GIT

John DeDourek
Professor Emeritus
Faculty of Computer Science
University of New Brunswick

What is Git?

- A source code control system
  - Implies program code
- A version control system
  - Implies any sort of files

And what is a version control system?

- A time machine
  - Maintains a history of project development in a repository
  - Allows one to go back in time and extract previous states of the project from the repository

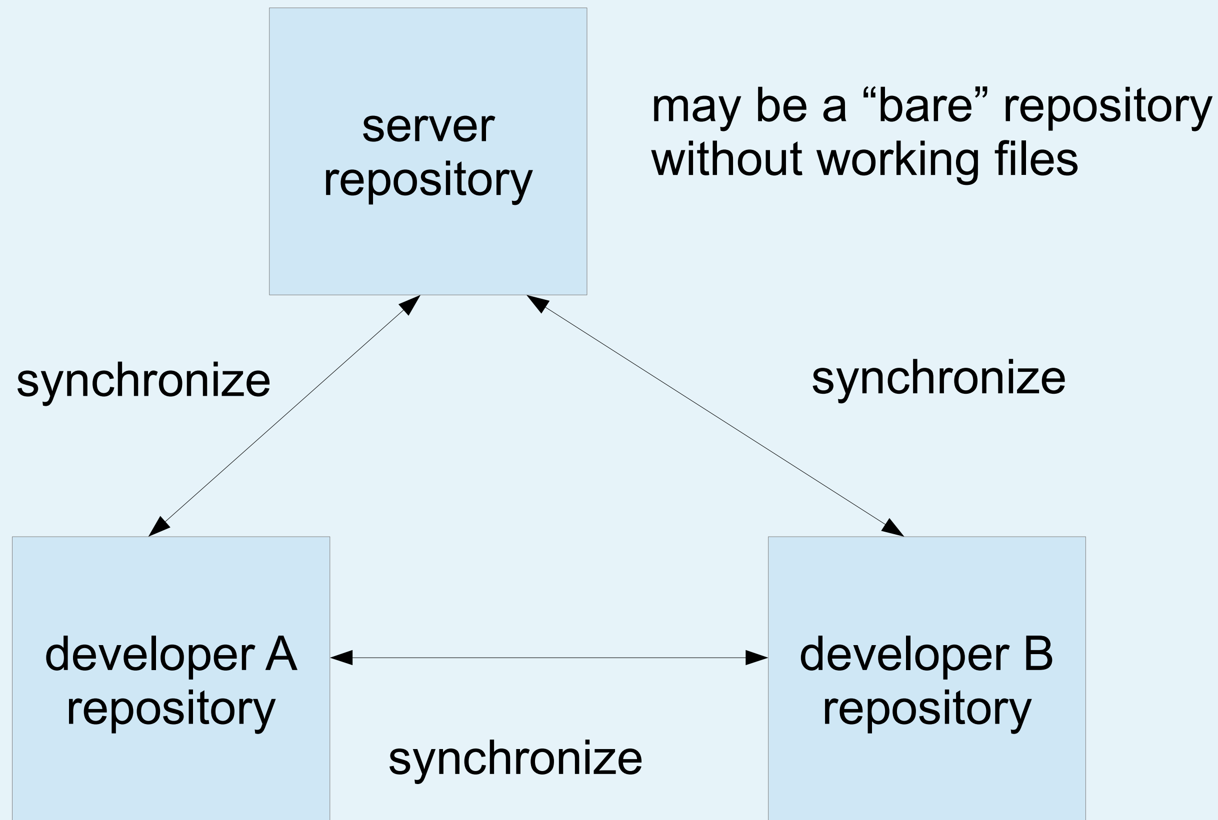
Why?

- Find where a regression error was introduced
- Support multiple versions of a project
- Safely integrate contributions to a project from others
- Can you think of other reasons?

Types of version control systems:

- Local
  - Local repository to track personal files
- Centralized
  - One repository for a project is shared by all developers
- Distributed
  - Each developer has a local repository periodically synchronized with others

Distributed



How to proceed

- Learn to use git as a local version control system
- Learn to synchronize repositories among computers

Git

- Does not solve all problems
- Makes some assumptions

Assumptions I

- Project is composed of files
- Files are under one root directory, optionally with subdirectories
- Files are of two types:
  - Originally created, eg program source code
  - Derived, eg compiled object code
  - These two types can be distinguished by file name pattern

Assumptions II

- Most original files are text files
- Development proceeds by repeating forever:
  - Repeat until satisfied
    - Create and edit files
    - Test the results
  - Commit the results to the history repository
- However, GIT is a set of tools and does not specify a workflow

GIT repository

- A hidden subdirectory under the project root directory (“.git” in Linux)
- Contains a content addressable memory of “objects” or “blobs”
- Also contains some miscellaneous other stuff

Content addressable memory

- In principle, the contents of an object are its name
- In practice, the name is a 20 byte (160 bit) SHA-1 hash
    - Shown as 40 hex digits

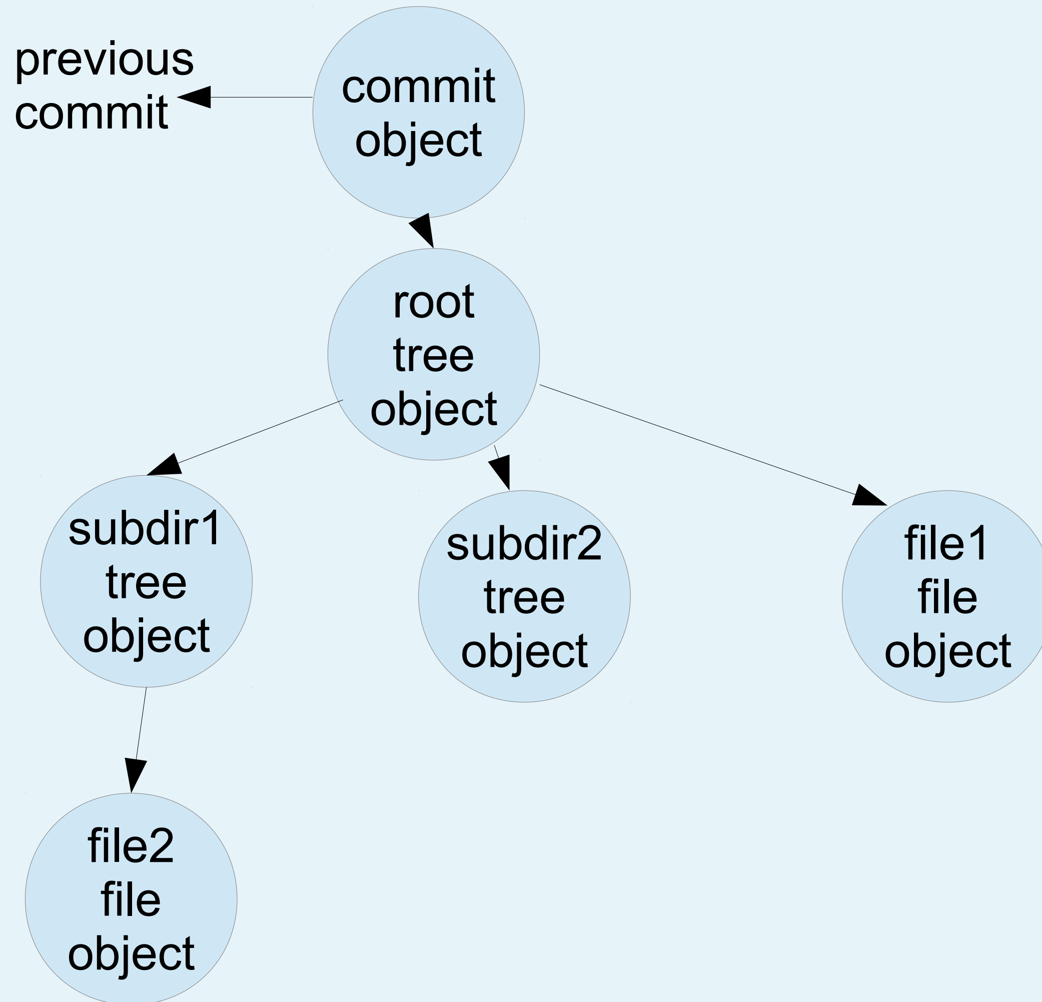
Objects I

- File
  - Header with some identifying stuff, eg your name for the file, followed by the contents of the file
- Tree
  - Corresponds to a directory (root or subdirectory)
  - Contains your name for the directory and then the hashes of the files and subdirectories contained in it

Objects II

- Commit
  - Date, committer name, description, hash of the root directory, hash of the previous commit, etc.

A commit



Finally, lets use it

- I will be demonstrating GIT on Linux using the command line, which is GIT's “native” interface
- Adapt this to whatever you use; the concepts are the same

Installation

- Linux: usually there; otherwise install for the distro repository
- Windows: try http://msysgit.github.io/
  - Note: msysgit was moved from Google Code to github
- MAC: temporarily try https://code.google.com/p/git-osx-installer/ but note that downloads have moved to sourceforge; not sure where the project will be moving

A bit about configuration

- There are four sources of configuration; each later one in the list over rides the previous one

Configuration sources

- Defaults built into git itself
- A system-wide configuration for the installation; modifiable only by the system administrator
- A global configuration for each user (in $HOME/.gitconfig on Linux)
- A local configuration for each repository, contained in the repository itself

Suggestion:

- After installation, set the “global” configuration to the one you use most often
- For any projects for which your standard global configuration is unsuitable, set the “local” configuration to over ride
  - Set “local” configuration after initializing the project's GIT repository and while in the projects directory

Some configuration examples

```
git config --global user.name "jane smith"
git config --global user.email js@example.com
git config --global core.editor vi
git config --list
git config core.editor
```

Now we are set up to create projects; Example I

```
mkdir exampleproject
cd exampleproject

git init

echo "line one" > filea.txt
echo "line two" >> filea.txt
echo "line three" >> filea.txt
echo "hello world" > fileb.txt

git status
```

What happened?
We have “nothing to commit”!
We have “2 untracked files”

A bit about the cache (aka index)

- A file to be committed must be added to the index
  - this makes it a tracked file
  - this places a copy of the file in the index
- A commit operation always takes the files from index to create the committed state
  - even if the index copy is different from the working copy

States of a file I

- Ignored: the file exists in the working directory but GIT ignores the file and does not track or report changes
- Untracked: the file exists in the working directory and GIT does report the presence of the file, but does not track changes

States of a file II

- Modified and not staged for commit; copies of the file exist in the working directory, in the index, and in the most recent commit; thus it must have been added to the index previously and is therefore being tracked; the copy in the index and the copy in the most recent commit are identical; however the copy in the working directory is different

States of a file III

- Modified and staged; again, copies of the file exist in the working directory, in the index, and in the most recent commit; the copies in the working directory and in the index are identical; however, these differ from the copy in the most recent commit; a copy of the file in the index will be part of the next commit

States of a file IV

- both an unstaged and a staged modified copy of the file exist; again copies exist in the working directory, in the index, and in the most recent commit; however, all three copies are different; the copy in the index will become part of the next commit

Example II

```
git add filea
git add fileb

git status

git commit -m "The first commit"

git status
```

Example III

```
echo "And everyone in it" >> fileb

git status

git add fileb
git commit -m "second commit"

git status
git log
```

Ignoring files

```
echo "*~" >> .gitignore

git status
git add .gitignore
git commit -m "ignore editor backup files"

git status
git log
```

Git branching

- A branch is simply a pointer to a commit
- One branch is normally the "current branch"
- After “git init” we have one branch named “master” and it is the current branch
- A commit “advances” the current branch

Branch example I

```
git branch treble
git checkout treble
```

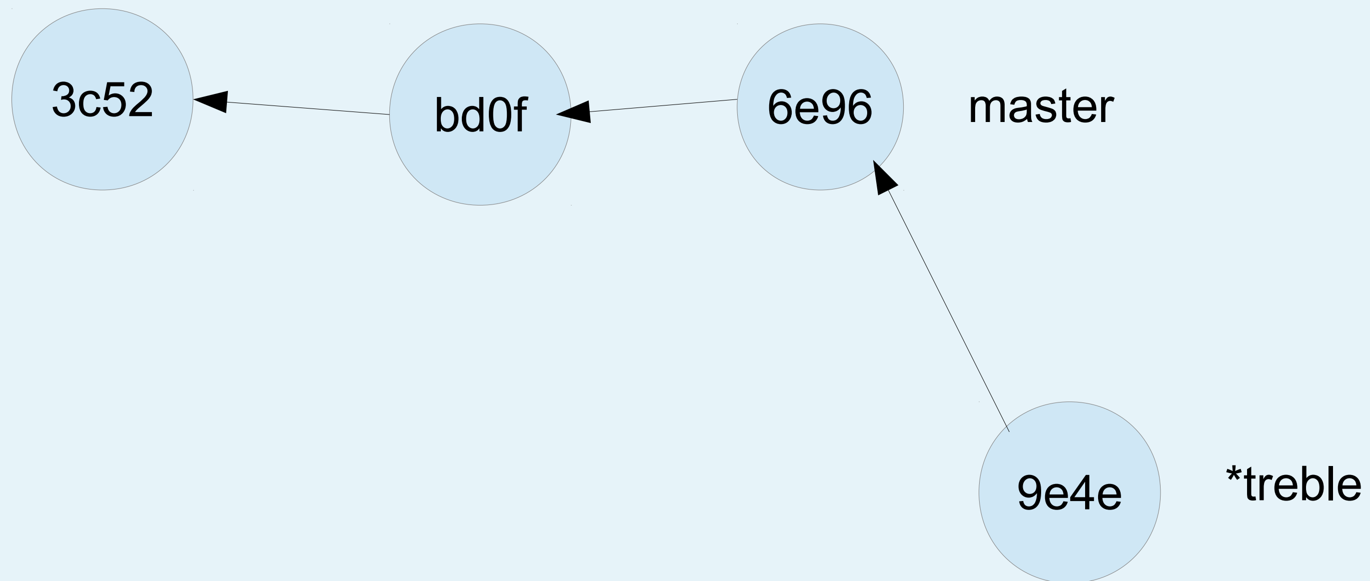
or

```
git checkout -b treble
```

then

```
git status
echo "Every Good Boy Does Fine" > filetreble
git add filetreble
git commit -m "Treble"
git status
```

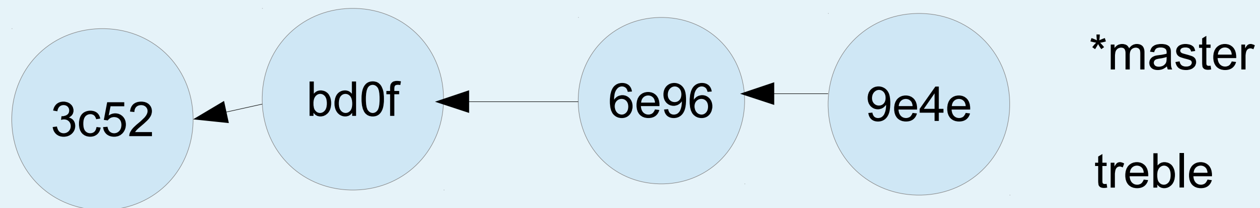
Branch example II



Branch example III

```
git checkout master
git merge treble
```

Branch example IV



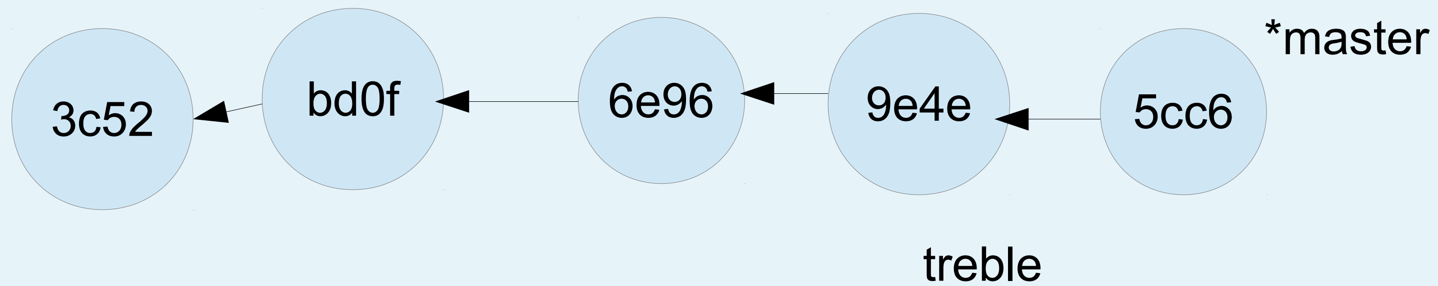
master just moved ahead from 6e96 to 9e4e

this is a fast forward merge

Branch example V

```
echo "Good Boys Do Fine Always" > filebass
echo "All Cows Eat Grass" >> filebass
git add filebass
git commit -m "Bass"
```

Branch example VI

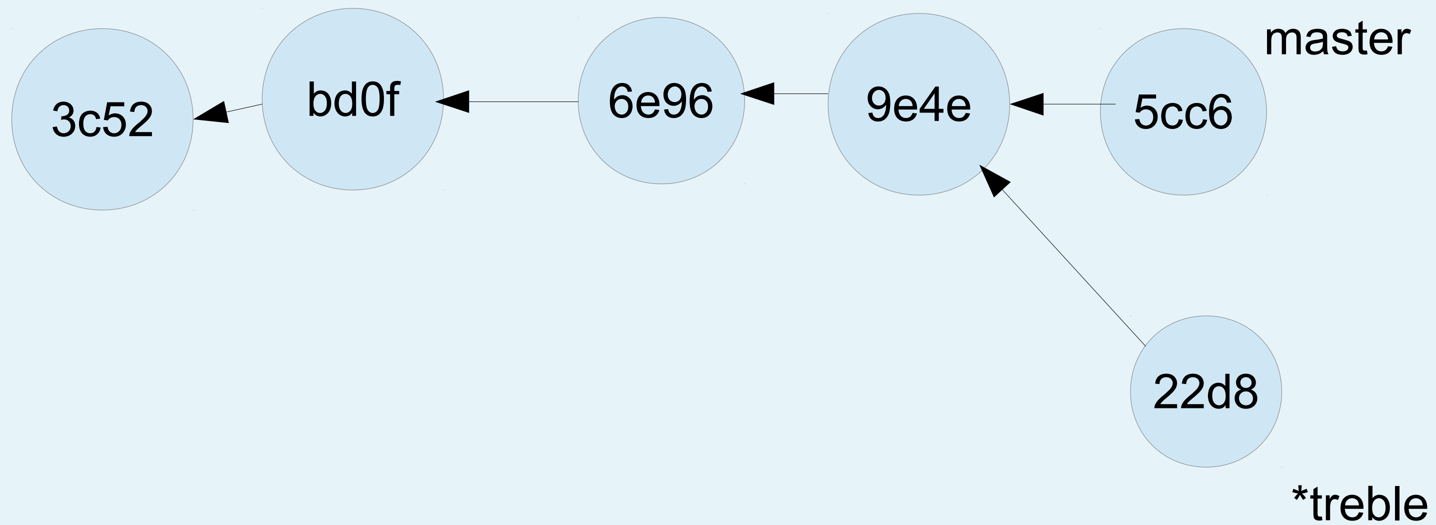


Branch example VII

```
git checkout treble
echo "F A C E" >> filetreble
git commit -a -m "Extend file treble"

git status
git log
```

Branch example VIII



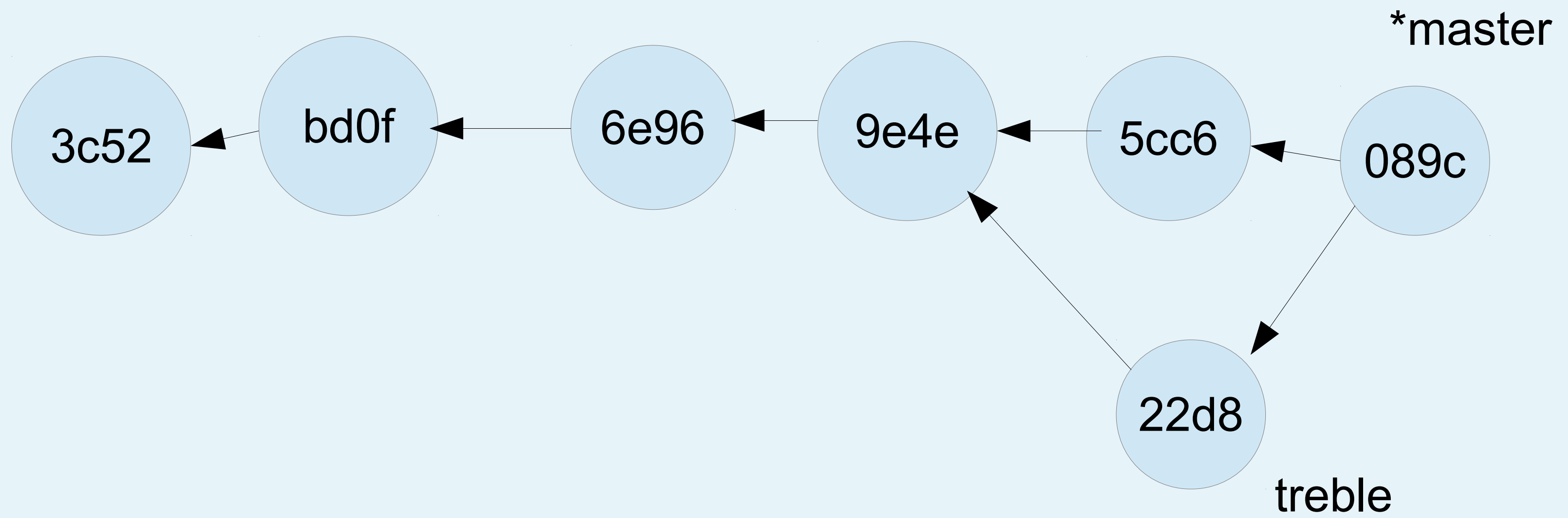
Branch example IX

```
git checkout master
git merge treble
```

a real merge, not fast forward
there are no conflicts in this case

```
git status
git log --graph
```

Branch example X



Conflict example I

```
git checkout master
echo "I add line" >> filetreble

git checkout treble
echo "You added line" >> filetreble

git checkout master
git merge treble
```

Oh! Oh! Trouble!!!

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

Conflict example II

```
git status

# On branch master
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to
#                                 mark resolution)
#
#  both modified:      filetreble
#
no changes added to commit (use "git add" and/or
                            "git commit -a")
```

Conflict example III

```
cat filetreble

Every Good Boy Does Fine
F A C E
<<<<<<< HEAD
I add line
=======
You added line
>>>>>>> treble
```

I changed this to

```
Every Good Boy Does Fine
F A C E
We both added lines
```

and then

```
git commit -a
git status
git log --graph
```

Next episode...

- Synchronizing repositories on multiple machines