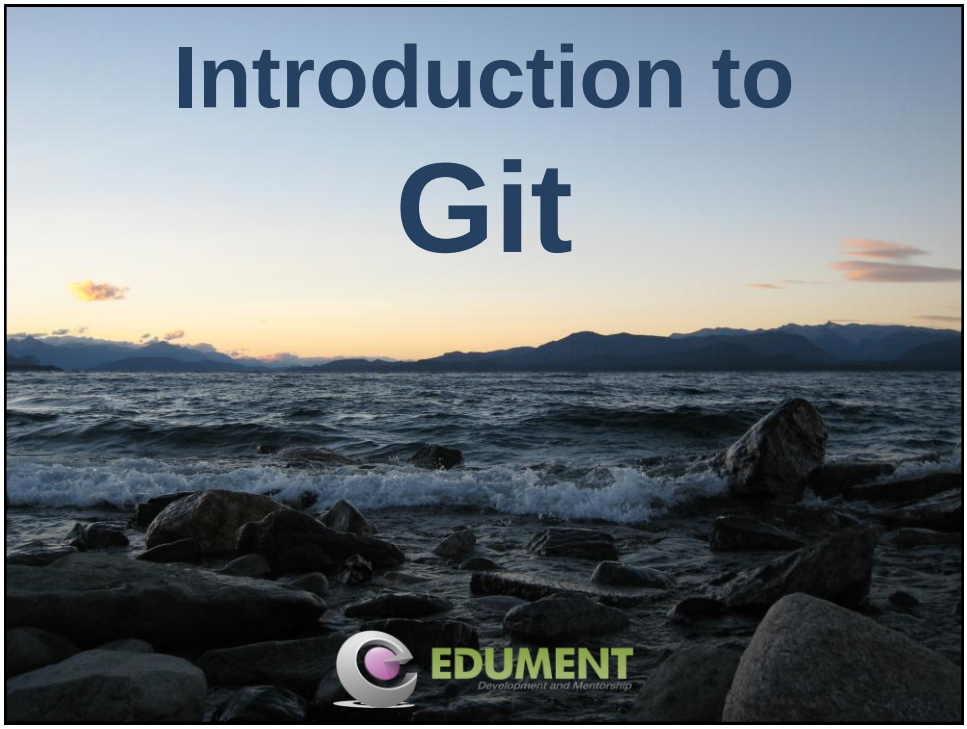


Course Objectives

Git looks so confusing!

Why another VCS?!

Commit AND push?!

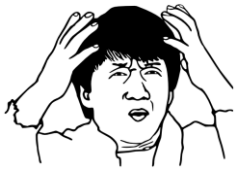
Take Introduction to Git Course

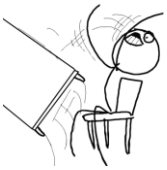
Git is pretty easy!

Hey, this is way better!

DVCS is well worth it!

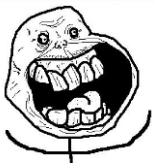
Illustrated Course Objectives





Take Introduction to Git Course





About the teacher

[generic teacher introduction]

Course Roadmap

1. Introduction

Version Control: what and why?

A little history

2. Baby Steps: Working Locally

Creating a repository

Working with commits

Inside Git: DAGs and SHA-1s

Branching and merging

3. Getting Distributed

Adding remotes

Pushing and pulling

Handling conflicts

4. Getting Awesome

Amending existing commits

Stashing and rebasing

Cherry-picking

Version Control: What and why?

What is version control?

Version control systems are tools that help us to...

Keep track of the history of changes to a set of files (source code, documentation, whatever)

Handle multiple developers working in parallel on that set of files in a safe and efficient manner

Cope with complex development workflows, such as bug fix releases to old versions of the software, isolated development and testing of new features and customer specific versions

Why does history matter?

Means that it is possible to answer questions about how the software got to its current state

Can find out who is responsible for a particular change or piece of code, if you need to ask something about it

Helps with implementing a system of code review; a good commit should be a good unit of review

Part of implementing continuous integration (e.g. can automatically test each commit)

Multiple developers

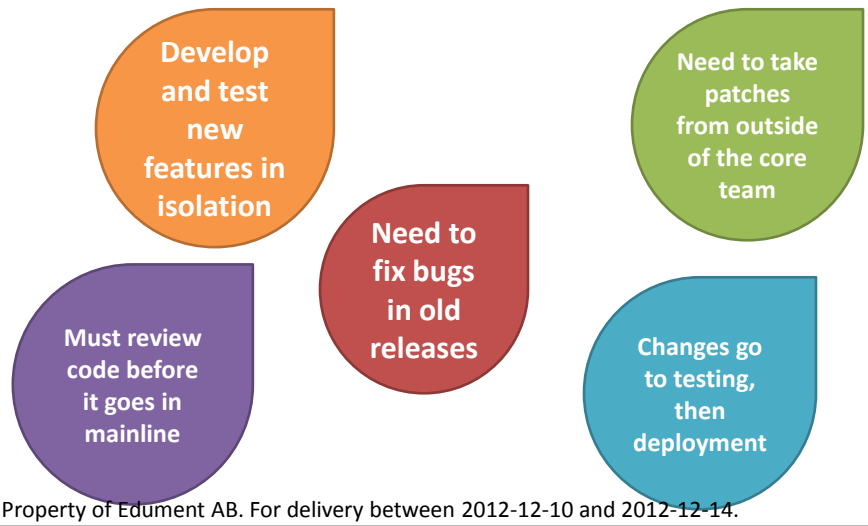
As soon as more than one person is working on a system, there is a risk of conflicting changes

Two people changing the same file but in different places should not be a problem
➔ want a version control system to handle the easy cases automatically, without fuss!

Real conflicts should be flagged up, so they can be resolved correctly
➔ want a version control system to tell us when we need to intervene and fix conflicts by hand

Development Workflows

A good VCS should support the way you develop



A Little History

RCS

Amongst the earliest implementations of version control, yet still in some use today!

Only works on individual files; mostly useful for individual documents, scripts and configuration files rather than large software projects

No central repository; all operations are local

Had branching support, but considered awkward by many users - and again, it was only branching at the level of an individual file

CVS

Developed in the mid-80s, and was highly popular well into the late 90s; still in active use today

Client-server system; server holds the entire history, and clients have a single working version

Can version entire sets of files - but version number is still per file, commits are not atomic and renaming/moving is not versioned

Branches are expensive; the assumption was that they would be rare, short-lived or historical

Subversion (aka. SVN)

Aims to be familiar to CVS users, but fixing longstanding issues and supplying new features

In widespread use today

Similar client-server approach to CVS, however commits are atomic commits and version number is for the whole repository, not individual files

Branches and tags both implemented by copying, with the result that they are still expensive; worse, tags can be committed to!

DVCS: have your cake and eat it

Feature	Local (e.g. RCS)	Centralized (e.g. SVN)	Distributed (e.g. Git)
Full history available locally	YES	NO	YES
Can perform all operations locally	YES	NO	YES
Support having a central repository to share work	NO	YES (It's required)	YES (But it's only a convention)
Support complex topologies, e.g. repositories for dev. and testing	NO	NO	YES

So really, distributed means...

Everybody has the full version history locally and can manipulate it as they wish

- Everyone serves as a backup
- Can query history locally - and fast!
- Can commit and branch locally

We can efficiently move one or more commits between different copies of a repository

- Can designate a copy as "central"
- Can implement rich workflows
- Outsiders can work versioned

Branching and merging

Many development workflows depend upon branching and merging

In just about all version control systems, creating a branch is easy - though it may be slow
(e.g. SVN wants to make a complete copy)

Generally, the difficult part is merging

With Git, branch creation is instant, and it works hard for you on merging, usually only leaving you conflicts that genuinely need human attention

Branches in the DVCS world

If you have never worked with a DVCS before...

...please now forget everything you think you know about branches! ☺

Especially, stop believing all of the following:

Branches are expensive
Only create a branch if you really need it
Merging is to be avoided because it's hard

Effective Git users work with branches every day!

Getting Started

Local Operations

For the first part of the course, we will be working with Git in an entirely local setting

This means that you will be creating a repository and performing operations on your own machine

Due to its distributed nature, Git allows you to do everything locally, from simple things like commits and history viewing through to more complex operations such as branching and merging

Very different from centralized version control!

git init

Creating a Git repository takes about 15 seconds!

1. Create a directory

```
$ mkdir my-first-git
```

2. Change into that directory

```
$ cd my-first-git
```

3. Tell Git to create a repository there

```
$ git init
```


What just happened?

The `init` command creates a `.git` directory

This is where the entirely version history of the repository will be stored

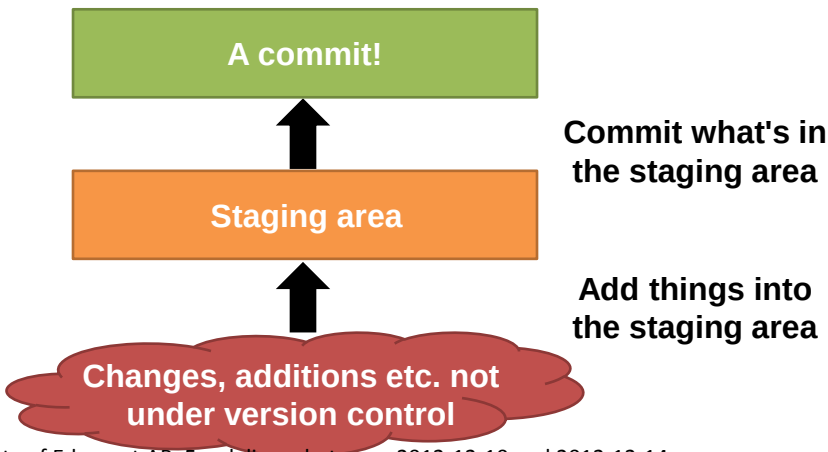
Also contains a `config` file, which we will not have to touch for now

The `.git` folder contains everything that matters. Want to give your full version history to a friend? Just zip up `.git` and send that; the "current version" can be recreated from what's in `.git`.

Basic Operations

Adding and Committing

In Git, committing is a two-step process
(but often, you can perform both with a single command)



Adding new files

The `add` command places a file in the staging area

```
$ git add README
```

The `commit` command then takes the contents of the staging area, and creates a commit

```
$ git commit -m "Add a README."
```

```
[master (root-commit) beb9dfd] Add a README.
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 README
```


Committing changes

With existing files, add places the changes to that file into the staging area, so you can commit changes with two commands:

```
$ git add README
$ git commit -m "Update README."
```

Usually, however, this is done just with `commit`, specifying the file(s) to be committed:

```
$ git commit -m "Update README." README
```

Directories

If you use add with a file inside a directory, it will retain the directory structure

```
$ git add src/main.c
$ git commit -m "Start le coding!"
```

There is only one `.git` at the top level of the repository, not one inside every directory

Minor limitation: Git doesn't support putting empty directories under version control

git status (1)

Gives an overview of what is currently staged, changed in tracked files and untracked

```
# On branch master
# Changes not staged for commit:
#
#       modified:   src/main.c
#
# Untracked files:
#
#       src/util.c
#       src/util.h
```

These are changes to files Git knows about, but that are not staged

These are files Git does not know about

TIP: If you get fed up of typing status, alias it to st

```
$ git config --global --add alias.st status
```

git status (2)

Let's add one file and look at the status again...

```
$ git add src/util.h
$ git status
```

```
# On branch master
# Changes to be committed:
#
#       new file:   src/util.h
#
# Changes not staged for commit:
#
#       modified:   src/main.c
#
# Untracked files:
#
#       src/util.c
```

This file is now in the staging area

Not in staging, but a `commit` of `src` would include it

.gitignore

One annoyance is that after building our code, the generated files show up in the status output

```
# On branch master
# Untracked files:
#
#       main.exe
#       main.obj
#       util.obj
```

The solution is to add a `.gitignore` file

`.gitignore`

```
*.obj
*.exe
```

```
$ git add .gitignore
$ git commit -m "Add a .gitignore."
```

git diff (1)

The status command gives you an overview of your changes; diff gives the details

```
diff --git a/src/util.c b/src/util.c
index 9fe5927..e826a45 100644
--- a/src/util.c
+++ b/src/util.c
@@ -1,7 +1,7 @@
 #include <stdio.h>

-void print_ten_times(char *msg) {
+void print_n_times(char *msg, int n) {
     int i;
-    for (i = 0; i < 10; i++)
+    for (i = 0; i < n; i++)
         printf(msg);
 }
```

Red and the "-", indicate removed lines

Green and the "+" indicate added lines

git diff (2)

By default diff only shows unstaged changes; used the staged option to see the staged changes

```
$ git add src/main.c
$ git diff --staged
```

```
diff --git a/src/main.c b/src/main.c
index 5f8a873..81f2948 100644
--- a/src/main.c
+++ b/src/main.c
@@ -1,5 +1,5 @@
 #include "util.h"

 int main() {
-    print_ten_times("Hello, world!\n");
+    print_n_times("Hello, world!\n", 10);
 }
```

git diff (3)

The `stat` option will just give you statistics on the changes that have been made to each file

```
$ git diff --stat
```

```
src/main.c |    2 +-
src/util.c |    4 ++--
src/util.h |    2 +-
3 files changed, 4 insertions(+), 4 deletions(-)
```

Has many more options than this; see:

```
$ git help status
```


git log (1)

Shows a (automatically paged) log of all the commits, most recent commit first

```
commit e20962ebed7b0288922320f217a6a3ab9371727c
Author: jnthn <jnthn@jnthn.net>
Date:   Wed Apr 18 18:09:02 2012 +0200

    Add a .gitignore.

commit eae16e7a7f34d1208ca8267c2fabbbc1eb8e3640
Author: jnthn <jnthn@jnthn.net>
Date:   Wed Apr 18 17:56:55 2012 +0200

    Factor printing out to a utility file.

...
```

git log (2)

Has more options than you can possibly imagine; one fairly useful one is:

```
$ git log --oneline
```

```
e20962e Add a .gitignore.
eae16e7 Factor printing out to a utility file.
887f06c Start le coding!
869cec3 Update README.
8356287 Add a README.
```

To learn about (literally) dozens more, see:

```
$ git help log
```


git show

Shows you the full details of a commit

```
$ git show 869cec3
```

```
commit 869cec3f4e200ded3e9b7d27823cfd4da8cd086d
Author: jnthn <jnthn@jnthn.net>
Date:   Wed Apr 18 17:34:10 2012 +0200

    Update README.

diff --git a/README b/README
index 7062ca7..79a326b 100644
--- a/README
+++ b/README
@@ -1 +1 @@
-This project will be awesome!
+This project will be awesome! REALLY awesome!
```

Deleting files

If you use the `rm` command, then the deletion of the file is immediately placed into the staging area

```
$ git rm README
$ git commit -m "Toss the README."
```

Alternatively, you can just delete the file in your OS or IDE; a commit of the directory that contains it will then do the right thing


```
$ del README
$ git commit -m "Toss the README." .
```


Moving files

Once again, two options; the `mv` command which immediately stages the rename:

```
$ git mv src\main.c src\hi.c
$ git commit -m "Rename the program."
```

Or move it with your OS or IDE, and `git add` the new file; it will figure out it's the same file!

```
$ move src\main.c src\hi.c
$ git add src\hi.c
$ git commit -m "Rename the program." src
```

Exercise 1

This exercise is aimed at getting you comfortable with the commands that we have covered so far

It includes...

Creating a new repository
Adding new files
Committing changes
The status command
Looking at diffs and logs
Moving and deleting files

A Peek Inside Git

SHA-1 Hashes

Every commit is identified by a SHA-1 hash; we came across these when using `log` and `show`

```
commit e20962ebed7b0288922320f217a6a3ab9371727c
Author: jnthn <jnthn@jnthn.net>
Date:   Wed Apr 18 18:09:02 2012 +0200

    Add a .gitignore.
```

Full hash of the commit

The hash is derived from the content of the commit along with the commit that came before it

This means that the SHA-1 is not only unique locally, but unique over all copies of a repository!

Writing SHA-1 Hashes

We have also seen SHA-1 hashes show up in an abbreviated form

Abbreviated hashes

```
e20962e Add a .gitignore.
eae16e7 Factor printing out to a utility file.
887f06c Start le coding!
869cec3 Update README.
8356287 Add a README.
```

Usually, the first six characters are enough to uniquely identify a commit in a repository

You can provide as few or as many characters as you wish, provided they identify a single commit

Why not a revision number?

Using numbers would imply there is one unique ordering of commits

This is fine if you have a centralized version control system, since the server can decide who gets to commit first if there is competition

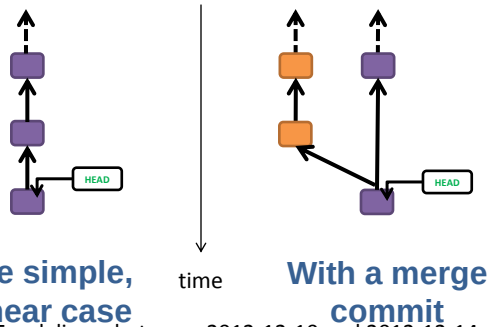
This doesn't make any sense in a distributed world, where different local copies will have commits that have not been shared yet

SHA-1 is always content-unique

The DAG

The version history is represented by a DAG (Directed Acyclic Graph) of commits

Each commit points to the commit that came before it (or "commits" when there is a merge)



The simple, linear case **With a merge commit**

And that's Git!

Just about everything in Git boils down to...

a. Commits that are identified by a SHA-1
b. Manipulating the DAG of commits

The rest of this course will show you a whole array of different commands and patterns

However, everything we will cover is really just about commits and placing them in graphs

The underlying model of Git is simple ☺

Local Branching and Merging

Creating and listing branches

Creating a branch in Git is easy, and also an extremely cheap operation

```
$ git branch calc-fibs
```

You can now get a list of branches that exist

```
$ git branch
```

```
  calc-fibs
* master
```

Notice the current branch is still `master`!

Switching to a branch

The `checkout` command is used to switch from one branch to another

```
$ git checkout calc-fibs
```

We can use the branch command again, to verify that we have indeed switched branch

```
$ git branch
```

```
* calc-fibs
  master
```

Committing on the branch

We make a change, and check the status

```
$ git status
```

```
# On branch calc-fibs
# Changes not staged for commit:
#
#       modified:   src/util.c
#
```

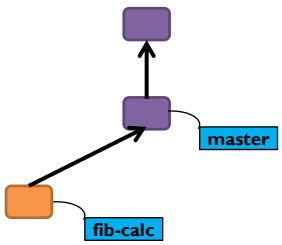
It notes we are on the calc-fibs branch. We now go ahead and commit the usual way...

```
$ git commit -m "Implement fib." src
```


So, what is a branch?

A branch is really just an alias for a commit

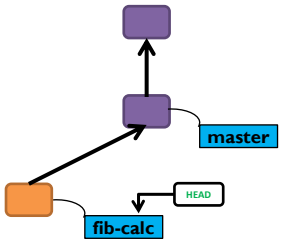
Given every commit knows its predecessor, then knowing the latest commit in a branch is enough to know the full history of the branch



HEAD

HEAD always refers to the current commit

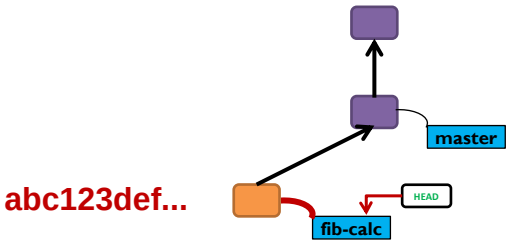
Usually, it does this by actually referring to the current branch, which in turn contains the SHA-1 of the latest commit on the branch



What a commit does (1)

First, HEAD is used to find the current branch
(provided there is one; there is also a detached HEAD mode, which we'll pass over for now)

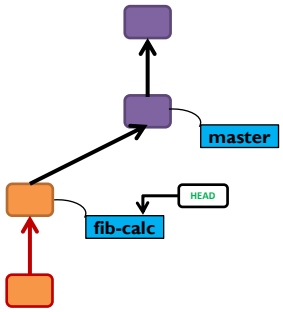
This in turn is used to get the SHA-1 of the current commit



What a commit does (2)

A new commit is created, with its parent set to the current commit, which we just located

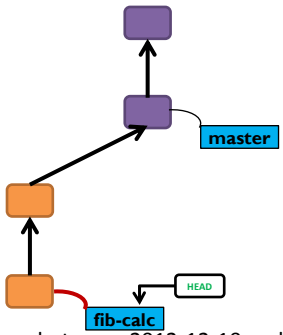
It is SHA-1'd and stored by Git



What a commit does (3)

Finally, the branch alias is updated to point to the new commit, so it becomes the branch's latest

Note that HEAD itself is not updated; it need not be as it points to the branch



Branch switching is "in place"

In some version control systems, different branches are stored in different directories, so moving between them is a directory change

With Git, using `checkout` to move to a different branch changes the contents of the repository directory in place, altering the files to represent the state they are in in the branch you switch to

Generally, this is a win; you can create a branch and start working in it without having to do a fresh build of your system!

Merging

Merging involves...

Doing a `checkout` of the target branch
Using the `merge` command

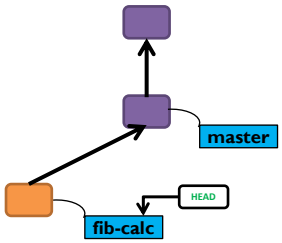
To merge fib-calc into master, we thus do:

```
$ git checkout master
$ git merge calc-fibs
```

```
Updating 4d0df83..8e66580
Fast-forward
 src/util.c |    7 +++++++
 1 files changed, 7 insertions(+), 0 deletions(-)
```

Fast-forward "merges" (1)

Consider the DAG at this point in time



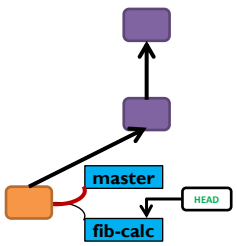
We have made no commits to master since we started the branch

This means we can do a "fast-forward" merge

Fast-forward "merges" (2)

In a fast-forward merge, we don't actually have to do any merging in the traditional sense

Instead, we simply update the master branch's current commit to alias the latest commit in the fib-calc branch - and we're done!



Fast-forward "merges" (3)

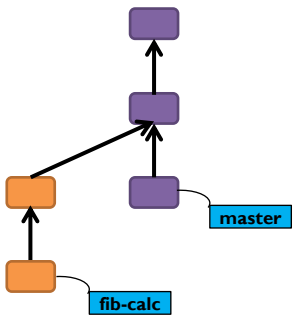
Fast-forward merges are cheap and completely painless - there's never going to be a conflict

Doing development work in a local branch can be useful if you need to urgently task-switch back to master for doing a bug fix, or you discover a need to work on some other pre-requisite first

TIP: If you're unsure whether you should do some work in a branch - Just Do It. If you turn out to need it, it's right there for you. If you don't need it, you get a painless, instant fast-forward merge.

Merge commits (1)

In many situations, there will have been commits in both branches; here we can see that master has a commit and fib-calc has a couple



A fast-forward is simply not possible here

Merge commits (2)

The output of a merge will look a little different:

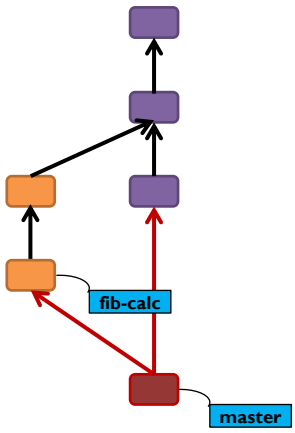
```
Auto-merging src/util.c
Merge made by recursive.
 src/hi.c   |    3 +++
 src/util.c |    7 +++++++
 src/util.h |    1 +
 3 files changed, 11 insertions(+), 0 deletions(-)
```

It doesn't mention fast-forward, but rather that there was a merge (by the recursive algorithm)

Additionally, we are informed that both branches changed src/util.c, but Git handled that by itself

Merge commits (3)

This kind of merge also produces a merge commit, which has a couple of parents

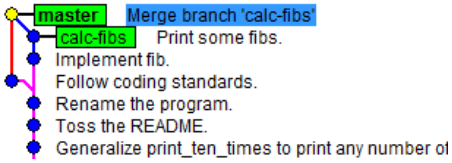


gitk

As well as the command line tool, Git ships with a couple of small, but useful GUI applications

One of them is `gitk`, which provides a nice way to explore the commit history

It also gives a visual representation of the DAG



Coping with conflicts (1)

There are some cases where a merge commit is needed, but Git can't completely work it out

For example, imagine calc-fibs has this commit:

```
 int main() {
     print_n_times("Hello, world!\n", 10);
+    printf("Fib 5 = %d\n", fib(5));
+    printf("Fib 10 = %d\n", fib(10));
 }
```

And master has this one:

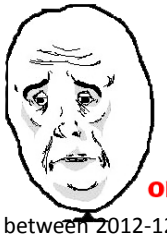
```
 int main() {
     print_n_times("Hello, world!\n", 10);
+    return 0;
 }
```

Coping with conflicts (2)

When we issue the `merge` command, we get some slightly scarier output:

```
Auto-merging src/hi.c
CONFLICT (content): Merge conflict in src/hi.c
Auto-merging src/util.c
Automatic merge failed; fix conflicts and then commit the result.
```

This demands...manual intervention!



Coping with conflicts (3)

We open the conflicting file(s) in our editor; the conflicts are indicated as follows:

```
int main() {
    print_n_times("Hello, world!\n", 10);
<<<<<<< HEAD
    return 0;
=======
    printf("Fib 5 = %d\n", fib(5));
    printf("Fib 10 = %d\n", fib(10));
>>>>>>> calc-fibs
}
```

The markers tell us which part came from which branch; we just edit the file to how it should be

Coping with conflicts (4)

After editing, the file looks like this:

```
int main() {
    print_n_times("Hello, world!\n", 10);
    printf("Fib 5 = %d\n", fib(5));
    printf("Fib 10 = %d\n", fib(10));
    return 0;
}
```

If we try and commit now, however, it won't work:

```
$ git commit -m "Merge in calc-fibs."
```

```
U       src/hi.c
fatal: 'commit' is not possible because you have unmerged files.
```


Coping with conflicts (5)

We have to inform Git that we have resolved the conflict to our satisfaction; this is done using the `add` command:

```
$ git add src/hi.c
```

At this point, the `commit` command can be issued, and this time it will be successful.

```
$ git commit -m "Merge in calc-fibs."
```

Exercise 2

This exercise will give you a chance to practice working with branches locally

Most importantly, it will give you a chance to get a feel for the different types of merge...

Fast-forwards
Merges that are fully automatic
Merges that need manual intervention

Do ask questions if you're unsure; branching and merging are key Git skills to master!

Getting Distributed

Remotes

Earlier on, it was mentioned that you can copy a Git repository, complete with history, by copying the contents of the `.git` directory

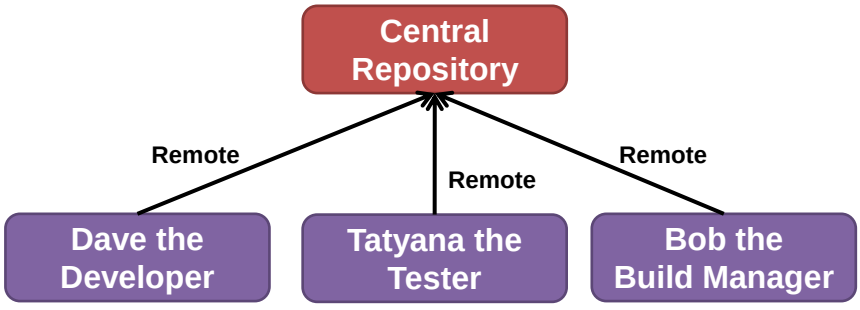
A remote is another copy of the current repository

It may have some commits we don't have
It may lack some commits we have

It should always be possible to trace back from the current latest commits locally and in the remote to a common ancestor

A central repository

The most common use of remotes is to set up a central repository



It's the central repository because everyone agrees it is. To Git, it's just another copy.

Hosted Git

There are a range of hosted Git services out there, which can host your central repository for you

GitHub is the largest today, offering free hosting for public repositories (used by thousands of open source projects) and private hosting for individuals and organizations

Gitourious is a lesser known alternative; some projects have chosen it over GitHub

You can also host it yourself, with a bit of setup

This course: GitHub

We'll use repositories on GitHub for the purpose of practicing with remotes

All of the commands and techniques we learn are completely transferable to working with any other remote, whether hosted internally or elsewhere.

Having a GitHub account also opens the door to contributing to thousands of open source projects; it may well come in useful to you far beyond this course. ☺

Adding a remote

After creating a repository on GitHub, we need to tell our local Git repository about it by adding it as a remote

```
$ git remote add origin …
git@github.com:user/repo.git
```

Since you can have multiple remotes, you have to specify a name as well as the address

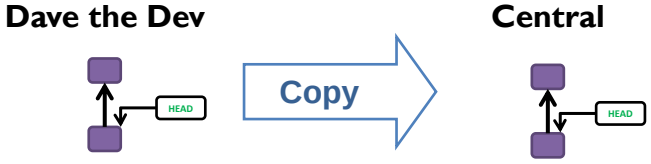
The convention used by almost all Git users is to call the central repository remote `origin`

The first push

Pushing is taking commits we have locally and copying them to a remote

If we have a new, empty, central repository then our first push should use the -u flag

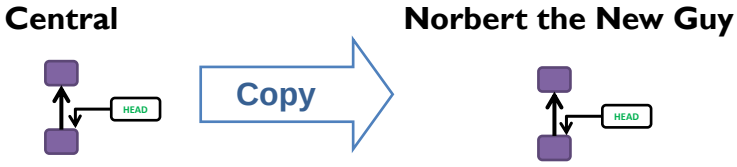
```
$ git push -u origin master
```



Cloning

The `clone` command is used when you want to get a local copy of a remote repository; it also sets up `origin` to point to the remote for you

```
$ git clone git@github.com:user/repo.git
```



You use clone once to get an initial copy of the remote; later you'll be using pull to synchronize.

Sharing changes (1)

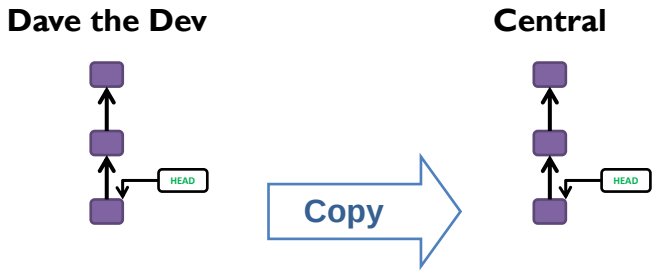
After you have done some work locally, you will have one or more commits that the central repository does not have

In this case, we need to copy the one extra, new commit over to the central server

Sharing changes (2)

The push command is used to do this:

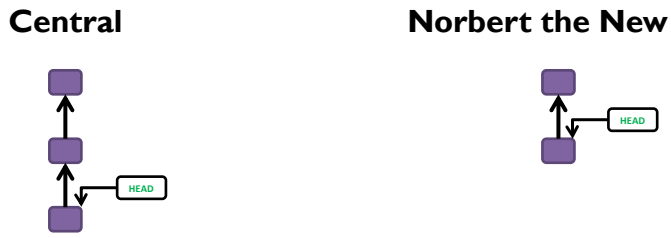
```
$ git push origin master
```



You don't need to tell Git which commit(s) it needs to copy. It always works that out for you. ☺

Getting the latest changes (1)

At this point, the central repository has a commit that Norbert the New Guy doesn't have in his local copy of the repository

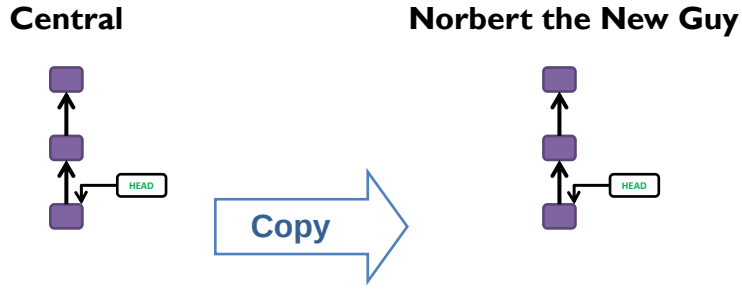


We need to copy that new commit from the central repository into the local copy

Getting the latest changes (2)

The `pull` command is used to `fetch` the latest changes and `merge` them into our local copy

```
$ git pull
```



Getting the latest changes (3)

`pull` really just calls two other commands for us:

`fetch`

This command fetches all of the changes from a remote that we don't know about locally

`merge`

We already saw this one! ☺

The exact same mechanism we used to merge local branches is also used to merge commits from a remote. Neat, huh? ☺

When `pull`ing gets tricky

Since a `pull` involves a `merge`, all of the cases of merging we saw earlier can come up again

Fast-forward

If you have no local commits, you always get a straightforward fast-forward merge

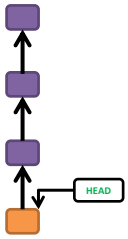
Merge Commit

When you have local commits and also bring in remote ones, the DAG looks just as it does in a branch situation. A merge commit is needed. Sometimes, manual intervention may be needed.

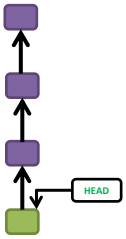
When `push`ing gets tricky (1)

Sometimes, both you and the remote repository may have extra commits

Central



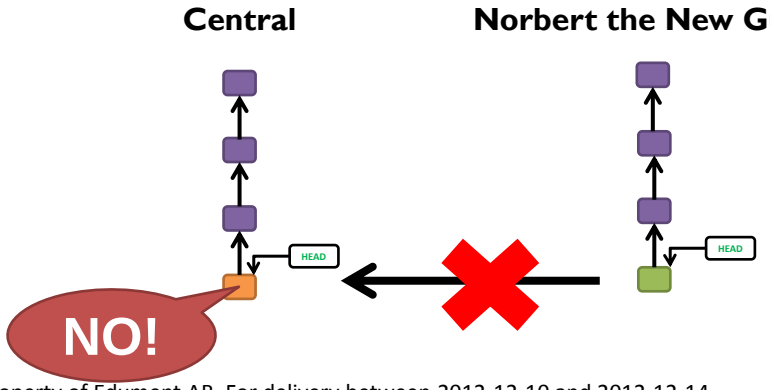
Norbert the New Guy



When pushing gets tricky (2)

If Norbert tries to push, it will fail

```
$ git push origin master
```



When pushing gets tricky (2)

The solution is to pull first

```
$ git pull
```

This will result in a merge commit being created locally; we then push it along with our commit

```
$ git push origin master
```

The underlying principle here is that all merging takes place locally

Exercise 3

In this exercise, you will work in small groups of two or three, practicing working with a remote repository hosted on GitHub

The exercise gets you to...

Add a remote and push
Clone that pushed repository
Make local commits, then push/pull them

You will also create conflicting changes for each other, and practice resolving them

Gerrit and code reviewing

Gerrit Rietveld

Dutch architect.

He designed these chairs.

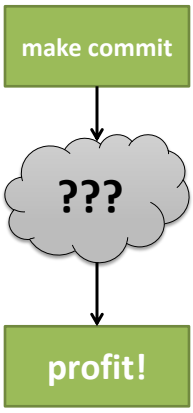
And this house.

His designs aim for simplicity and purity of ideas.

Not a bad role model for software projects. ☺

Gerrit the tool

Git is great at handling commits, but it doesn't provide any specialized tools to review commits.



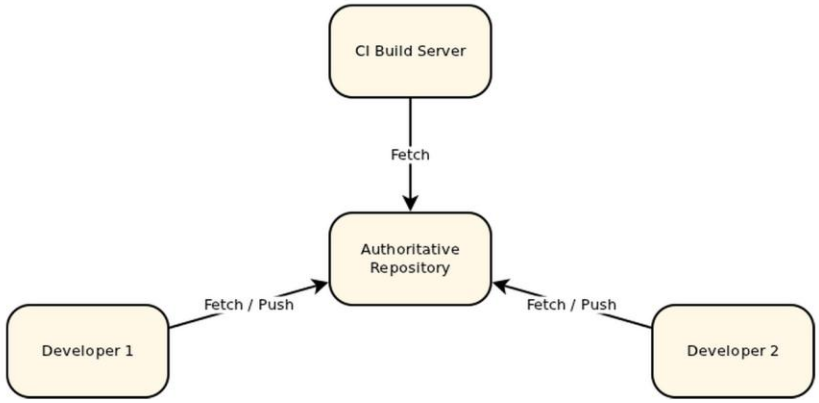
Before accepting a commit, we need to know three things:

1. Is it any good? (review)
2. Does it compile? (validation)
3. Does it apply cleanly? (merge)

Gerrit answers these questions in an organized way.

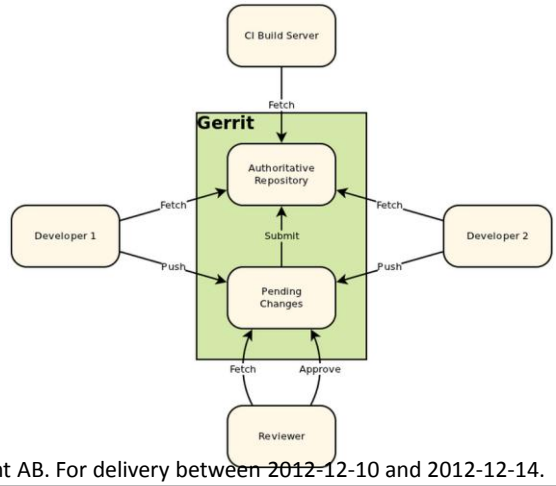
Where Gerrit fits in

The usual configuration of a central repository. Devs push to and fetch from the same location.



Where Gerrit fits in

With Gerrit, commits have to be approved before ending up in the repository.



The default Gerrit workflow

There are three roles involved in the life of a commit:

author (wrote the commit)
approver (does code review)
verifier (asserts correctness)

Some of these roles may be combined into one person. The verifier is often completely automated.

The approver

The approver for a change needs to determine the following:

- **Does this change fit within this project's stated purpose?**
- **Is this change valid within the project's existing architecture?**
- **Does this change introduce design flaws that will cause problems in the future?**
- **Does this change follow the best practices that have been established for this project?**
- **Is this change a good way to perform the described function?**
- **Does this change introduce any security or instability risks?**

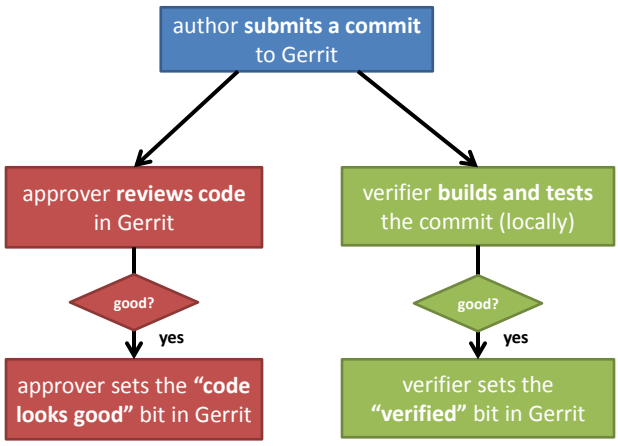
The verifier

The verifier for a change needs to determine the following:

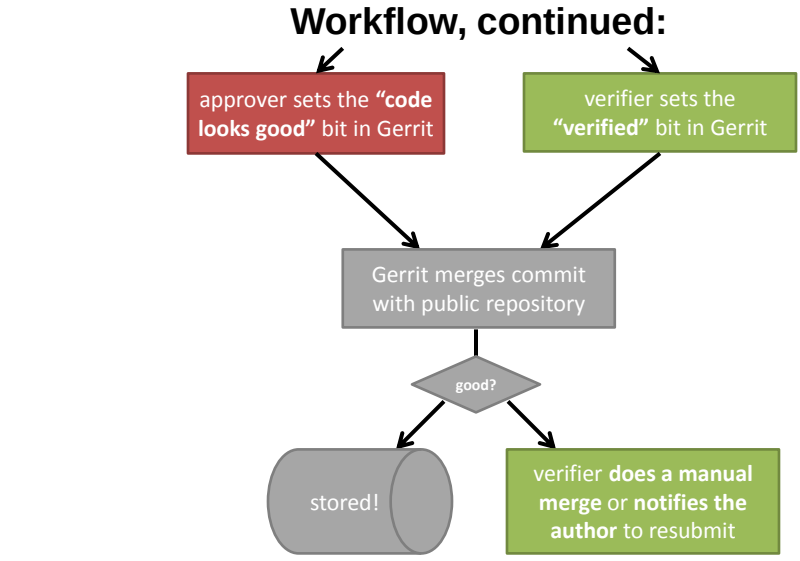
- **Patch the change into your local client using one of the Download commands.**
- **Build and test the change.**
- **Within Gerrit, use Publish Comments to mark the commit as "Verified" or "Fails", and add a message explaining what problems were identified.**

The default Gerrit workflow

We may illustrate the workflow like this:



The default Gerrit workflow



Levels

Approver and verifier may both set labels after review of the commit:

Code Review

+2 Looks good to me, approved

+1 Looks good to me, but someone else must approve

0 No score

-1 I would prefer that you didn't submit this

-2 Do not submit

Verified

+1 Verified

0 No score

-1 Fails

Command-line tools

Gerrit runs on a server. It can be used either through a web browser, or with commands through SSH.

`ls-groups`, `ls-projects`
`rename-group`
`set-reviewers` **(add or remove reviewers)**
`query` **(list change commits)**
`review` **(verify or approve)**
`stream-events` **(watch the action live)**

Change-Ids

It's not uncommon to have to commit several drafts of a change to Gerrit. The SHA-1 hashes will change every time, so we need some other way to identify the "same" change across commits:

```
Change-Id: Ic8aaa0728a43936cd4c6e1ed590e01ba8f0fbf5b
```

You're encouraged to install Gerrit's default `commit-msg` hook that automatically adds a Change-Id line during `git commit`.

Change-Ids

Gerrit will use the Change-Id line to

- **Create a new change**
  if it's the first time it sees the Change-Id
- **Update an existing change**
  if it's an old Change-Id but a new commit
- **Close a change**
  if it's an old Change-Id, and the commit is pushed directly to a branch

Signed-off-by

Gerrit has a concept of "sign-offs", lines that track who is responsible for a commit. You can provide a Signed-off-by: line, if

(a) you wrote the code, or
(b) you copied it from an appropriately licensed code base, or
(c) you received it from someone who certified (a), (b), or (c).

By signing off your code, you show that you understand and agree to the project's license and redistribution practices.

Other similar lines

There's also the Acked-by: line. Expressing that the person wasn't involved in the handling of the patch, but has approved it.

Similarly, a Cc: line indicates that someone might have an interest in the patch. (But the person might not have looked at it yet.)

The Reported-by:, Tested-by: and Reviewed-by: tags all similarly indicate other roles involved in the patch. The Reviewed-by: tag indicates not only review, but acceptance.

Access Controls

Gerrit has a fine-tuned set of access control settings.

Access privileges are granted to groups, not individual users.

The Verified and Code Review labels mentioned previously are provided as changeable defaults in Gerrit's access control system.

Gerrit, background

Google developed Mondrian, a code review tool. Not open source. Too tied to the use of Perforce and Google-only services.

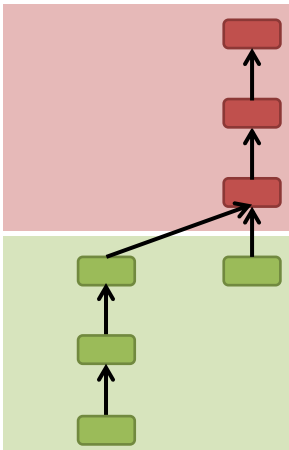
Guido van Rossum cloned parts of Mondrian into Rietveld, running on Google App Engine, targeting Subversion.

The Android Open Source Project uses Git. Gerrit Code Review started as patches to Rietveld, to service AOSP. It soon diverged due to different goals. Was renamed to Gerrit.

The Wonder of Rebasing

Re-writing history (1)

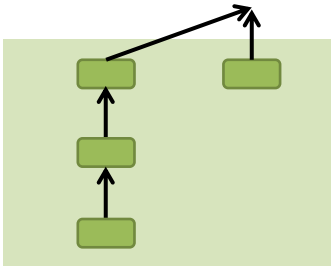
When working with Git and you have remotes, your version history falls into two categories



History that you share with others, e.g. that you have pushed to or pulled from a remote

History consisting of commits you have made locally, but have never pushed to a remote

Re-writing history (2)



Since nobody has seen your local commits, you are free to manipulate that part of the history without causing any problems

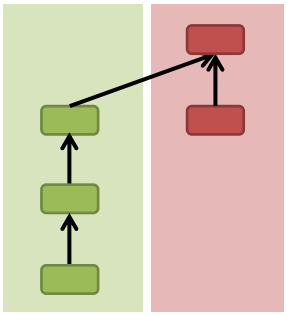
Very Important Warning

You must be very careful to never change history that you have already shared. The big hint you've got it wrong is that a `push` will fail, and Git will suggest using `--force`. DO NOT DO THIS; you will cause problems for others.

Rebasing (1)

You started a branch to work on a feature, and have made a few commits. Then you switched back to master to do a bug fix. You pushed the bug fix, and now return to your feature branch.

Your work on the feature branch is local - you did not push it to a remote



On the other hand, the bug fix is now pushed; it is shared history

Rebasing (2)

A couple of things are now less than ideal...

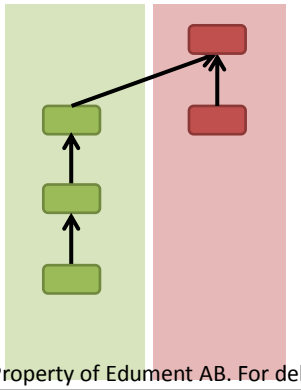
Your feature branch is missing the bug fix ☹
You'd really like to have the bug fix in your branch too; it may affect your feature, and you'd like to be able to test the two in combination

You won't be able to do a fast-forward merge ☹
This isn't a huge problem, but merge commits do create a little clutter in the version history; many Git users prefer fast-forward merges

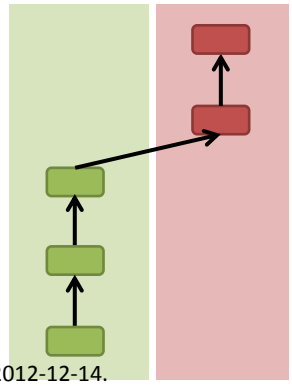
Rebasing (3)

The `rebase` command can help here. In the feature branch, we can run:

```
$ git rebase master
```



In a rebase, Git takes the commits in your branch, then "replays" them as if the branch had started at the master branch's current commit, so it has all master does and can fast-forward!



Pulling with rebasing

We've seen that merge commits can also appear when you have local commits, then pull commits from a remote repository

It is often desirable to avoid these merge commits; pulling is such a common scenario that this will quickly clutter the history

Thankfully, it's as simple as:

```
$ git pull --rebase
```

Amending commits

Ever made a commit...then instantly realized it had a mistake - perhaps even in the commit message?

Provided you did not push the commit, this is easy to fix locally; correct any files, and then use the `amend` flag:

```
$ git commit --amend -m "New message."
```

This will take the previous commit, incorporate the current changes into it, and use the new commit message you specify also

Exercise 4

This exercise focuses on re-writing local history

First, you will use the `--amend` option with a commit - it's too useful not to have to hand!

After this, you will look at the difference you get in the DAG when you use rebasing compared to created merge commits

You will be able to visualize the DAG in `gitk`

Other Useful Stuff

stash

Git provides a "stack" that you can push your current changes on to...

```
$ git stash
```

Giving you a clean repository again. Later, you can pop the changes from the stack:

```
$ git stash apply
```

Note you can switch branch in the meantime.

When might `stash` be useful?

cherry-pick

What if you have...

A master branch
A feature branch
A bug fix in the feature branch

You don't want to merge, you just want to take the single commit. Look up the commit ID, then:

```
$ git cherry-pick abc123
```

Done. ☺

reset

Another command with loads of different possibilities - check out the help page!

If you want to throw away all of your uncommitted changes (including staged ones):

```
$ git reset --hard
```

You can also use it to clear the staging area:

```
$ git reset
```

Tags

In Git, a tag is simply an alias for a commit

Created using the `tag` command, specifying the name for the tag to create along with the commit that you want the tag to reference:

```
$ git tag my-tag-name abc123
```

Unlike a branch name (also an alias to a commit), a tag can not be committed to

What might you use tags for?

Line and hunk level commits

So far, we have seen that we can move files into the staging area, then commit them

If you end up with two changes tangled in a single file, you can also just stage individual lines or hunks rather than the whole file

This is easiest done with another GUI tool:

```
$ git gui
```

<live demo>

Bonus Exercise (If Time)

Review some of the topics covered in this final section of the course.

Pick one of them, and explore it by yourself or, if you prefer, in a small group.

If you're unsure which to choose, then stashing is perhaps the most invaluable one to practice.

If you have time and are comfortable, you may wish to explore a second topic also.

Closing Remarks

A paradigm shift

Distributed version control is more a step of revolution rather than evolution

Branching becomes lightweight, natural and cheap - both in terms of branch creation and especially thanks to efficient merging

Working locally makes things fast

More complex workflows can be implemented since all copies of a repository are on an equal footing; you give them their place and value

Remember...

In the end, it's all about commits and the DAG

Branches are a central part of effective Git usage. It's better to create a branch you then merge right away, than to not create one and wish you had

Rebasing and amending commits are powerful techniques - but you must only apply them to local history that has not been shared

Look at your desired development process, and shape your Git usage around it - it's malleable!

Evaluation

Please complete the evaluation for this course

What did you like most?

Was anything not as clear as it could have been?

Was there anything that you think should have been done differently?

We take all feedback into account, and use it to help us improve and optimize the course

Thank you! ☺

Some other interesting courses offered by – or coming soon from - Informator + Edument:

Advanced Git
Git in practice
Software Architecture
Modern Business Applications with DDD and CQRS
C# Master Class
Modern web development with JavaScript
ASP.NET MVC 4 with TDD

You can contact us at

info@edument.se