurring tasks

For your daily standup:

```
[alias]
standup = !git log --all --author=$USER --since="9am yesterday" --format=%s
```

Source: Tim Pettersen from BitBucket

```
lazy-standup = !git standup | say
```

Adapting Git to your needs

Sharing Aliases in your project

Use Git of course!

Store them e.g. in a separate repository

Then link them in your local or global config via `[include]`

```
[include]
./path/to/your/repository
```

Note: do not include random stuff - it's code you execute locally

Rewriting History

Rewriting History

Things can get messy

Rewriting History

Erasing your mistakes

```
$ git commit --amend
```

To *add* the staged changes to the latest commit.

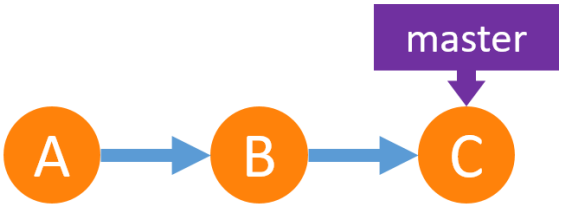
Lets you update the commit message too.

```
$ git commit --amend -C HEAD
```

To reuse the current commit message. Alias worthy!

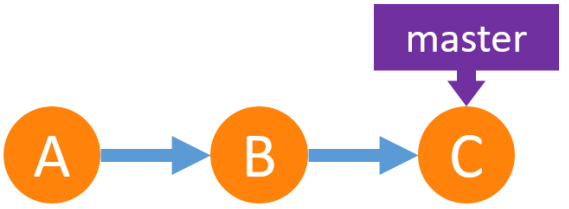
Rewriting History

Example for amending commits

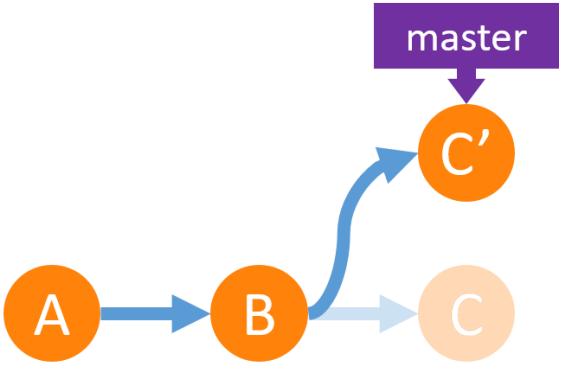


Rewriting History

Example for amending commits




```
$ git commit --amend
```



Rewriting History

Example for amending commits

Goal: maintaining a **linear** project history

Rebasing is saying

> “I want to base my changes on what everybody has already done.”

Rewriting History

Example for amending commits

Goal: maintaining a **linear** project history

Rebasing is saying

> “I want to base my changes on what everybody has already done.”

```
$ git rebase <branch>
```

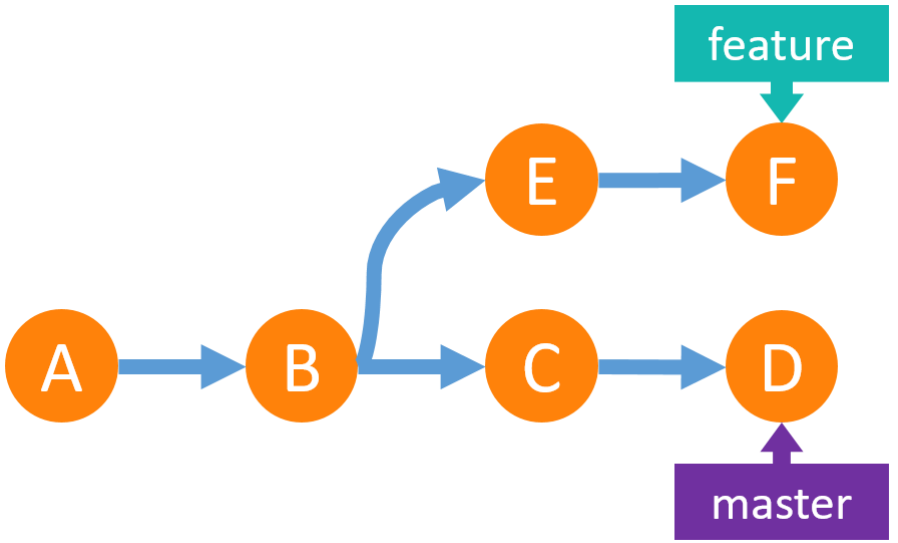
E.g.

```
$ git checkout feature
$ git rebase master
```

- goes to the common ancestor (where `feature` branched from `master`)
- takes all commits which are not in `master`
- applies them onto latest commit on `master`

Rewriting History

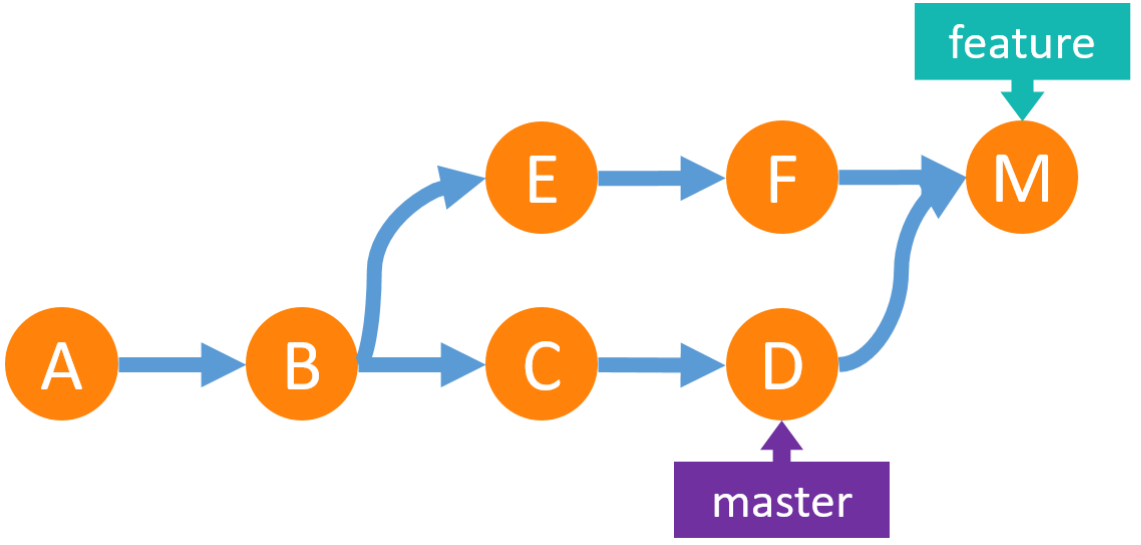
Merging example



Initial state

Rewriting History

Merging example

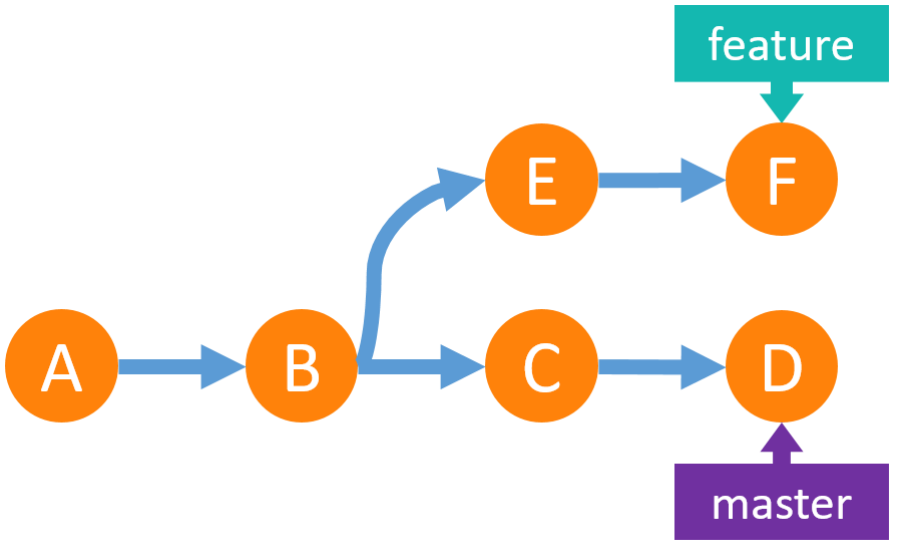



```
$ git checkout master
$ git merge feature
```

Results in a merge commit

Rewriting History

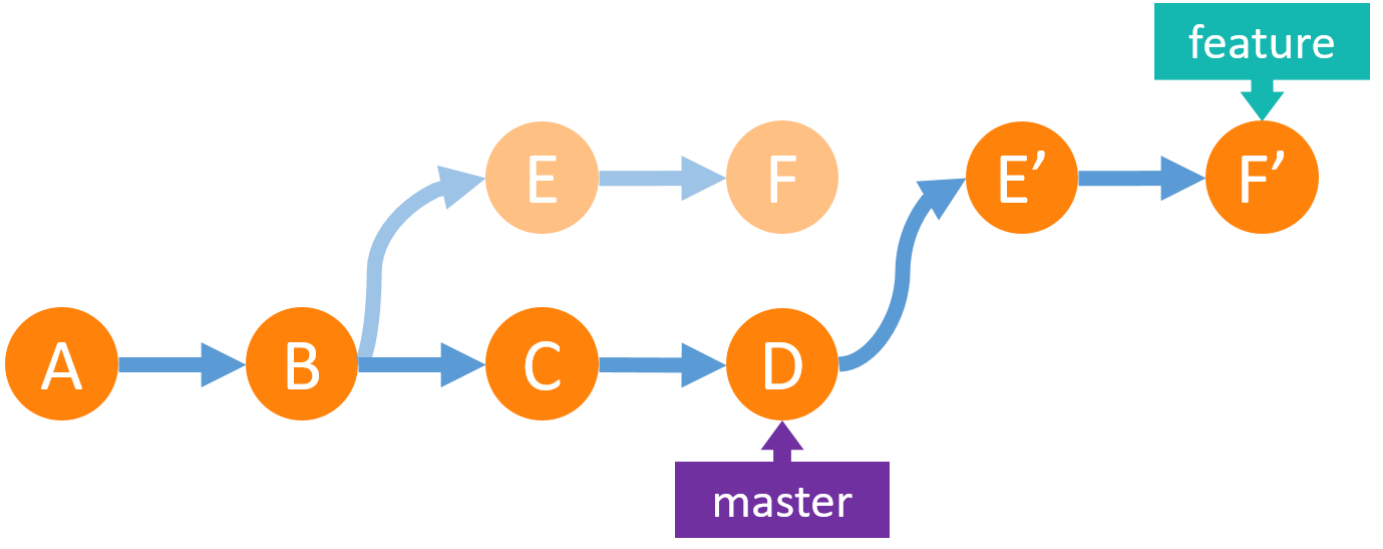
Rebasing example



Let's try this again

Rewriting History

Rebasing example

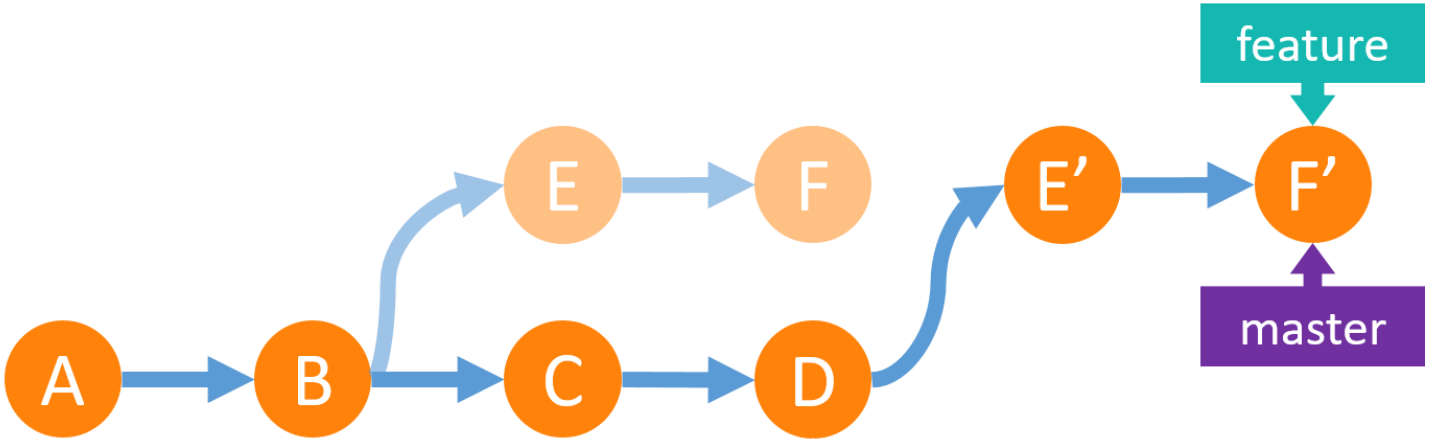



```
$ git checkout feature
$ git rebase master
```

Reapplies the change in `feature` onto `master`

Rewriting History

Rebasing example

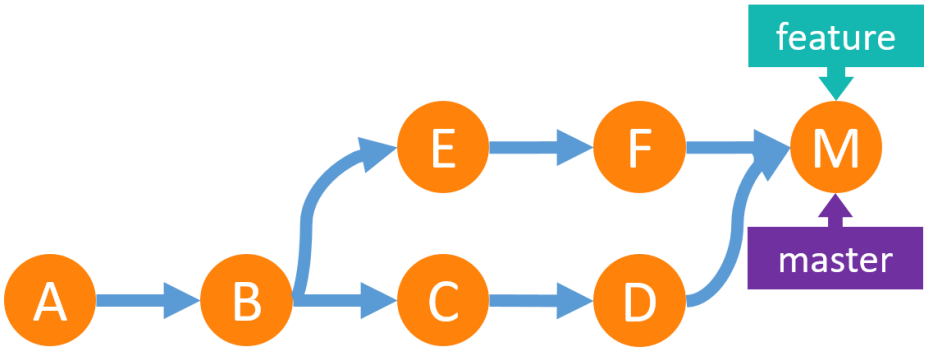



```
$ git checkout master
$ git merge feature
```

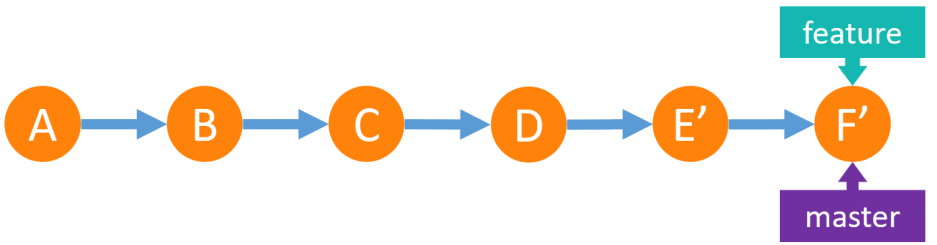
`master` can now be *fast forwarded* to `feature`

Rewriting History

Merging vs. Rebasing example



vs



Rewriting History

Interactive Rebase: Refactor your history

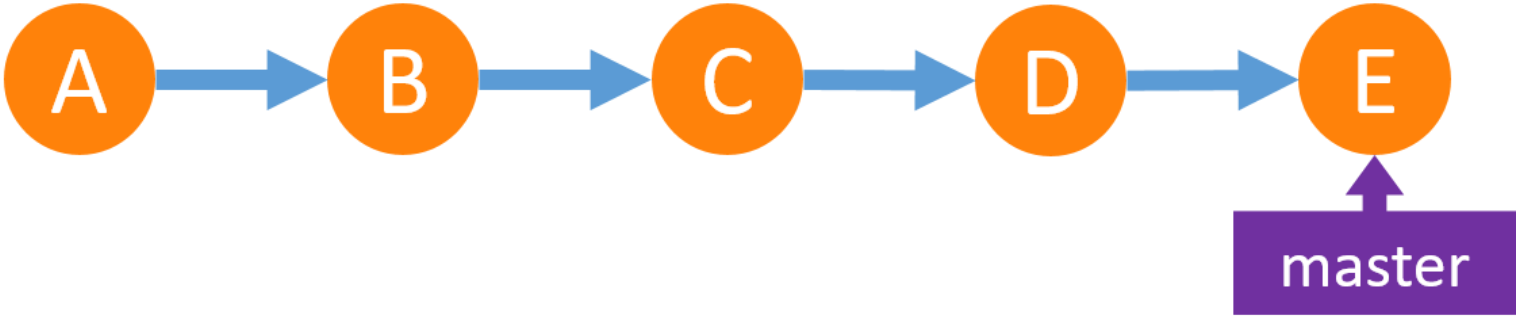
```
$ git rebase -i HEAD~3
```

```
pick f7f3f6d JRA-123 set up basic skeleton
pick 310154e JRA-123 implemented parts of the feature
pick a5f4a0d JRA-123 added some documentation

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Rewriting History

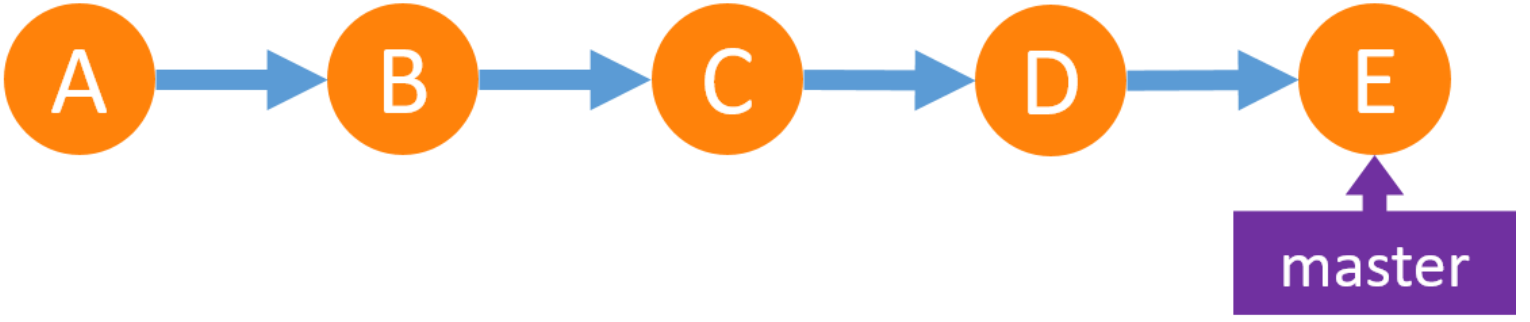
Interactive Rebase: Example



Initial state

Rewriting History

Interactive Rebase: Example

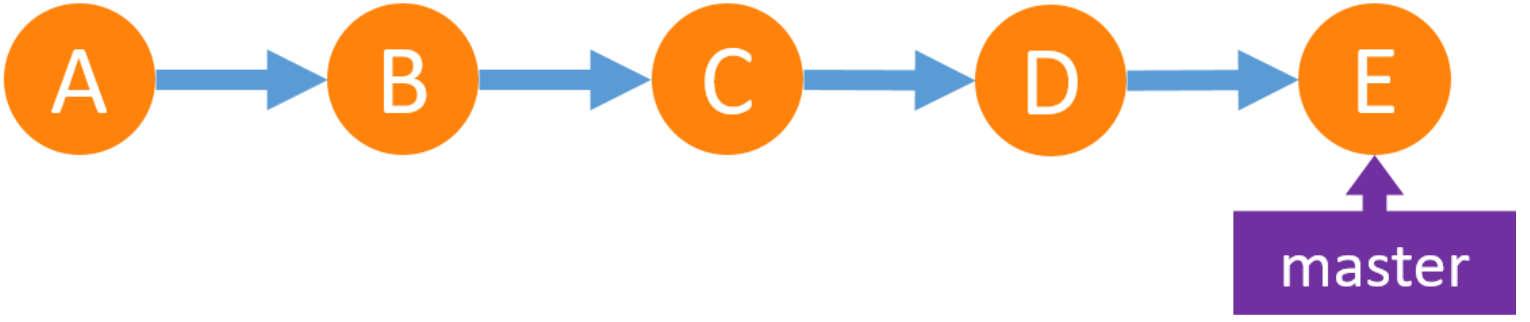



```
$ git rebase -i

pick B  JRA-123 changed config
pick C  JRA-123 implemented parts of the feature
pick D  JRA-123 implemented other parts of the feature
pick E  JRA-123 added some documentation
```

Rewriting History

Interactive Rebase: Example

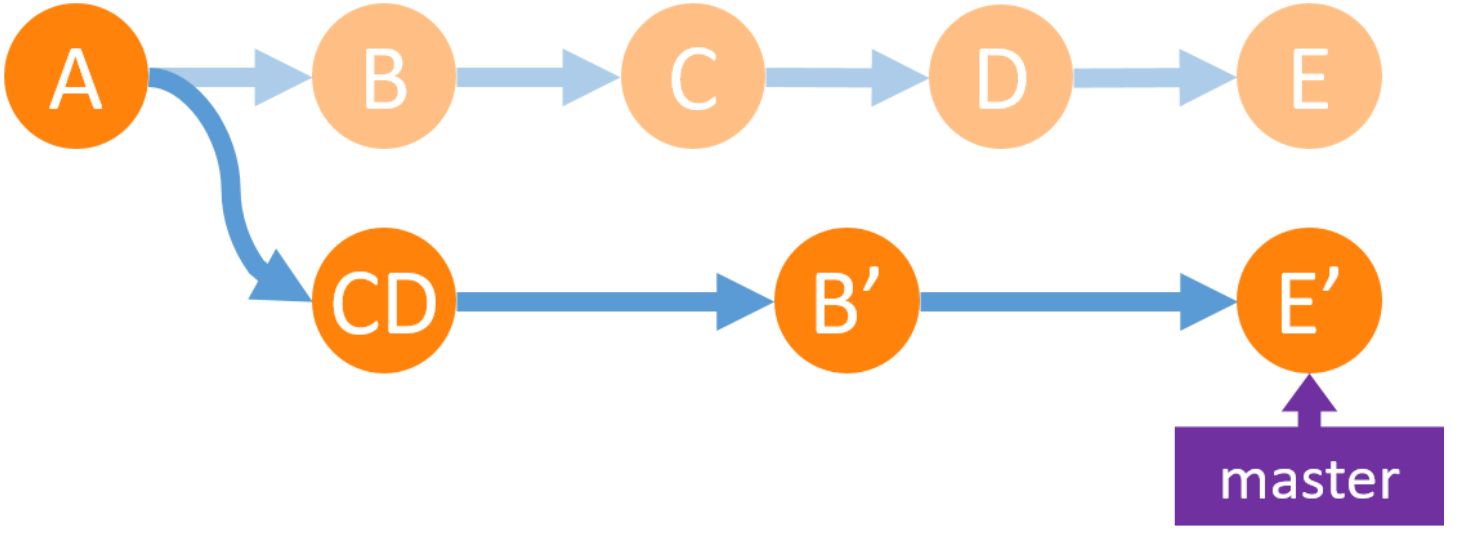



```
$ git rebase -i

pick C  JRA-123 implemented parts of the feature
squash D  JRA-123 implemented other parts of the feature
pick B  JRA-123 changed config
edit E  JRA-123 added some documentation
```

Rewriting History

Interactive Rebase: Example




```
$ git rebase -i

pick C  JRA-123 implemented parts of the feature
squash D  JRA-123 implemented other parts of the feature
pick B  JRA-123 changed config
edit E  JRA-123 added some documentation
```

Rewriting History

History is written by winners

Changing your log is actually encouraged!

Git offers powerful tools to do so:

- `git commit --amend`
- `git rebase`
- `git rebase -i`

This allows you to get rid of all those *"added comment"*, *"fixed typo"*, *"stylecop"* commits

Important: only update the history of your **private** branches

Don't Panic

Don't Panic

Git never forgets

Git makes sure that **anything you committed is safe**

1. A commit is defined and adressed by its hash
2. Commits are never modified, any change results in a *new* commit with a new hash.
3. Commits can be unreachable, but are not deleted *

There are only two potentially destructive commands:

`git reset --hard` and `git checkout`

* Unreachable commits do eventually get garbage collected

Don't Panic

Reflog: The saviour of lives

The **Reflog** is a log of all operations performed in a repository.

```
$ git reflog

05ca429 HEAD@{11}: rebase -i (finish): returning to refs/heads/master
05ca429 HEAD@{12}: rebase -i (pick): test: fixed some unit tests
51c02dc HEAD@{13}: rebase -i (squash): refactor: removed unused imports and variables
caa3df9 HEAD@{14}: rebase -i (start): checkout HEAD~5
65586a3 HEAD@{15}: rebase -i (finish): returning to refs/heads/master
65586a3 HEAD@{16}: rebase -i (start): checkout HEAD~5
65586a3 HEAD@{17}: rebase -i (finish): returning to refs/heads/master
65586a3 HEAD@{18}: rebase -i (start): checkout HEAD~5
65586a3 HEAD@{19}: commit: removed TODO
cc71c4c HEAD@{20}: commit: test: fixed some unit tests
caa3df9 HEAD@{21}: commit: refactor: removed unnecessary imports
ec563bf HEAD@{22}: reset: moving to HEAD~1
79a1c6f HEAD@{23}: commit: refactor: removed unnecessary imports
ec563bf HEAD@{24}: commit: refactor: Minor refactorings
f2f056f HEAD@{25}: commit: chore: added tslint rules to find unused code
```

It logs the commit in which the operation *ended* too, so you can reset to it anytime:

```
$ git reset 65586a3
```

Further pointers

Further pointers

Just a dump of some more stuff

`git bisect`

- performs a binary search over a part of your history - great to pinpoint bugs

`git rerere`

- lets you record changes you made during merges and applies them automatically the next time

`git filter-branch`

- lets you scrub the entire history
- useful if you e.g. want to
  - remove a file from the entire history
  - change an authors email adress globally

Git LFS

- An extension for versioning large files
- If you need to store huge assets in your repository

Links

Tools:

- Official Website: https://git-scm.com
- Git for Windows: https://git-for-windows.github.io
- cmder: http://cmder.net

Learning Resources:

- Think like (a) Git - A Guide for the Perplexed: http://think-like-a-git.net
- Interactive branching tutorial & sandbox: http://learngitbranching.js.org
- Atlassian Git Tutorials: https://www.atlassian.com/git/tutorials

Find the extended version of these slides at https://github.com/ymekesser/leveraging-git-presentation

Sources

- Git Logo by Jason Long is licensed under the Creative Commons Attribution 3.0 Unported License.
- Stack Overflow Developer Survey Results 2017
- Tips and Tricks: Gotta Git Them All - GitHub Universe 2016
- Git Aliases of the Gods! - Git Merge 2017
- Atlassian Git Tutorials