Linux For Embedded Systems

For Arabs

Cairo University
Computer Eng. Dept.
CMP445-Embedded Systems

Ahmed ElArabawy

Lecture 11:
Introduction to Git & GitHub (Part 2)

Git Object Model

Object Identifier

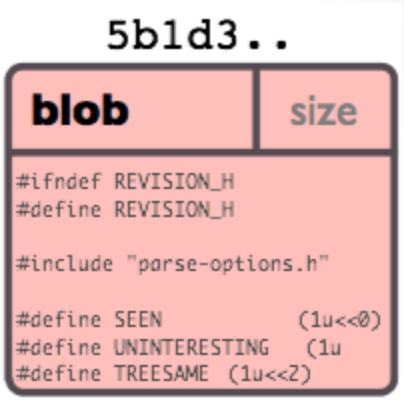
- Git operates by manipulating different types of objects
- Any object has an id, which is an **SHA1 hash** of its contents
- Outcome is a 40 characters hash
- Most of the time, we only use the first few characters that distinguish the object (at least 5 characters)
- This guarantees,
  - Never have two different files with same id
  - Any identical files, on two different machines/repos will have the same object-id
  - Can compare files/folders easily by comparing id's (no need to compare the contents)

Git Objects

- Each object will have
  - Type
  - Size
  - Content
- Object types are:
  - **Blob**: represents any file or any content
  - **Tree**: represents any subdirectory
  - **Commit**: represents a snapshot in time, of the tree upon a commit
  - **Tag**: represents a special milestone in the tree; normally marks a special commit
- To show the content of any object via its Object id

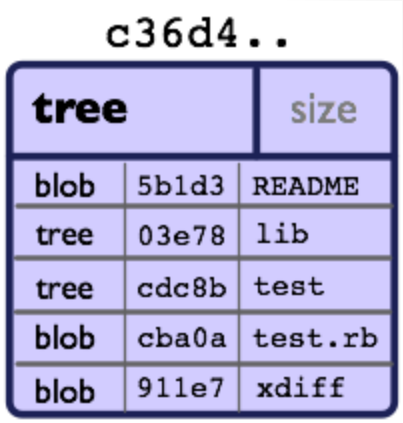
  $ git show <object id>

Blob Object



- Object Attributes:
  - **Type**: Blob
  - **Size**: Size of file
  - **Contents**: the file contents
- Note that the blob object does not refer to:
  - File name (renaming the file does not change the blob object id)
  - Location (moving the file does not change the blob object id)

Tree Object



- Object Attributes:
  - **Type**: tree
  - **Size**: size of object
  - **Content**: a table for the objects within the tree
    - Mode of object (read/write/execute)
    - Type of object (blob for a file, tree for a subdirectory)
    - Obj Id for the object
    - File name of file/subdirectory
- So when we rename a file or move it, the file object (blob) does not change, but the tree object changes
- Note that if two tree objects ids are identical, then this means that they have identical file structure and file contents. This simplifies folder comparison significantly
- Tree objects can be shown via

  $ git show <obj id>

  $git ls-tree <obj id> (gives more details)

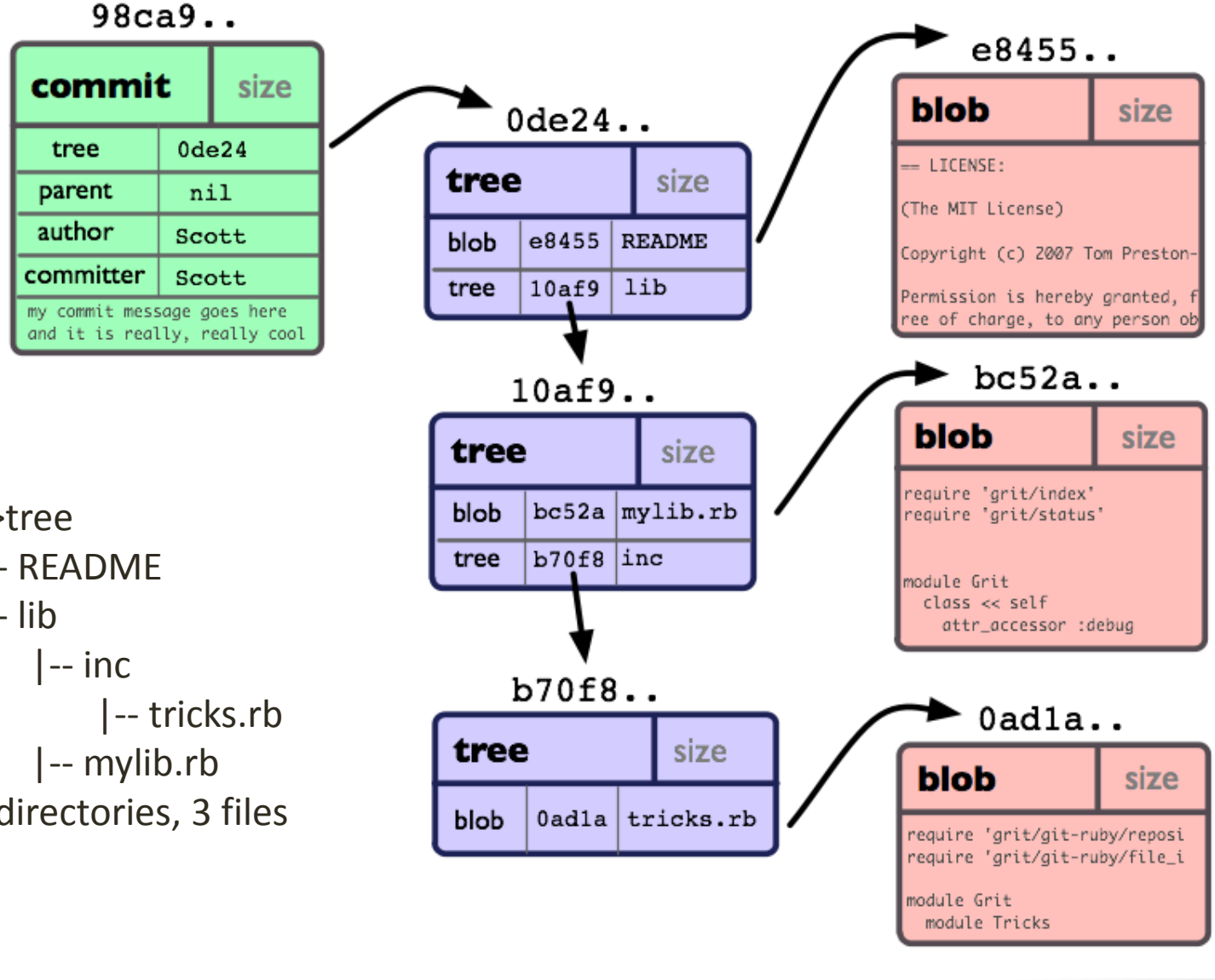
Commit Object

- Object Attributes:
  - **Type**: commit
  - **Size**: size of object
  - **Content**: the following fields
    - Tree object Id (for the tree to be committed)
    - Parent (s) commit object Id (parent commit or commits in case of a merge)
    - Author Id
    - Committer Id (different if the committer is committing a patch sent by the author)
    - Commit message
- A commit with no parent, is called the root commit which is the initial revision of the project
- Commit objects can be shown via,

  $ git show <obj id>

  $ git show -s --pretty=raw <obj id>

Example:




```
$>tree
|-- README
|-- lib
      |-- inc
            |-- tricks.rb
      |-- mylib.rb
2 directories, 3 files
```

Tag Object

- Object Attribute:
  - **Type**: tag
  - **Size**: size of object
  - **Content**: the following fields,
    - Tagged object id
    - Tagged object type
    - Tag name
    - Tagger id
    - Tag message (may contain a signature)
- Note that there are "***lightweight tags***" which are not tag objects, they are just simple references

Starting a Repo

1. Initializing a new Repo

- To start a git repo from scratch do the following:
  - Create all the files of the project
  - Go to the root of the project where you want your repo to be created
  - Do the command

    $ git init
  - Now the repo is ready, with no tracked files yet
  - Files to be tracked will need to be staged, then committed (to be discussed later)

2. Copying an existing repo

- To copy a remote repo from another machine (whether server or another developer)

  $ git clone <the url for the remote repo>
- Examples:

  $ git clone git://cworth.org/git/hello.git

  $ git clone http://cworth.org/git/hello.git

  $git clone ssh://cworth.org/git/hello.git
- The result is a folder named hello containing the **.git** repo
- If we wanted to name it differently,

  $ git clone git://cworth.org/git/hello.git <newFoldername>
- Cloning a remote repo,
  - Copies the repo (.git directory) to the local machine
  - Checks out the latest files from the repo to the working directory

Query of the Origin

- Git calls the remote repo which was copied (the url we used in the clone command), the **origin**
- Git stores the origin of the repo into its ***.git/config*** file
- To get the origin read the config file, or,

  $ git config --get remote.origin.url
- Note,
  - The origin will be used as the default remote (that we push to or pull from) in several commands that address remote repo's

Browsing Git Repo History

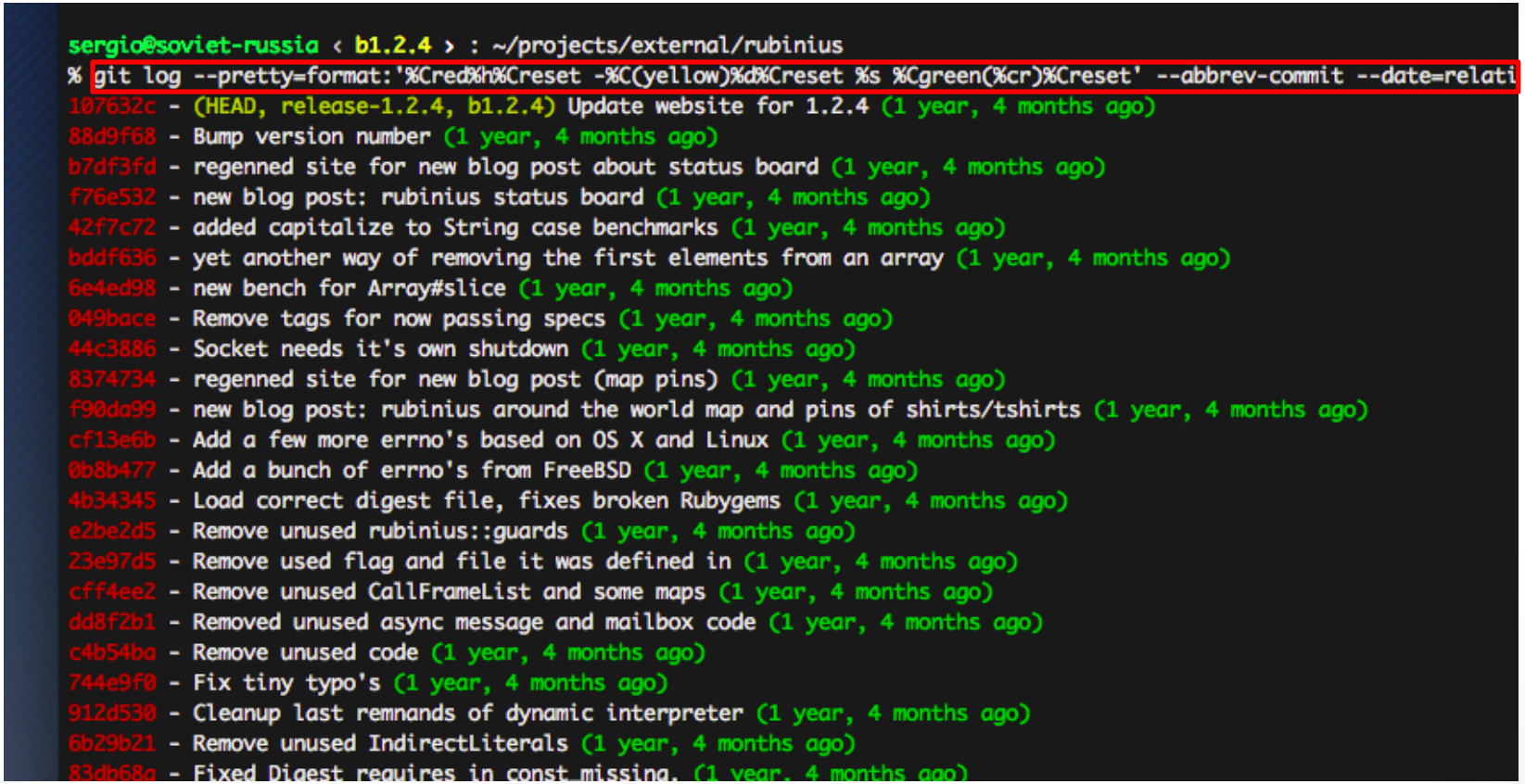
View the Repo History (git log Command)

- To view the history of a repo, use,

  $ git log

- This shows the history of the project with a full list of the commits
- The log can be manipulated as follows,
  - The format can be changed to other preset formats, or customized by the user
  - The order of commits can be changed
  - Commits can be filtered to show only a subset based on a count, date, affected file, a search string, …

$ git log


sergio@soviet-russia ‹ b1.2.4 › : ~/projects/external/rubinius
% git log --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr)%Creset' --abbrev-commit --date=relati
107632c - (HEAD, release-1.2.4, b1.2.4) Update website for 1.2.4 (1 year, 4 months ago)
88d9f68 - Bump version number (1 year, 4 months ago)
b7df3fd - regenned site for new blog post about status board (1 year, 4 months ago)
f76e532 - new blog post: rubinius status board (1 year, 4 months ago)
42f7c72 - added capitalize to String case benchmarks (1 year, 4 months ago)
bddf636 - yet another way of removing the first elements from an array (1 year, 4 months ago)
6e4ed98 - new bench for Array#slice (1 year, 4 months ago)
049bace - Remove tags for now passing specs (1 year, 4 months ago)
44c3886 - Socket needs it's own shutdown (1 year, 4 months ago)
8374734 - regenned site for new blog post (map pins) (1 year, 4 months ago)
f90da99 - new blog post: rubinius around the world map and pins of shirts/tshirts (1 year, 4 months ago)
cf13e6b - Add a few more errno's based on OS X and Linux (1 year, 4 months ago)
0b8b477 - Add a bunch of errno's from FreeBSD (1 year, 4 months ago)
4b34345 - Load correct digest file, fixes broken Rubygems (1 year, 4 months ago)
e2be2d5 - Remove unused rubinius::guards (1 year, 4 months ago)
23e97d5 - Remove used flag and file it was defined in (1 year, 4 months ago)
cff4ee2 - Remove unused CallFrameList and some maps (1 year, 4 months ago)
dd8f2b1 - Removed unused async message and mailbox code (1 year, 4 months ago)
c4b54ba - Remove unused code (1 year, 4 months ago)
744e9f0 - Fix tiny typo's (1 year, 4 months ago)
912d530 - Cleanup last remnands of dynamic interpreter (1 year, 4 months ago)
6b29b21 - Remove unused IndirectLiterals (1 year, 4 months ago)
83db68a - Fixed Digest requires in const_missing. (1 year, 4 months ago)

Changing the format of the log

- Formatting the log

  $ git log --pretty=short (For a short list)

  $ git log --pretty=full (default)

  $ git log --pretty=fuller (for more detailed)

  $ git log --pretty=oneline (each commit in one line)

- Custom format logs

  $ git log --pretty=format: '%h was %an , %ar , message: %s'

“git log --pretty=format” Options

Option	Description of Output
%H	Commit hash
%h	Abbreviated commit hash
%T	Tree hash
%t	Abbreviated tree hash
%P	Parent hashes
%p	Abbreviated parent hashes
%an	Author name
%ae	Author e-mail
%ad	Author date (format respects the date= option)
%ar	Author date, relative
%cn	Committer name
%ce	Committer email
%cd	Committer date
%cr	Committer date, relative
%s	Subject

Limiting the Range of Commits

- Tip of current branch (most recent commit) is called "***HEAD***"
- Parent commit is defined by the "***~***", so,
  - "***HEAD~***" means the one before last commit
  - "***HEAD~3***" means, 3 commits before last
- Use of "***..***" to specify range
- For example,

  $ git log HEAD~3.. (log from 3 commits back to tip)

  $ git log <commit id>.. (the range is exclusive)

More Advanced Logging Showing more Info

- To show the statistics for each commit (affected files, number of additions, number of deletions)

  $ git log --stat

- To show full diffs (patch) for the commits

  $ git log -p

- Note that will show a lot of info, so normally, it is done on a single commit using for example,

  $ git log -p - n 1 <commit id >

  $ git show <commit id>

Changing Order

- Topological order, very useful to see commits ordered with respect to their topology of merge, specially if merged with one line

  $ git log --oneline --topo-order --graph

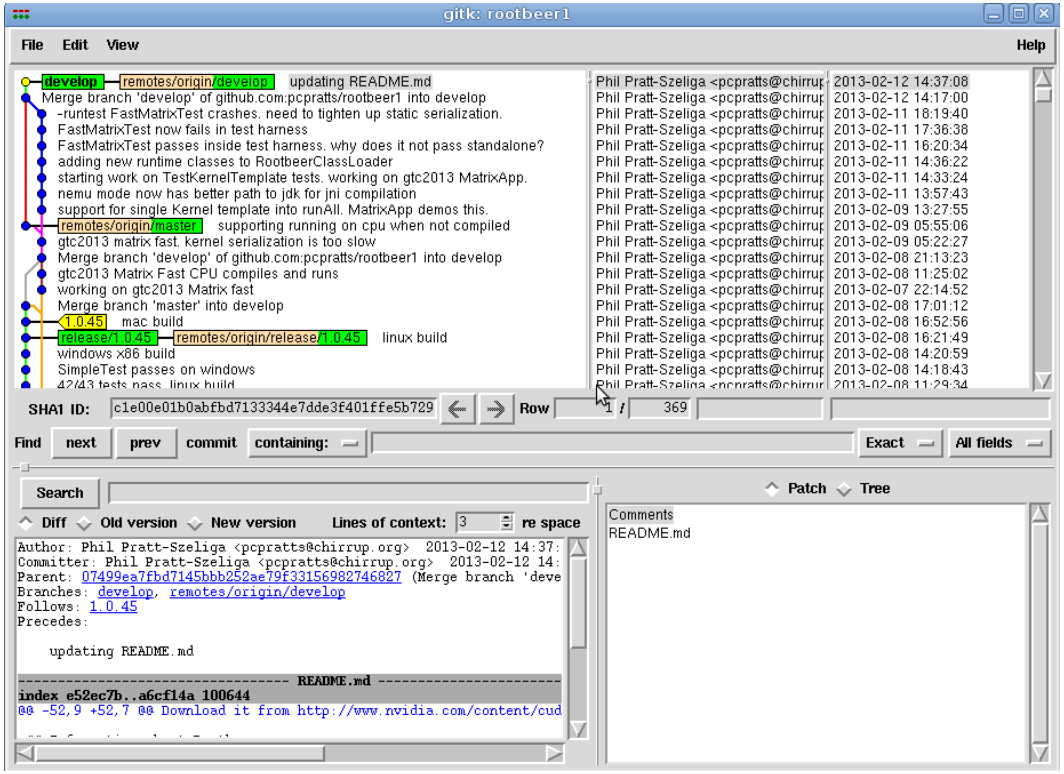
- Date order

  $ git log --oneline --date-order --graph

- reversing order

  $ git log --reverse

Gitk



- ***gitk*** is a graphical tool (tcl/Tk) to show the history of a repo
- It needs to be installed separately

  $ sudo apt-get install gitk
- A simpler (text based) graph can be obtained via

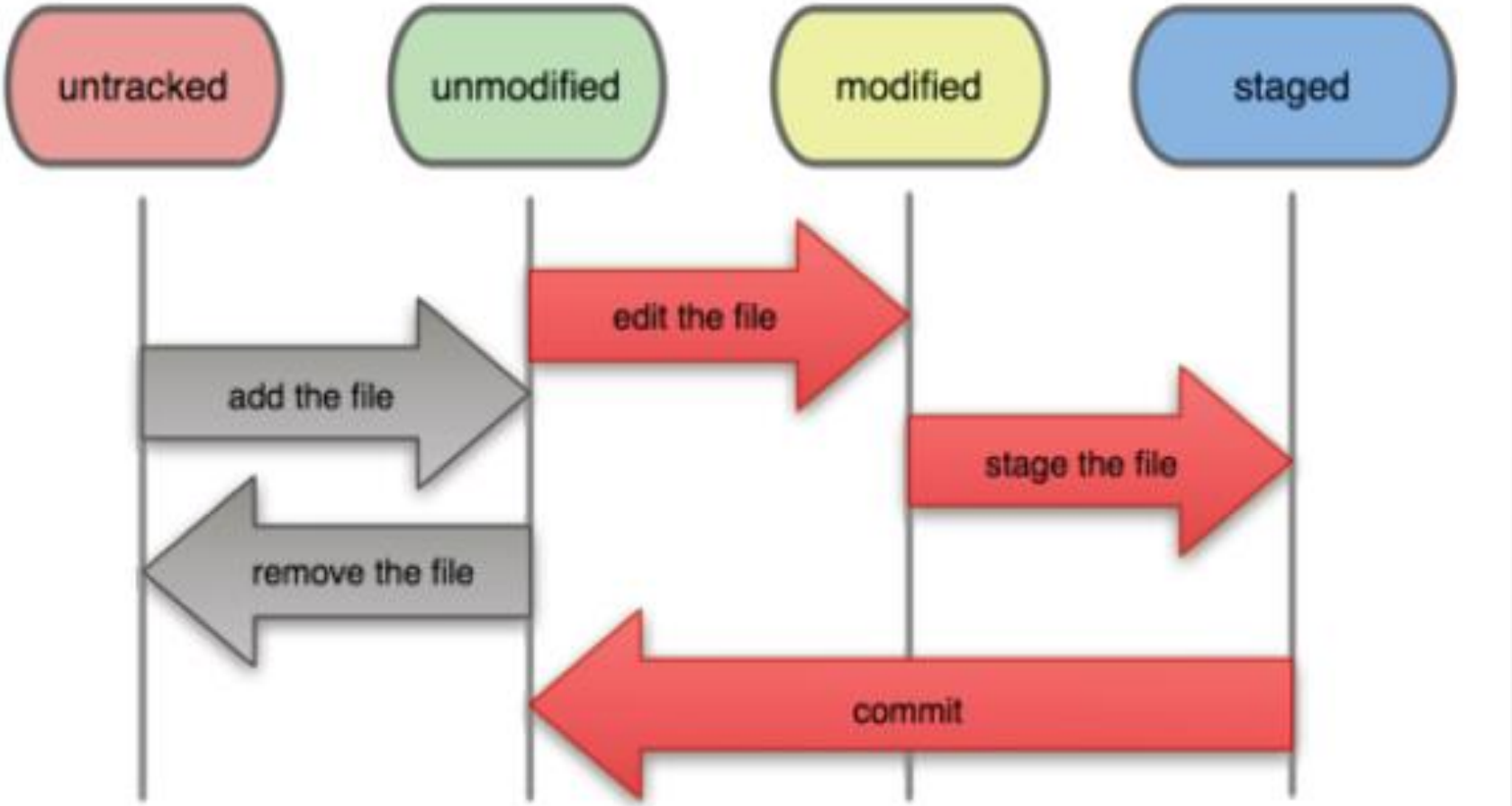
  $ git log --graph

Making Changes

Summary of Operations

- Now if we modify a file in the working directory and save it
  - To show which files have been updated
    $ git status
  - To show the changes we did to the files
    $ git diff
- To add the updated files to the index
  $ git add <file1> <file2> <file3>
- To unstage a file (remove it from the index)
  $ git reset HEAD (empty the index)
  $ git reset HEAD <filename> (remove this file from the index)
- To commit files to the repo
  $ git commit (commit files in the index to the repo)
  $ git commit -a (commit files directly from the working directory)
- To revert a file to the committed version
  $ git checkout --<filename>

File States


untracked
unmodified
modified
staged
edit the file
add the file
stage the file
remove the file
commit

Checking the Current Status (git status Command)

- The current status of the files show,
  - The current branch (Where the HEAD points)
  - The modified & unstaged files that needs to be added to the index
  - The modified & staged files that need to be committed
  - The Untracked files, that needs to be tracked with git (by staging and/or committing them
- Note that the same file can be in both the staged and modified unstaged status, since staging is done on content level and not on file level
  - i.e. if a file is modified, then added to the index, then modified again, then, we will have one staged version of the file and one unstaged
  - Solution, is either to commit the file, then stage and commit OR stage and commit the final version in one step

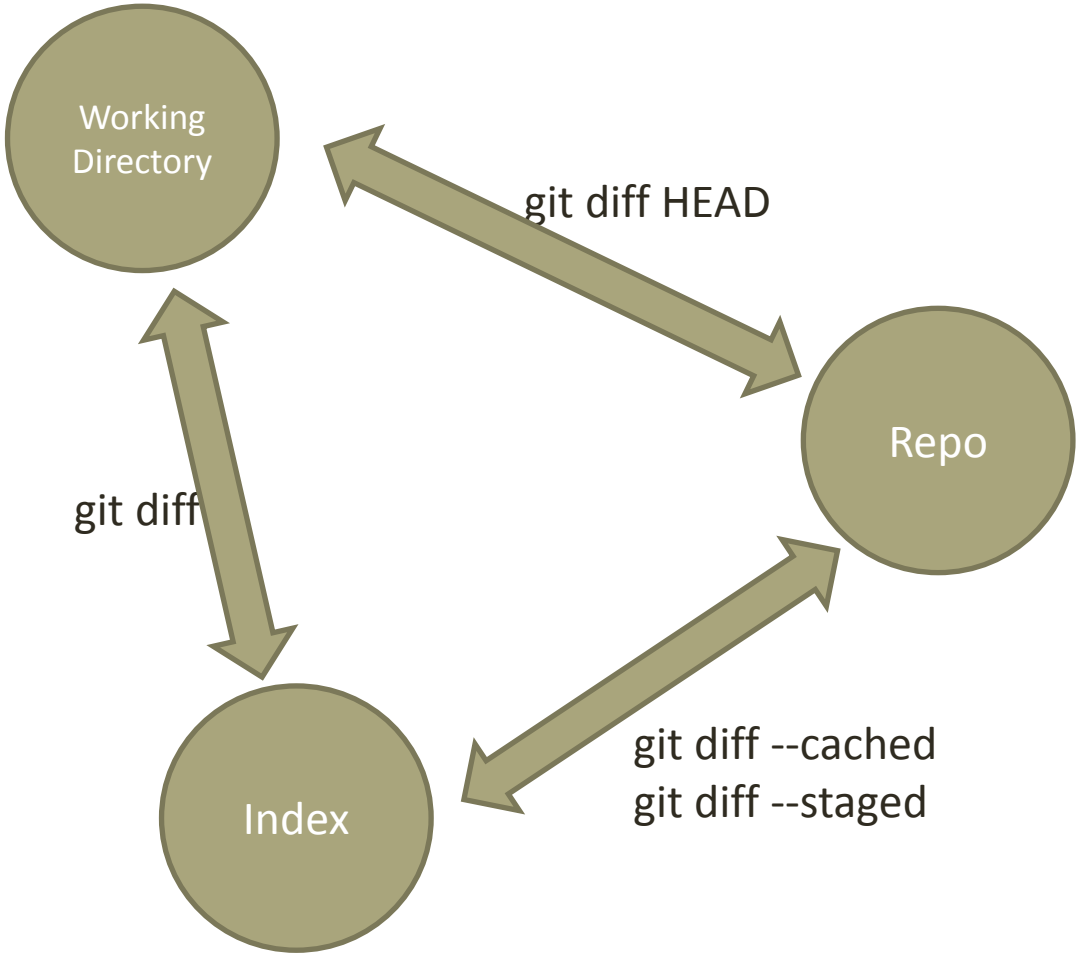
Ignoring files

- Some project files does not need to be tracked, such as
  - Data files generated by running the app
  - Log files
  - Object and library files generated by the build
  - Any other files
- They will keep showing up in the "***git status***"
- Accordingly, there is a need to configure git to ignore these files. This is done via creating the file ***".gitignore"*** in the working directory of the project
- If the settings should apply only to a subdirectory within the project, then place ***.gitignore*** in this directory.
- This means, we can have multiple ***.gitignore*** files in different places in the project
- Normally it is beneficial to track the ***.gitignore*** files through git as for other files, since it can grow with time and it is useful to maintain them

Example of .gitignore

```
# Lines starting with '#' are considered comments.
# Ignore any file named foo.txt
foo.txt
# Ignore (generated) html files,
*.html
# except foo.html which is maintained by hand.
!foo.html
# Ignore objects and archives
*.[oa]
#ignore temp files (ending with a ~)
*~
#ignore everything under the tmp directory
tmp/
```

Checking the current Changes (git diff Command)



Committing Changes

- To commit only files that are added to the index

  $ git commit (git opens the editor to enter the commit message)

  $ git commit -m "this is my commit message" (pass commit message in the command)

- To commit changes to the repo directly from the working directory

  $ git commit -a (does not include new or to-be-ignored files)

- The commit message should be
  1. 1 short line for abstract (shows in $git log --pretty short)
  2. A blank line
  3. A detailed description

Untracking a File (git rm Command)

- If a file is tracked, and we want to remove it from both the working directory and the repo

  $ git rm <file name>

  $ git commit -m "removing the file from the repo"

- If the file is already staged, then removal has to be forced

  $ git rm -f <filename>

  $ git commit

- To remove the file from the repo, but keep it in the working directory (as untracked file)

  $ git rm --cached <filename>

  $ git commit

- Example to remove all log files from the repo

  $ git rm --cached log/*.log (note the '\' before the * for git expansion)

  $ git commit

Moving a file (git mv Command)

- When renaming or moving a file,

  $ git mv <file-from> <file-to>

  $ git commit

Linux 4 Embedded Systems

http://Linux4EmbeddedSystems.com