1. Preparation

Goals

- To be fully prepared to work with Git.

01 Setting up name and e-mail address

If you've never used git before, first you need to set up your name and e-mail. Run the following commands to let git know your name and e-mail address. If git is already installed, skip down to the end of the line.

RUN:

```
git config --global user.name "Your Name"

git config --global user.email "your_email@whatever.com"
```

02 Installation Options line endings

Also, for users of Unix/Mac:

RUN:

```
git config --global core.autocrlf input

git config --global core.safecrlf warn
```

For Windows users:

RUN:

```
git config --global core.autocrlf true

git config --global core.safecrlf warn
```

3. Creating a Project

Goals

- To learn how to create a git repository from scratch.

01 Create a “Hello, World!” page
Get started in an empty working directory (for example, work, if you downloaded the file from the previous step) and create an empty directory named “hello”, then create a `hello.html` file in it with the following contents.

RUN:

```
mkdir hello

cd hello

touch hello.html
```

FILE: *HELLO.HTML*

```
Hello, World!
```

02 Create a repository
So you have a directory that contains one file. Run `git init` in order to create a git repo from that directory.

RUN:

```
git init
```

RESULT:

```
$ git init

Initialized empty Git repository in
/Users/alex/Documents/Presentations/githowto/auto/hello/.git/
```

03 Add the page to the repository
Now let’s add the “Hello, World” page to the repository.

RUN:

```
git add hello.html

git commit -m "First Commit"
```

RESULT:

```
$ git add hello.html

$ git commit -m "First Commit"

[master (root-commit) 911e8c9] First Commit

 1 files changed, 1 insertions(+), 0 deletions(-)

 create mode 100644 hello.html
```

4. Checking the status of the repository

Goals

- To learn how to check the repository's status

01Check the status of the repository

Use the `git status` command, to check the current state of the repository.

RUN:

```
git status
```

You will see

RESULT:

```
$ git status

# On branch master

nothing to commit (working directory clean)
```

The command checks the status and reports that there's nothing to commit, meaning the repository stores the current state of the working directory, and there are no changes to record.

We will use the `git status` command to keep monitoring the states of both the working directory and the repository.

5. Making changes

Goals

- To learn to monitor the working directory's state

01Changing the "Hello, World" page

Let's add some HTML-tags to our greeting. Change the file contents to:

FILE: *HELLO.HTML*

```
<h1>Hello, World!</h1>
```

02Checking the status

Check the working directory's status.

RUN:

```
git status
```

You will see …

RESULT:

```
$ git status

# On branch master

# Changes not staged for commit:

#   (use "git add <file>..." to update what will be committed)

#   (use "git checkout -- <file>..." to discard changes in working directory)

#

#   modified:   hello.html

#

no changes added to commit (use "git add" and/or "git commit -a")
```

The first important aspect here is that git knows `hello.html` file has been changed, but these changes are not yet committed to the repository.
Another aspect is that the status message hints about what to do next. If you want to add these changes to the repository, use `git add`. To undo the changes use `git checkout`.

6. Staging the changes

Goals

- To learn to stage changes for the upcoming commits

01Adding changes

Now command git to stage changes. Check the status

RUN:

```
git add hello.html

git status
```

You will see …

RESULT:

```
$ git add hello.html

$ git status

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#   modified:   hello.html

#
```

Changes to the hello.html have been staged. This means that git knows about the change, but it is not permanent in the repository. The next commit will include the changes staged.

Should you decide not to commit the change, the status command will remind you that you can use the `git reset` command to unstage these changes.

7. Staging and committing

A staging step in git allows you to continue making changes to the working directory, and when you decide you wanna interact with version control, it allows you to record changes in small commits.

Suppose you have edited three files (`a.html`, `b.html`, and `c.html`). After that you need to commit all the changes so that the changes to `a.html` and `b.html` were a single commit, while the changes to `c.html` were not logically associated with the first two files and were done in a separate commit.
In theory you can do the following:

```
git add a.html

git add b.html

git commit -m "Changes for a and b"

git add c.html

git commit -m "Unrelated change to c"
```

Separating staging and committing, you get the chance to easily customize what goes into a commit.

8. Commiting the changes

Goals

- To learn to commit to the repository

01 Committing changes

Well, enough about staging. Let’s commit the staged changes to the repository.

When you previously used `git commit` for committing the first `hello.html` version to the repository, you included the `-m` flag that gives a comment on the command line. The commit command allows interactively editing comments for the commit. And now, let’s see how it works. If you omit the `-m` flag from the command line, git will pop you into the editor of your choice from the list (in order of priority):

- GIT_EDITOR environment variable
- core.editor configuration setting
- VISUAL environment variable
- EDITOR environment variable

I have the EDITOR variable set to `emacsclient` (available for Linux and Mac).
Let us commit now and check the status.

RUN:

```
git commit
```

You will see the following in your editor:

RESULT:

```
|

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#   modified:   hello.html

#
```

On the first line, enter the comment: “Added h1 tag”. Save the file and exit the editor (to do it in default editor, press ESC and then type `:wq` and hit Enter). You should see …

RESULT:

```
git commit
```

```
Waiting for Emacs...

[master 569aa96] Added h1 tag

 1 files changed, 1 insertions(+), 1 deletions(-)
```

"Waiting for Emacs…" is obtained from the `emacsclient` program sending the file to a running emacs program and waiting for it to be closed. The rest of the data is the standard commit messages.

02Checking the status

At the end let us check the status.

RUN:

```
git status
```

You will see …

RESULT:

```
$ git status

# On branch master

nothing to commit (working directory clean)
```

The working directory is clean, you can continue working.

9. Changes, not files

Goals

- Understanding that git works with the changes, not the files.

Most version control systems work with files. You add the file to source control and the system tracks changes from that moment on.

Git concentrates on the changes to a file, not the file itself. A `git add file` command does not tell git to add the file to the repository, but to note the current state of the file for it to be commited later.
We will try to investigate the difference in this lesson.

01 First Change: Adding default page tags
Change the “Hello, World” page so that it contained default tags `<html>` and `<body>`.

FILE: *HELLO.HTML*

```
<html>
  <body>
    <h1>Hello, World!</h1>
  </body>
</html>
```

02 Add this change
Now add this change to the git staging.

RUN:

```
git add hello.html
```

03 Second change: Add the HTML headers
Now add the HTML headers (`<head>` section) to the “Hello, World” page.

FILE: *HELLO.HTML*

```
<html>

  <head>
  </head>
  <body>
    <h1>Hello, World!</h1>
  </body>
</html>
```

04 Check the current status

RUN:

```
git status
```

You will see …

RESULT:

```
$ git status

# On branch master
# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#   modified:   hello.html

#

# Changes not staged for commit:

#   (use "git add <file>..." to update what will be committed)

#   (use "git checkout -- <file>..." to discard changes in working directory)

#

#   modified:   hello.html

#
```

Please note that `hello.html` is listed in the status twice. The first change (the addition of default tags) is staged and ready for a commit. The second change (adding HTML headers) is unstaged. If you were making a commit right now, headers would not have been saved to the repository. Let’s check.

05Commit

Commit the staged changes (default values), then check the status one more time.

RUN:

```
git commit -m "Added standard HTML page tags"

git status
```

You will see …

RESULT:

```
$ git commit -m "Added standard HTML page tags"

[master 8c32287] Added standard HTML page tags

 1 files changed, 3 insertions(+), 1 deletions(-)

$ git status

# On branch master

# Changes not staged for commit:

#   (use "git add <file>..." to update what will be committed)

#   (use "git checkout -- <file>..." to discard changes in working directory)
```

```
#

#   modified:   hello.html

#

no changes added to commit (use "git add" and/or "git commit -a")
```

The status command suggests that `hello.html` has unrecorded changes, but is no longer in the buffer zone.

06 Adding the second change

Add the second change to the staging area, after that run the `git status` command.

RUN:

```
git add .

git status
```

Note: The current directory ('.') will be our file to add. This is the most convenient way to add all the changes to the files of the current directory and its folders. But since it adds everything, it is a good idea to check the status prior to doing an `add .`, to make sure you don't add any file that should not be added.
I wanted you to see the "add ." trick, and we will continue adding explicit files later on just in case.

You will see …

RESULT:

```
$ git status

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#   modified:   hello.html

#
```

The second change has been staged and is ready for a commit.

07 Commit the second change

RUN:

```
git commit -m "Added HTML header"
```

10. History

Goals

- To learn to view the project’s history.

Getting a list of changes made is a function of the `git log` command.

RUN:

```
git log
```

You will see …

RESULT:

```
$ git log

commit fa3c1411aa09441695a9e645d4371e8d749da1dc

Author: Alexander Shvets <alex@githowto.com>

Date:   Wed Mar 9 10:27:54 2011 -0500



    Added HTML header



commit 8c3228730ed03116815a5cc682e8105e7d981928

Author: Alexander Shvets <alex@githowto.com>

Date:   Wed Mar 9 10:27:54 2011 -0500



    Added standard HTML page tags



commit 43628f779cb333dd30d78186499f93638107f70b

Author: Alexander Shvets <alex@githowto.com>

Date:   Wed Mar 9 10:27:54 2011 -0500



    Added h1 tag



commit 911e8c91caeab8d30ad16d56746cbd6eef72dc4c
```

```
Author: Alexander Shvets <alex@githowto.com>

Date:   Wed Mar 9 10:27:54 2011 -0500



    First Commit
```

Here is a list of all the four commits to the repository, which we were able to make so far.

01One line history

You fully control what the `log` shows. I like the single line format:

RUN:

```
git log --pretty=oneline
```

You will see …

RESULT:

```
$ git log --pretty=oneline

fa3c1411aa09441695a9e645d4371e8d749da1dc Added HTML header

8c3228730ed03116815a5cc682e8105e7d981928 Added standard HTML page tags

43628f779cb333dd30d78186499f93638107f70b Added h1 tag

911e8c91caeab8d30ad16d56746cbd6eef72dc4c First Commit
```

02Controlling the display of entries

There are many options to choose which entries appear in the log. Play around with the following parameters:

```
git log --pretty=oneline --max-count=2

git log --pretty=oneline --since='5 minutes ago'

git log --pretty=oneline --until='5 minutes ago'

git log --pretty=oneline --author=<your name>

git log --pretty=oneline --all
```

Details are provided in the `git-log` instruction.

03Getting fancy

This is what I use to review the changes made within the last week. I will add `--author=alex` if I want to see only the changes made by me.

```
git log --all --pretty=format:"%h %cd %s (%an)" --since='7 days ago'
```

04The ultimate format of the log

Over time, I found the following log format to be the most suitable.

RUN:

```
git log --pretty=format:"%h %ad | %s%d [%an]" --graph --date=short
```

It looks like this:

RESULT:

```
$ git log --pretty=format:"%h %ad | %s%d [%an]" --graph --date=short

* fa3c141 2011-03-09 | Added HTML header (HEAD, master) [Alexander Shvets]

* 8c32287 2011-03-09 | Added standard HTML page tags [Alexander Shvets]

* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]

* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

Let’s look at it in detail:

- `--pretty="..."` defines the output format.
- `%h` is the abbreviated hash of the commit
- `%d` commit decorations (e.g. branch heads or tags)
- `%ad` is the commit date
- `%s` is the comment
- `%an` is the name of the author
- `--graph` tells git to display the commit tree in the form of an ASCII graph layout
- `--date=short` keeps the date format short and nice

So, every time you want to see a log, you'll have to do a lot of typing. Fortunately, we will find out about the git aliases in the next lesson.

05Other tools

Both `gitx` (for Mac) and `gitk` (for any platform) can help to explore log history.

11. Aliases

Goals

- To learn how to setup aliases and shortcuts for git commands

01Common aliases

For Windows users:

RUN:

```
git config --global alias.co checkout

git config --global alias.ci commit

git config --global alias.st status

git config --global alias.br branch

git config --global alias.hist "log --pretty=format:'%h %ad | %s%d [%an]' --
graph --date=short"

git config --global alias.type 'cat-file -t'

git config --global alias.dump 'cat-file -p'
```

Also, for users of Unix/Mac:

git status, git add, git commit, and git checkout are common commands so it is a good idea to have abbreviations for them.
Add the following to the `.gitconfig` file in your `$HOME` directory.

FILE: ***.GITCONFIG***

```
[alias]

  co = checkout

  ci = commit

  st = status

  br = branch

  hist = log --pretty=format:\"%h %ad | %s%d [%an]\" --graph --date=short

  type = cat-file -t

  dump = cat-file -p
```

We’ve already talked about commit and status commands. In the previous lesson we covered the `log` command and will get to know the checkout command very soon. The most important thing to learn from this lesson is that you can type `git st` wherever you had to type `git status`. Best of all, the `git hist` command will help you avoid the really long `log` command.
Go ahead and try using the new commands.

02 Define the `hist` alias in the .gitconfig file

For the most part, I will continue to type out the full command in these instructions. The only exception is that I will use the `hist` alias defined above, when I need to see the git log. Make sure you have a `hist` alias setup in your `.gitconfig` file before continuing if you wish to repeat my actions.

03 Type and `Dump`

We've added a few aliases for commands we haven't yet discussed. We will talk about the `git branch` command very soon, and the `git cat-file` command is useful for exploring git.

04 Command aliases (optional)

If your shell supports aliases, or shortcuts, you can add aliases on this level, too. I use:

FILE: ***.profile***

```
alias gs='git status '

alias ga='git add '

alias gb='git branch '

alias gc='git commit'

alias gd='git diff'

alias gco='git checkout '

alias gk='gitk --all&'

alias gx='gitx --all'



alias got='git '

alias get='git '
```

The `gco` abbreviation for `git checkout` is very useful, allowing me to type:

```
gco <branch>
```

to checkout a particular branch.

Also, I often mistype `git` as `get` or `got` so I created aliases for them too.

12. Getting older versions

Goals

- To learn how to checkout any previous snapshot into the working directory.

Going back in history is very simple. The checkout command can copy any snapshot from the repo to the working directory.

01Getting hashes for the previous versions

RUN:

```
git hist
```

Note: Do not forget to define `hist` in your `.gitconfig` file? If you do not remember how, review the lesson on aliases.

RESULT:

```
$ git hist

* fa3c141 2011-03-09 | Added HTML header (HEAD, master) [Alexander Shvets]

* 8c32287 2011-03-09 | Added standard HTML page tags [Alexander Shvets]

* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]

* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

Check the log data and find the hash for the first commit. You will find it in the last line of the `git hist` data. Use the code (its first 7 chars are enough) in the command below. After that check the contents of the hello.html file.

RUN:

```
git checkout <hash>

cat hello.html
```

Note: Many commands depend on the hash values in the repository. Since my hash values will be different from yours, substitute in the appropriate hash value for your repository everytime you see `<hash>` or `<treehash>` in the command.
You will see …

RESULT:

```
$ git checkout 911e8c9

Note: checking out '911e8c9'.



You are in 'detached HEAD' state. You can look around, make experimental

changes and commit them, and you can discard any commits you make in this
```

```
state without impacting any branches by performing another checkout.



If you want to create a new branch to retain commits you create, you may

do so (now or later) by using -b with the checkout command again. Example:



  git checkout -b new_branch_name



HEAD is now at 911e8c9... First Commit

$ cat hello.html

Hello, World!
```

The `checkout` command output totally clarifies the situation. Older git versions will complain about not being on a local branch. But you don’t need to worry about that right now.
Note that the content of the `hello.html` file is the default content.

02Returning to the latest version in the master branch

RUN:

```
git checkout master

cat hello.html
```

You will see …

RESULT:

```
$ git checkout master

Previous HEAD position was 911e8c9... First Commit

Switched to branch 'master'

$ cat hello.html

<html>

  <head>

  </head>

  <body>

    <h1>Hello, World!</h1>

  </body>

</html>
```

‘master’ is the name of the default branch. By checking out a branch by name, you go to its latest version.

13. Tagging versions

Goals

- To learn how to tag commits for future references

Let’s call the current version of the hello program version 1 (v1).

01Creating a tag for the first version

RUN:

```
git tag v1
```

Now, the current version of the page is referred to as *v1*.

02Tags for previous versions

Let’s tag the version prior to the current version with the name v1-beta. First of all we will checkout the previous version. Instead of looking up the hash, we are going to use the `^` notation indicating “the parent of v1”.
If the `v1^` notation causes troubles, try using `v1~1`, referencing the same version. This notation means “the first version prior to v1”.

RUN:

```
git checkout v1^

cat hello.html
```

RESULT:

```
$ git checkout v1^

Note: checking out 'v1^'.



You are in 'detached HEAD' state. You can look around, make experimental

changes and commit them, and you can discard any commits you make in this

state without impacting any branches by performing another checkout.



If you want to create a new branch to retain commits you create, you may

do so (now or later) by using -b with the checkout command again. Example:



  git checkout -b new_branch_name



HEAD is now at 8c32287... Added standard HTML page tags
```

```
$ cat hello.html

<html>

  <body>

    <h1>Hello, World!</h1>

  </body>

</html>
```

This is the version with `<html>` and `<body>` tags, but without `<head>`. Let’s make it’s the v1-beta version.

RUN:

```
git tag v1-beta
```

03Check out by the tag name

Now try to checkout between the two tagged versions.

RUN:

```
git checkout v1

git checkout v1-beta
```

RESULT:

```
$ git checkout v1

Previous HEAD position was 8c32287... Added standard HTML page tags

HEAD is now at fa3c141... Added HTML header

$ git checkout v1-beta

Previous HEAD position was fa3c141... Added HTML header

HEAD is now at 8c32287... Added standard HTML page tags
```

04Viewing tags with the `tag` command

You can see the available tags using the `git tag` command.

RUN:

```
git tag
```

RESULT:

```
$ git tag

v1

v1-beta
```

05Viewing tags in logs

You can also check for tags in the log.

RUN:

```
git hist master --all
```

RESULT:

```
$ git hist master --all

* fa3c141 2011-03-09 | Added HTML header (v1, master) [Alexander Shvets]

* 8c32287 2011-03-09 | Added standard HTML page tags (HEAD, v1-beta) [Alexander
Shvets]

* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]

* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

You can see tags (`v1` and `v1-beta`) listed in the log together with the name of the branch (`master`). The `HEAD` shows the commit you checked out (currently `v1-beta`).

14. Discarding local changes (before staging)

Goals

- To learn how to discard the working directory changes

01Checking out the Master branch

Make sure you are on the lastest commit in the master brach before you continue.

RUN:

```
git checkout master
```

02Change hello.html

It happens that you modify a file in your local working directory and sometimes wish just to discard the committed changes. Here is when the checkout command will help you.

Make changes to the hello.html file in the form of an unwanted comment.

FILE: *HELLO.HTML*

```
<html>

  <head>

  </head>

  <body>

    <h1>Hello, World!</h1>

    <!-- This is a bad comment.  We want to revert it. -->
  </body>
</html>
```

03Check the status

First of all, check the working directory’s status.

RUN:

```
git status
```

RESULT:

```
$ git status

# On branch master

# Changes not staged for commit:

#   (use "git add <file>..." to update what will be committed)

#   (use "git checkout -- <file>..." to discard changes in working directory)

#

#   modified:   hello.html
```

```
#

no changes added to commit (use "git add" and/or "git commit -a")
```

We see that the `hello.html` file has been modified, but not staged yet.

04Undoing the changes in the working directory

Use the `checkout` command in order to checkout the repository’s version of the `hello.html` file.

RUN:

```
git checkout hello.html

git status

cat hello.html
```

RESULT:

```
$ git checkout hello.html

$ git status

# On branch master

nothing to commit (working directory clean)

$ cat hello.html

<html>

  <head>
  </head>
  <body>
    <h1>Hello, World!</h1>
  </body>
</html>
```

The status command shows there were no unstaged changes in the working directory. And the “bad comment” is no longer contained in the file.

15. Cancel Staged changes (before committing)

Goals

- To learn how to undo changes that have been staged

01 Edit file and stage changes

Make changes to the `hello.html` file in the form of an unwanted comment

FILE: *HELLO.HTML*

```
<html>

  <head>

    <!-- This is an unwanted but staged comment -->
  </head>
  <body>
    <h1>Hello, World!</h1>
  </body>
</html>
```

Stage the modified file.

RUN:

```
git add hello.html
```

02 Check the status

Check the status of unwanted changes .

RUN:

```
git status
```

RESULT:

```
$ git status

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#   modified:   hello.html

#
```

Status shows that the change has been staged and is ready to commit.

03 Reset the buffer zone

Fortunately, the displayed status shows us exactly what we should do to cancel staged changes.

RUN:

```
git reset HEAD hello.html
```

RESULT:

```
$ git reset HEAD hello.html

Unstaged changes after reset:

M   hello.html
```

The `reset` command resets the buffer zone to HEAD. This clears the buffer zone from the changes that we have just staged.
The `reset` command (default) does not change the working directory. Therefore, the working directory still contains unwanted comments. We can use the checkout command from the previous tutorial to remove unwanted changes from working directory.

04 Switch to commit version

RUN:

```
git checkout hello.html

git status
```

RESULT:

```
$ git status

# On branch master

nothing to commit (working directory clean)
```

Our working directory is clean again.

16. Cancelling commits

Goals

- To learn how to undo commits to the local repository.

01 **Cancelling commits**
Sometimes you realize that the new commits are wrong, and you want to cancel them. There are several ways to handle the issue, and we use the safest here.

To cancel the commit we will create a new commit, cancelling the unwanted changes.

02 **Edit the file and make a commit**
Replace `hello.html` with the following file.

FILE: *HELLO.HTML*
```
<html>

  <head>

  </head>

  <body>

    <h1>Hello, World!</h1>

    <!-- This is an unwanted but committed change -->
  </body>
</html>
```

RUN:
```
git add hello.html

git commit -m "Oops, we didn't want this commit"
```

03 **Make a commit with new changes that discard previous changes**
To cancel the commit, we need to create a commit that deletes the changes saved by unwanted commit.

RUN:
```
git revert HEAD
```

Go to the editor, where you can edit the default commit message or leave it as is. Save and close the file.

You will see …

RESULT:
```
$ git revert HEAD --no-edit

[master 45fa96b] Revert "Oops, we didn't want this commit"

 1 files changed, 1 insertions(+), 1 deletions(-)
```

Since we have cancelled the last commit, we can use `HEAD` as the argument for cancelling. We may cancel any random commit in history, pointing out its hash value.
Note: The `--no-edit` command can be ignored. It was necessary to generate the output data without opening the editor.

04 Check the log

Checking the log shows the unwanted cancellations and commits in our repository.

RUN:

```
git hist
```

RESULT:

```
$ git hist

* 45fa96b 2011-03-09 | Revert "Oops, we didn't want this commit" (HEAD, master) [Alexander Shvets]

* 846b90c 2011-03-09 | Oops, we didn't want this commit [Alexander Shvets]

* fa3c141 2011-03-09 | Added HTML header (v1) [Alexander Shvets]

* 8c32287 2011-03-09 | Added standard HTML page tags (v1-beta) [Alexander Shvets]

* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]

* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

This technique can be applied to any commit (however there may be conflicts). It is safe to use even in public branches of remote repositories.

17. Removing a commit from a branch

Goals

- To learn to delete the branch's latest commits

`Revert` is a powerful command of the previous section that allows you to cancel any commits to the repository. However, both original and cancelled commits are seen in the history of the branch (when using `git log` command).
Often after a commit is already made, we realize it was a mistake. It would be nice to have an undo command which allows the incorrect commit(s) to be immediately deleted. This command would prevent the appearance of one or more unwanted commits in the `git log` history.

01 The `reset` command

We have already used the `reset` command to match the buffer zone and the selected commit (HEAD commit was used in the previous lesson).
When a commit reference is given (ie, a branch, hash, or tag name), the `reset` command will...

1. Overwrite the current branch so it will point to the correct commit
2. Optionally reset the buffer zone so it will comply with the specified commit
3. Optionally reset the working directory so it will match the specified commit

02 Check our history

Let us do a quick scan of our commit history.

RUN:

```
git hist
```

RESULT:

```
$ git hist

* 45fa96b 2011-03-09 | Revert "Oops, we didn't want this commit" (HEAD, master)
[Alexander Shvets]

* 846b90c 2011-03-09 | Oops, we didn't want this commit [Alexander Shvets]

* fa3c141 2011-03-09 | Added HTML header (v1) [Alexander Shvets]

* 8c32287 2011-03-09 | Added standard HTML page tags (v1-beta) [Alexander
Shvets]

* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]

* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

We see the last two commits in this branch are "Oops" and "Revert Oops". Let us remove them with the `reset` command.

03 Mark this branch first

Let us mark the last commit with `tag, so you can find it after removing a commit(s).`

RUN:

```
git tag oops
```

04 Reset commit to previous Oops

In the history log above, the commit tagged «v1» is before the "Oops" and "Revert Oops" commits. Let us reset the branch to that point. As the branch has a tag, we can use the tag name in the reset command (if it does not have a tag, we can use the hash value).

RUN:

```
git reset --hard v1

git hist
```

RESULT:

```
$ git reset --hard v1

HEAD is now at fa3c141 Added HTML header

$ git hist

* fa3c141 2011-03-09 | Added HTML header (HEAD, v1, master) [Alexander Shvets]

* 8c32287 2011-03-09 | Added standard HTML page tags (v1-beta) [Alexander Shvets]

* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]

* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

Our master branch is pointing at commit v1 and the "Revert Oops" and "Oops" commits no longer exist in the branch. The `--hard` parameter makes the working directory reflect the new branch head.

05 Nothing is ever lost

What happened to the wrong commits? They are still in the repository. Actually, we can still refer to them. At the beginning of the lesson, we created the «oops» tag for the canceled commit. Let us take a look at *all* commits.

RUN:

```
git hist --all
```

RESULT:

```
$ git hist --all

* 45fa96b 2011-03-09 | Revert "Oops, we didn't want this commit" (oops) [Alexander Shvets]

* 846b90c 2011-03-09 | Oops, we didn't want this commit [Alexander Shvets]

* fa3c141 2011-03-09 | Added HTML header (HEAD, v1, master) [Alexander Shvets]

* 8c32287 2011-03-09 | Added standard HTML page tags (v1-beta) [Alexander Shvets]

* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]

* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

We can see that the wrong commits are not gone. They are not listed in the master branch anymore but still remain in the repository. They would still be in the repository if we did not tag them, but then we could reference them only by their hash names. Unreferenced commits remain in the repository until the garbage collection software is run by system.

06 Reset dangers

Resets on local branches are usually harmless. The consequences of any "accident" can be reverted by using the proper commit.

However, other users sharing the branch can be confused if the branch is shared on remote repositories.

18. Removing the oops tag

Goals

- Removing the oops tag (cleaning up)

01 Removal of the oops tag

Oops tag has performed it’s function. Let us remove that tag and permit the garbage collector to delete referenced commit.

RUN:

```
git tag -d oops

git hist --all
```

RESULT:

```
$ git tag -d oops

Deleted tag 'oops' (was 45fa96b)

$ git hist --all

* fa3c141 2011-03-09 | Added HTML header (HEAD, v1, master) [Alexander Shvets]

* 8c32287 2011-03-09 | Added standard HTML page tags (v1-beta) [Alexander
Shvets]

* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]

* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

Oops tag will no longer appear in the repository.

19. Changing commits

Goals

- To learn how to modify an already existing commit

01 Change the page and commit

Put an author comment on the page.

FILE: *HELLO.HTML*

```
<!-- Author: Alexander Shvets -->
<html>
  <head>
  </head>
  <body>
    <h1>Hello, World!</h1>
  </body>
</html>
```

RUN:

```
git add hello.html

git commit -m "Add an author comment"
```

02 Oops... email required

After making the commit you understand that every good comment should include the author's email. Edit the hello page to provide an email.

FILE: *HELLO.HTML*

```
<!-- Author: Alexander Shvets (alex@githowto.com) -->
<html>
  <head>
  </head>
  <body>
    <h1>Hello, World!</h1>
  </body>
</html>
```

03 Change the previous commit

We do not want to create another commit for adding the e-mail address. Let us change the previous commit and add an e-mail address.

RUN:

```
git add hello.html

git commit --amend -m "Add an author/email comment"
```

RESULT:

```
$ git add hello.html

$ git commit --amend -m "Add an author/email comment"
```

```
[master 6a78635] Add an author/email comment

 1 files changed, 2 insertions(+), 1 deletions(-)
```

04View history

RUN:

```
git hist
```

RESULT:

```
$ git hist

* 6a78635 2011-03-09 | Add an author/email comment (HEAD, master) [Alexander Shvets]

* fa3c141 2011-03-09 | Added HTML header (v1) [Alexander Shvets]

* 8c32287 2011-03-09 | Added standard HTML page tags (v1-beta) [Alexander Shvets]

* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]

* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

The new "author/email" commit replaces the original "author" commit. The same effect can be achieved by resetting the last commit in the branch, and recommitting new changes.

20. Moving files

Goals

- To learn how to move a file within the repository.

01Move the hello.html file to the *lib* directory

Now we will create the structure in our repository. Let us move the page in the lib directory.

RUN:

```
mkdir lib

git mv hello.html lib

git status
```

RESULT:

```
$ mkdir lib

$ git mv hello.html lib

$ git status

# On branch master

# Changes to be committed:

#   (use "git reset HEAD <file>..." to unstage)

#

#   renamed:    hello.html -> lib/hello.html

#
```

By moving files with git, we notify git about two things

1. The `hello.html` file was deleted.
2. The `lib/hello.html` file was created.

Both facts are staged immediately and ready for a commit. Git status command reports the file has been moved.

02One more way to move files

A positive fact about git is that you don’t need to think about version control the moment when you need to commit code. What would happen if we were using the operating system command line instead of the git command to move files?

The following set of commands have the same result as the ones we used above, but the ones below require a little more work.

We can do:

```
mkdir lib
```

```
mv hello.html lib

git add lib/hello.html

git rm hello.html
```

03 Commit new directory

Let us commit this movement.

RUN:

```
git commit -m "Moved hello.html to lib"
```

21. More information about the structure

Goals

- Add one more file in our repository

01Adding index.html

Let us add an `index.html` file to the repository. The following file is perfect for this purpose.

FILE: *INDEX.HTML*

```
<html>

  <body>

    <iframe src="lib/hello.html" width="200" height="200" />

  </body>

</html>
```

Add the file and make a commit.

RUN:

```
git add index.html

git commit -m "Added index.html."
```

Now when you open `index.html`, you should see a part of the hello page in a small window.

22. Inside Git: .Git directory

Goals

- To learn about Git directory structure `.git`

01The `.git` directory

It is time to do some research. Starting from the project's root directory...

RUN:

```
ls -C .git
```

RESULT:

```
$ ls -C .git

COMMIT_EDITMSG  MERGE_RR    config      hooks       info        objects     rr-
cache

HEAD        ORIG_HEAD   description index       logs        refs
```

This is a special folder where all the git stuff is. Let us explore the directory.

02Object Database

RUN:

```
ls -C .git/objects
```

RESULT:

```
$ ls -C .git/objects

09  24  28  45  59  6a  77  80  8c  97  af  c4  e7  info

11  27  43  56  69  6b  78  84  91  9c  b5  e4  fa  pack
```

You should see a lot of folders named with two characters. The first two letters of the SHA1 hash of the objects stored in git are the directory names.

03Inquire the database objects

RUN:

```
ls -C .git/objects/<dir>
```

RESULT:

```
$ ls -C .git/objects/09

6b74c56bfc6b40e754fc0725b8c70b2038b91e  9fb6f9d3a104feb32fcac22354c4d0e8a182c1
```

Let us look at one of the folders named with two characters. There should be files with names of 38 characters. These files contain objects stored in git. They are compressed and encrypted, so it's impossible to view their contents directly. Let us have a better look at Git directory

04Config File

RUN:

```
cat .git/config
```

RESULT:

```
$ cat .git/config

[core]

    repositoryformatversion = 0

    filemode = true

    bare = false

    logallrefupdates = true

    ignorecase = true

[user]

    name = Alexander Shvets

    email = alex@githowto.com
```

This configuration file is created for each individual project. At least in this project, entries in this file will overwrite the entries in the `.gitconfig` file of your main directory.

05 Branches and tags

RUN:

```
ls .git/refs

ls .git/refs/heads

ls .git/refs/tags

cat .git/refs/tags/v1
```

RESULT:

```
$ ls .git/refs

heads

tags

$ ls .git/refs/heads

master

$ ls .git/refs/tags

v1

v1-beta

$ cat .git/refs/tags/v1
```

```
fa3c1411aa09441695a9e645d4371e8d749da1dc
```

Files in the tags subdirectory should be familiar to you. Each file corresponds to the tag previously created using the git tag command. Its content is nothing but a hash commit attached to the tag.

The *heads* folder is almost identical and is used not for tags, but branches. At the moment we have only one branch, and everything you see in this folder is a *master* branch.

06HEAD File

RUN:

```
cat .git/HEAD
```

RESULT:

```
$ cat .git/HEAD

ref: refs/heads/master
```

There is a reference to the current branch in the HEAD file. At the moment it must be the master branch.

24. Creating a Branch

Goals

- To learn how to create a local branch in the repository

It is time to make our hello world more expressive. Since it may take some time, it is best to move these changes into a new branch to isolate them from master branch changes.

01Create a branch
Let us name our new branch «style».

RUN:
```
git checkout -b style

git status
```

Note: `git checkout -b <branch name>` is a shortcut for `git branch <branch name>` followed by a `git checkout <branch name>`.
Note that the `git status` command reports that you are in the style branch.

02Add style.css file

RUN:
```
touch lib/style.css
```

FILE: ***LIB/STYLE.CSS***
```
h1 {

  color: red;

}
```

RUN:
```
git add lib/style.css

git commit -m "Added css stylesheet"
```

03Change the main page
Update the `hello.html` file, to use style.css.

FILE: ***LIB/HELLO.HTML***
```
<!-- Author: Alexander Shvets (alex@githowto.com) -->

<html>

  <head>

    <link type="text/css" rel="stylesheet" media="all" href="style.css" />
  </head>
  <body>
    <h1>Hello, World!</h1>
  </body>
</html>
```

RUN:

```
git add lib/hello.html

git commit -m "Hello uses style.css"
```

04Change index.html

Update the `index.html` file, so it uses `style.css`

FILE: *INDEX.HTML*

```
<html>

  <head>
    <link type="text/css" rel="stylesheet" media="all" href="lib/style.css" />
  </head>
  <body>
    <iframe src="lib/hello.html" width="200" height="200" />
  </body>
</html>
```

RUN:

```
git add index.html

git commit -m "Updated index.html"
```

25. Navigating Branches

Goals

- To learn how to navigate between the repository branches

Now your project has two branches:

RUN:

```
git hist --all
```

RESULT:

```
$ git hist --all

* 07a2a46 2011-03-09 | Updated index.html (HEAD, style) [Alexander Shvets]

* 649d26c 2011-03-09 | Hello uses style.css [Alexander Shvets]

* 1f3cbd2 2011-03-09 | Added css stylesheet [Alexander Shvets]

* 8029c07 2011-03-09 | Added index.html. (master) [Alexander Shvets]

* 567948a 2011-03-09 | Moved hello.html to lib [Alexander Shvets]

* 6a78635 2011-03-09 | Add an author/email comment [Alexander Shvets]

* fa3c141 2011-03-09 | Added HTML header (v1) [Alexander Shvets]

* 8c32287 2011-03-09 | Added standard HTML page tags (v1-beta) [Alexander
Shvets]

* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]

* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

01 Switching to the Master branch

To switch between branches simply use the `git checkout` command.

RUN:

```
git checkout master

cat lib/hello.html
```

RESULT:

```
$ git checkout master

Switched to branch 'master'

$ cat lib/hello.html

<!-- Author: Alexander Shvets (alex@githowto.com) -->

<html>
```

```
  <head>

  </head>

  <body>

    <h1>Hello, World!</h1>

  </body>

</html>
```

Now we are on the Master branch. It can be proven by the fact the `hello.html` file does not use styles from `style.css`.

02**Let us return to the style branch.**

RUN:

```
git checkout style

cat lib/hello.html
```

RESULT:

```
$ git checkout style

Switched to branch 'style'

$ cat lib/hello.html

<!-- Author: Alexander Shvets (alex@githowto.com) -->

<html>

  <head>

    <link type="text/css" rel="stylesheet" media="all" href="style.css" />

  </head>

  <body>

    <h1>Hello, World!</h1>

  </body>

</html>
```

We are back to the **style** branch which can be proven by the fact the `hello.html` file uses styles from `style.css`

26. Changes to master branch

Goals

- To learn how to work with several branches with different (sometimes conflicting) changes.

At the time you are changing the style branch, someone decided to change the master branch. He added a README file.

01 Update the README file with the changes.

FILE: *README*

```
This is the Hello World example from the git_G.
```

02 Commit changes of README file in the master branch.

RUN:

```
git checkout master

git add README

git commit -m "Added README"
```

27. View the different branches

Goals

- To learn how to view the different branches in the repository.

01View current branches

Now we have a repository with two different branches. To view branches and their differences use log command as follows.

RUN:

```
git hist --all
```

RESULT:

```
$ git hist --all

* 6c0f848 2011-03-09 | Added README (HEAD, master) [Alexander Shvets]

| * 07a2a46 2011-03-09 | Updated index.html (style) [Alexander Shvets]

| * 649d26c 2011-03-09 | Hello uses style.css [Alexander Shvets]

| * 1f3cbd2 2011-03-09 | Added css stylesheet [Alexander Shvets]

|/

* 8029c07 2011-03-09 | Added index.html. [Alexander Shvets]

* 567948a 2011-03-09 | Moved hello.html to lib [Alexander Shvets]

* 6a78635 2011-03-09 | Add an author/email comment [Alexander Shvets]

* fa3c141 2011-03-09 | Added HTML header (v1) [Alexander Shvets]

* 8c32287 2011-03-09 | Added standard HTML page tags (v1-beta) [Alexander
Shvets]

* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]

* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

We have an opportunity to see `--graph` of `git hist` in action. Adding the `--graph` option to `git log` causes the construction of a commit tree with the help of simple ASCII characters. We see both branches (style and master) and that the current branch is master HEAD. The Added index.html branch goes prior to both branches.
The `--all` flag guarantees that we see all the branches. By default, only the current branch is displayed.

28. Merging

Goals

- To learn how to merge two distinct branches to restore changes to a single branch.

01Merging to a single branch

Merging brings changes from two branches into one. Let us go back to the style branch and merge it with master.

RUN:

```
git checkout style

git merge master

git hist --all
```

RESULT:

```
$ git checkout style

Switched to branch 'style'

$ git merge master

Merge made by recursive.

 README |    1 +

 1 files changed, 1 insertions(+), 0 deletions(-)

 create mode 100644 README

$ git hist --all

*   5813a3f 2011-03-09 | Merge branch 'master' into style (HEAD, style)
[Alexander Shvets]

|\

| * 6c0f848 2011-03-09 | Added README (master) [Alexander Shvets]

* | 07a2a46 2011-03-09 | Updated index.html [Alexander Shvets]

* | 649d26c 2011-03-09 | Hello uses style.css [Alexander Shvets]

* | 1f3cbd2 2011-03-09 | Added css stylesheet [Alexander Shvets]

|/

* 8029c07 2011-03-09 | Added index.html. [Alexander Shvets]

* 567948a 2011-03-09 | Moved hello.html to lib [Alexander Shvets]
* 6a78635 2011-03-09 | Add an author/email comment [Alexander Shvets]

* fa3c141 2011-03-09 | Added HTML header (v1) [Alexander Shvets]

* 8c32287 2011-03-09 | Added standard HTML page tags (v1-beta) [Alexander
Shvets]

* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]

* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

Through periodic master branch merging with the style branch you can pick up any changes or modifications to the master to maintain compatibility with the style changes in the mainline.

However, this makes the commit graphics look ugly. Later we will consider relocation as an alternative to fusion.

02Next

But what if changes to the master branch conflict with changes in style?

29. Creating a conflict

Goals

- Creating a conflicting of changes in the master branch.

01**Return to the master and create conflict**

Return to the master branch and make the following changes:

```
git checkout master
```

FILE: *LIB/HELLO.HTML*

```
<!-- Author: Alexander Shvets (alex@githowto.com) -->

<html>

  <head>

    <!-- no style -->
  </head>
  <body>
    <h1>Hello, World! Life is great!</h1>
  </body>
</html>
```

RUN:

```
git add lib/hello.html

git commit -m 'Life is great!'
```

(**Warning:** make sure you've used single-quotes to avoid problems with bash and the `!` character)

02**View branches**

RUN:

```
git hist --all
```

RESULT:

```
$ git hist --all

* 454ec68 2011-03-09 | Life is great! (HEAD, master) [Alexander Shvets]

| * 5813a3f 2011-03-09 | Merge branch 'master' into style (style) [Alexander
Shvets]

| |\

| |/

|/|

* | 6c0f848 2011-03-09 | Added README [Alexander Shvets]

| * 07a2a46 2011-03-09 | Updated index.html [Alexander Shvets]
| * 649d26c 2011-03-09 | Hello uses style.css [Alexander Shvets]

| * 1f3cbd2 2011-03-09 | Added css stylesheet [Alexander Shvets]

|/

* 8029c07 2011-03-09 | Added index.html. [Alexander Shvets]

* 567948a 2011-03-09 | Moved hello.html to lib [Alexander Shvets]

* 6a78635 2011-03-09 | Add an author/email comment [Alexander Shvets]

* fa3c141 2011-03-09 | Added HTML header (v1) [Alexander Shvets]

* 8c32287 2011-03-09 | Added standard HTML page tags (v1-beta) [Alexander
Shvets]

* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]

* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

After the Added README commit, the master branch has been merged with the style branch, but there is an additional master commit, which was not merged back to the style branch.

03 **Next**

The last change in master conflicts with some changes in style. In the next step we will solve this conflict.

30. Resolving Conflicts

Goals

- To learn to resolve merging conflicts

01Merge the master branch with style

Let us go back to the style branch and merge it with a new master branch.

RUN:

```
git checkout style

git merge master
```

RESULT:

```
$ git checkout style

Switched to branch 'style'

$ git merge master

Auto-merging lib/hello.html

CONFLICT (content): Merge conflict in lib/hello.html

Automatic merge failed; fix conflicts and then commit the result.
```

If you open the `lib/hello.html` you will see:

FILE: *LIB/HELLO.HTML*

```
<!-- Author: Alexander Shvets (alex@githowto.com) -->

<html>

  <head>

<<<<<<< HEAD

    <link type="text/css" rel="stylesheet" media="all" href="style.css" />

=======

    <!-- no style -->

>>>>>>> master

  </head>

  <body>

    <h1>Hello,World! Life is great!</h1>

  </body>
```

```
</html>
```

The first section is the version of the current branch (style) head. The second section is the version of master branch.

02Resolution of the conflict

You need to resolve the conflict manually. Make changes to `lib/hello.html` to achieve the following result.

FILE: *LIB/HELLO.HTML*

```
<!-- Author: Alexander Shvets (alex@githowto.com) -->

<html>

  <head>

    <link type="text/css" rel="stylesheet" media="all" href="style.css" />

  </head>

  <body>

    <h1>Hello, World! Life is great!</h1>

  </body>

</html>
```

03Make a commit of conflict resolution

RUN:

```
git add lib/hello.html

git commit -m "Merged master fixed conflict."
```

RESULT:

```
$ git add lib/hello.html

$ git commit -m "Merged master fixed conflict."

Recorded resolution for 'lib/hello.html'.

[style 645c4e6] Merged master fixed conflict.
```

04Advanced Merging

Git has no graphical merging tools, but it will accept any third-party merge tool (read more about such tools on StackOverflow.)

31. Relocating as an alternative to merging

Goals

- To learn the difference between relocating and merging.

Discussion

Let us look at the differences between relocating and merging. To do this, we need to get back into the repository at the time prior to the first merge, and then repeat the same steps but using relocating instead of merging.

We will use the reset command to return the branch to a previous state.

32. Resetting the style branch

Goals

- Resetting the branch style to the point prior to the first merge.

01Resetting the style branch
Let us go to the style branch to the point *before* we merged it with the master branch. We can **reset** the branch to any commit. In fact, **reset** can change the branch pointer to point to any commit in the tree.
Here, we want to go back in the style branch to a point before merging with the master. We have to find the last commit prior to the merge.

RUN:

```
git checkout style

git hist
```

RESULT:

```
$ git checkout style

Already on 'style'

$ git hist

*   645c4e6 2011-03-09 | Merged master fixed conflict. (HEAD, style) [Alexander Shvets]

|\

| * 454ec68 2011-03-09 | Life is great! (master) [Alexander Shvets]

* |   5813a3f 2011-03-09 | Merge branch 'master' into style [Alexander Shvets]

|\ \

| |/

| * 6c0f848 2011-03-09 | Added README [Alexander Shvets]

* | 07a2a46 2011-03-09 | Updated index.html [Alexander Shvets]

* | 649d26c 2011-03-09 | Hello uses style.css [Alexander Shvets]

* | 1f3cbd2 2011-03-09 | Added css stylesheet [Alexander Shvets]

|/

* 8029c07 2011-03-09 | Added index.html. [Alexander Shvets]

* 567948a 2011-03-09 | Moved hello.html to lib [Alexander Shvets]

* 6a78635 2011-03-09 | Add an author/email comment [Alexander Shvets]
* fa3c141 2011-03-09 | Added HTML header (v1) [Alexander Shvets]

* 8c32287 2011-03-09 | Added standard HTML page tags (v1-beta) [Alexander
Shvets]

* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]

* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

It's a little hard to read, but we can see from the output that the Updated index.html commit was the latest on the style branch prior to merging. Let us reset the style branch to this commit.

RUN:

```
git reset --hard <hash>
```

RESULT:

```
$ git reset --hard 07a2a46

HEAD is now at 07a2a46 Updated index.html
```

02**Check the branch.**
Look for the style branch log. There are no merge commits in our history.

RUN:

```
git hist --all
```

RESULT:

```
$ git hist --all

* 454ec68 2011-03-09 | Life is great! (master) [Alexander Shvets]

* 6c0f848 2011-03-09 | Added README [Alexander Shvets]

| * 07a2a46 2011-03-09 | Updated index.html (HEAD, style) [Alexander Shvets]

| * 649d26c 2011-03-09 | Hello uses style.css [Alexander Shvets]

| * 1f3cbd2 2011-03-09 | Added css stylesheet [Alexander Shvets]

|/

* 8029c07 2011-03-09 | Added index.html. [Alexander Shvets]

* 567948a 2011-03-09 | Moved hello.html to lib [Alexander Shvets]

* 6a78635 2011-03-09 | Add an author/email comment [Alexander Shvets]

* fa3c141 2011-03-09 | Added HTML header (v1) [Alexander Shvets]

* 8c32287 2011-03-09 | Added standard HTML page tags (v1-beta) [Alexander
Shvets]

* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]
* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

33. Reset of the Master branch

Goals

- Reset the master branch to the point prior to the conflicting commit.

01Resetting the master branch

The interactive mode we added to the master branch has become a change conflicting with the changes in the style branch. Let’s revert the changes in the master branch up to the point before the conflict change was made. This allows us to demonstrate the rebase command without having to worry about conflicts.

RUN:

```
git checkout master

git hist
```

RESULT:

```
$ git hist

* 454ec68 2011-03-09 | Life is great! (HEAD, master) [Alexander Shvets]

* 6c0f848 2011-03-09 | Added README [Alexander Shvets]

* 8029c07 2011-03-09 | Added index.html. [Alexander Shvets]

* 567948a 2011-03-09 | Moved hello.html to lib [Alexander Shvets]

* 6a78635 2011-03-09 | Add an author/email comment [Alexander Shvets]

* fa3c141 2011-03-09 | Added HTML header (v1) [Alexander Shvets]

* 8c32287 2011-03-09 | Added standard HTML page tags (v1-beta) [Alexander
Shvets]

* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]

* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

The "Added README" commit goes directly before the conflicting interactive mode we added. Right now we need to reset the master branch to the "Added README" branch.

RUN:

```
git reset --hard <hash>

git hist --all
```

Examine the log. It should look as if we rewound the repository to a point in time, prior to any mergers.

RESULT:

```
$ git reset --hard 6c0f848
```

```
HEAD is now at 6c0f848 Added README

$ git hist --all

* 6c0f848 2011-03-09 | Added README (HEAD, master) [Alexander Shvets]

| * 07a2a46 2011-03-09 | Updated index.html (style) [Alexander Shvets]

| * 649d26c 2011-03-09 | Hello uses style.css [Alexander Shvets]

| * 1f3cbd2 2011-03-09 | Added css stylesheet [Alexander Shvets]

|/

* 8029c07 2011-03-09 | Added index.html. [Alexander Shvets]

* 567948a 2011-03-09 | Moved hello.html to lib [Alexander Shvets]

* 6a78635 2011-03-09 | Add an author/email comment [Alexander Shvets]

* fa3c141 2011-03-09 | Added HTML header (v1) [Alexander Shvets]

* 8c32287 2011-03-09 | Added standard HTML page tags (v1-beta) [Alexander
Shvets]

* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]

* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

34. Rebase

Goals

- To use rebase instead of the merge command.

So we went back in history before the first merge and wanna relocate the changes in master to our style branch.

This time we are going to use the rebase command rather than merge.

RUN:

```
git checkout style

git rebase master

git hist
```

RESULT:

```
$ git checkout style

Switched to branch 'style'

$ git rebase master

First, rewinding head to replay your work on top of it...

Applying: Added css stylesheet

Applying: Hello uses style.css

Applying: Updated index.html

$ git hist

* 6e6c76a 2011-03-09 | Updated index.html (HEAD, style) [Alexander Shvets]

* 1436f13 2011-03-09 | Hello uses style.css [Alexander Shvets]

* 59da9a7 2011-03-09 | Added css stylesheet [Alexander Shvets]

* 6c0f848 2011-03-09 | Added README (master) [Alexander Shvets]

* 8029c07 2011-03-09 | Added index.html. [Alexander Shvets]

* 567948a 2011-03-09 | Moved hello.html to lib [Alexander Shvets]

* 6a78635 2011-03-09 | Add an author/email comment [Alexander Shvets]

* fa3c141 2011-03-09 | Added HTML header (v1) [Alexander Shvets]

* 8c32287 2011-03-09 | Added standard HTML page tags (v1-beta) [Alexander
Shvets]
```

```
* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]

* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

01Merging VS rebasing

The result of the rebase command looks much like that of the merge command. The style branch currently contains all its changes, plus all the changes of the master branch. The commit tree, however, is a bit different. The style branch commit tree has been rewritten to make the master branch a part of the commit history. This makes the chain of commits linear and more readable.

02When to use the rebase command, and when the merge command?

Don’t use the rebase command …

1. If the branch is public and shared. Rewriting such branches will hinder the work of other team members.
2. When the *exact* commit branch history is important (because the rebase command rewrites the history of commits).

Given the above recommendations, I prefer to use rebase for short-term, local branches and the merge command for branches in the public repository.

35. Merging to the Master branch

Goals

- We have kept our style branch up to date with the master branch (using rebase), but now let's merge the style branch changes back into the master.

01 Merging style into master

RUN:

```
git checkout master

git merge style
```

RESULT:

```
$ git checkout master

Switched to branch 'master'

$ git merge style

Updating 6c0f848..6e6c76a

Fast-forward

 index.html     | 2 +-

 lib/style.css  | 8 ++++++++

 lib/hello.html | 6 ++++--

 3 files changed, 13 insertions(+), 3 deletions(-)

 create mode 100644 lib/style.css
```

Since the last master commit directly precedes the last commit of the style branch, git can merge fast-forward by simply moving the branch pointer forward, pointing to the same commit as the style branch.

Conflicts do not arise in the fast-forward merge.

02 Check the logs

RUN:

```
git hist
```

RESULT:

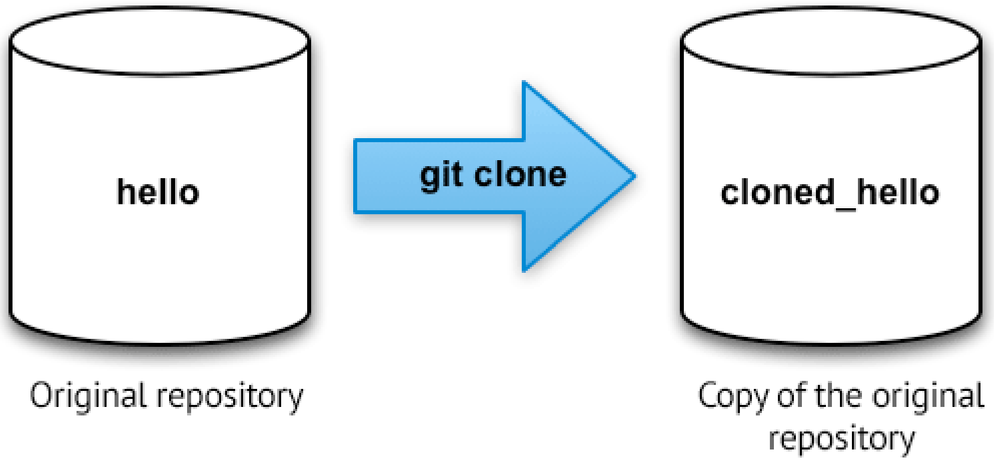
```
$ git hist

* 6e6c76a 2011-03-09 | Updated index.html (HEAD, master, style) [Alexander
Shvets]

* 1436f13 2011-03-09 | Hello uses style.css [Alexander Shvets]
* 59da9a7 2011-03-09 | Added css stylesheet [Alexander Shvets]

* 6c0f848 2011-03-09 | Added README [Alexander Shvets]

* 8029c07 2011-03-09 | Added index.html. [Alexander Shvets]

* 567948a 2011-03-09 | Moved hello.html to lib [Alexander Shvets]

* 6a78635 2011-03-09 | Add an author/email comment [Alexander Shvets]

* fa3c141 2011-03-09 | Added HTML header (v1) [Alexander Shvets]

* 8c32287 2011-03-09 | Added standard HTML page tags (v1-beta) [Alexander
Shvets]

* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]

* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

Now style and master are identical.

36. Multiple repositories

So far we have been working with only one git repository. However, git is great for working with several repositories. These additional repositories can be stored locally, or accessed via network connection.

In the next section we will create a new repo called “cloned_hello”. We will discuss moving changes from one repo to another, and dealing with conflicts when working with two repositories.



Meanwhile, we will work with local repositories (the ones stored on your local HDD). Most of the information in this section can be also applied to working with multiple repositories no matter if they are stored locally or shared over a network.

NOTE: We will make changes to both copies of our repositories. Notice the repository you are on every stage of the next lessons.

37. Cloning repositories

Goals

- To learn how to make copies of the repositories.

If you are working in a team, the following 12 chapters are quite important to understand because you almost always have to work with cloned repositories.

01 Go to your working directory

Go to the working directory and clone your hello repository.

RUN:

```
cd ..

pwd

ls
```

NOTE: Now we are in the work directory.

RESULT:

```
$ cd ..

$ pwd

/Users/alex/Documents/Presentations/githowto/auto

$ ls

hello
```

At this point you should be in your “working” directory. It should contain a single repository named “hello”.

02 Create a clone of the hello repository

Let’s create a clone of the repository.

RUN:

```
git clone hello cloned_hello

ls
```

RESULT:

```
$ git clone hello cloned_hello

Cloning into cloned_hello...

done.

$ ls

cloned_hello
```

```
hello
```

Right now there should now be two repos in your working directory: the original “hello” repo and the cloned repository named “cloned_hello”.

38. Examine the cloned repository

Goals

- To find out about branches in the remote repositories.

01**Viewing the cloned repository**
Let's have a look at our cloned repository.

RUN:

```
cd cloned_hello

ls
```

RESULT:

```
$ cd cloned_hello

$ ls

README

index.html

lib
```

You will see a list of all files in the top level of the original repository (`README`, `index.html` and `lib`).
02**View the history of the cloned repository**

RUN:

```
git hist --all
```

RESULT:

```
$ git hist --all

* 6e6c76a 2011-03-09 | Updated index.html (HEAD, origin/master, origin/style,
origin/HEAD, master) [Alexander Shvets]

* 1436f13 2011-03-09 | Hello uses style.css [Alexander Shvets]

* 59da9a7 2011-03-09 | Added css stylesheet [Alexander Shvets]

* 6c0f848 2011-03-09 | Added README [Alexander Shvets]

* 8029c07 2011-03-09 | Added index.html. [Alexander Shvets]

* 567948a 2011-03-09 | Moved hello.html to lib [Alexander Shvets]

* 6a78635 2011-03-09 | Add an author/email comment [Alexander Shvets]

* fa3c141 2011-03-09 | Added HTML header (v1) [Alexander Shvets]
* 8c32287 2011-03-09 | Added standard HTML page tags (v1-beta) [Alexander Shvets]

* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]

* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

You will see a list of all the commits in the new repository, and it should match the commit history of the original repository. The only difference should be in the names of the branches.

03 Remote branches

You will see a **master** branch (**HEAD**) in the history. You will also find branches with strange names (**origin/master**, **origin/style** and **origin/HEAD**). We'll discuss them a bit later.

38. Examine the cloned repository

Goals

- To find out about branches in the remote repositories.

01 Viewing the cloned repository

Let's have a look at our cloned repository.

RUN:

```
cd cloned_hello

ls
```

RESULT:

```
$ cd cloned_hello

$ ls

README

index.html

lib
```

You will see a list of all files in the top level of the original repository (`README`, `index.html` and `lib`).

02 View the history of the cloned repository

RUN:

```
git hist --all
```

RESULT:

```
$ git hist --all

* 6e6c76a 2011-03-09 | Updated index.html (HEAD, origin/master, origin/style,
origin/HEAD, master) [Alexander Shvets]

* 1436f13 2011-03-09 | Hello uses style.css [Alexander Shvets]

* 59da9a7 2011-03-09 | Added css stylesheet [Alexander Shvets]

* 6c0f848 2011-03-09 | Added README [Alexander Shvets]

* 8029c07 2011-03-09 | Added index.html. [Alexander Shvets]

* 567948a 2011-03-09 | Moved hello.html to lib [Alexander Shvets]

* 6a78635 2011-03-09 | Add an author/email comment [Alexander Shvets]

* fa3c141 2011-03-09 | Added HTML header (v1) [Alexander Shvets]
* 8c32287 2011-03-09 | Added standard HTML page tags (v1-beta) [Alexander
Shvets]

* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]

* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

You will see a list of all the commits in the new repository, and it should match the commit history of the original repository. The only difference should be in the names of the branches.

03 Remote branches

You will see a **master** branch (**HEAD**) in the history. You will also find branches with strange names (**origin/master**, **origin/style** and **origin/HEAD**). We'll discuss them a bit later.

40. Remote branches

Goals

- To learn about local and remote branches

Let’s take a look at the branches in our cloned repository.

RUN:

```
git branch
```

RESULT:

```
$ git branch

* master
```

As we can see only the master branch is listed in it. Where is the style branch? `git branch` only lists the local branches by default.

*01*01 List of the remote branches

To see all the branches, try the following command:

RUN:

```
git branch -a
```

RESULT:

```
$ git branch -a

* master

  remotes/origin/HEAD -> origin/master

  remotes/origin/style

  remotes/origin/master
```

Git lists all the branches from the original repo, but the remote repository branches are not treated as local ones. If we need our own **style** branch, we need to create it on our own. In a minute you will see how it is done.

41. Changing the original repository

Goals

- To make changes to the original repository so we can try to pull the changes

01Make a change in the original hello repository

RUN:

```
cd ../hello

# (You should be in the original hello repository now)
```

NOTE: We are now in the repository *hello*

Make the following changes to the README file:

FILE: *README*

```
This is the Hello World example from the git_G.

(changed in original)
```

Now add and commit this change

RUN:

```
git add README

git commit -m "Changed README in original repo"
```

02Next

Now the original repo has more recent changes that are not included in the cloned version. Next we will pull those changes across to the cloned repo.

42. Fetching changes

Goals

- To learn how to pull changes from a remote repository.

RUN:

```
cd ../cloned_hello

git fetch

git hist --all
```

NOTE: We are now in the repository *cloned_hello*.

RESULT:

```
$ git fetch

From /Users/alex/Documents/Presentations/githowto/auto/hello

   6e6c76a..2faa4ea  master     -> origin/master

$ git hist --all

* 2faa4ea 2011-03-09 | Changed README in original repo (origin/master,
origin/HEAD) [Alexander Shvets]

* 6e6c76a 2011-03-09 | Updated index.html (HEAD, origin/style, master)
[Alexander Shvets]

* 1436f13 2011-03-09 | Hello uses style.css [Alexander Shvets]

* 59da9a7 2011-03-09 | Added css stylesheet [Alexander Shvets]

* 6c0f848 2011-03-09 | Added README [Alexander Shvets]

* 8029c07 2011-03-09 | Added index.html. [Alexander Shvets]

* 567948a 2011-03-09 | Moved hello.html to lib [Alexander Shvets]

* 6a78635 2011-03-09 | Add an author/email comment [Alexander Shvets]

* fa3c141 2011-03-09 | Added HTML header (v1) [Alexander Shvets]

* 8c32287 2011-03-09 | Added standard HTML page tags (v1-beta) [Alexander
Shvets]

* 43628f7 2011-03-09 | Added h1 tag [Alexander Shvets]

* 911e8c9 2011-03-09 | First Commit [Alexander Shvets]
```

At the moment, the repository contains all the commits from the original repo; however, they aren’t integrated into the local branches of the cloned repository.

You’ll find the commit named “Changed README in original repo” in the history. Notice that the commit includes “origin/master” and “origin/HEAD”.
Now let’s take a look at the “Updated index.html” commit. You’ll see that the local master branch points to this very commit, not the new commit we’ve just fetched.

This brings us to the conclusion that the “git fetch” command will fetch new commits from the remote repo, but won’t merge them into the local branches.

01 **Check the README**
We can show that the cloned README file has not been changed.

RUN:

```
cat README
```

RESULT:

```
$ cat README

This is the Hello World example from the git_G.
```

No changes, as you can see.

43. Merging pulled changes

Goals

- To learn to get the fetched changes into the current branch and working directory.

01 Merge the pulled changes into the local master branch

RUN:

```
git merge origin/master
```

RESULT:

```
$ git merge origin/master

Updating 6e6c76a..2faa4ea

Fast-forward

 README |    1 +

 1 files changed, 1 insertions(+), 0 deletions(-)
```

02 Check the README again

Now we should see the changes.

RUN:

```
cat README
```

RESULT:

```
$ cat README

This is the Hello World example from the git_G.

(changed in original)
```

These are the changes. Although “git fetch” does not merge the changes, we can manually merge them from the remote repo

44. Pulling and merging changes

Goals

- To learn that `git pull` command is identical to `git fetch` plus `git merge`.

Discussion

We are not going to run through the entire process of making and pulling a new change, but we want you to know that:

```
git pull
```

is actually equivalent to the following two steps:

```
git fetch

git merge origin/master
```

45. Adding a tracking branch

Goals

- To learn how to add a local branch that tracks a remote branch.

Branches that start with remotes/origin belong to the the original repository. Note that you don't have a style branch anymore, but it knows that it was in the original repository.

01 **Add a local branch tracking the remote branch.**

RUN:

```
git branch --track style origin/style

git branch -a

git hist --max-count=2
```

RESULT:

```
$ git branch --track style origin/style

Branch style set up to track remote branch style from origin.

$ git branch -a

  style

* master

  remotes/origin/HEAD -> origin/master

  remotes/origin/style

  remotes/origin/master

$ git hist --max-count=2

* 2faa4ea 2011-03-09 | Changed README in original repo (HEAD, origin/master,
origin/HEAD, master) [Alexander Shvets]

* 6e6c76a 2011-03-09 | Updated index.html (origin/style, style) [Alexander
Shvets]
```

Now we can see the style branch in the branch list and log

46. Bare repos

Goals

- To learn to create bare repos.

Bare repos (without working directories) are typically needed for sharing.

01 Creating a bare repository.

RUN:

```
cd ..

git clone --bare hello hello.git

ls hello.git
```

NOTE: We are now in the working directory.

RESULT:

```
$ git clone --bare hello hello.git

Cloning into bare repository hello.git...

done.

$ ls hello.git

HEAD

config

description

hooks

info

objects

packed-refs

refs
```

Typically repositories ending in '.git' are bare. As you can see there is no working directory in the hello.git repository. Actually it is nothing but the .git directory of a non-bare repository.

47. Adding a remote repository

Goals

- To add a bare repo as a remote to our original repo.

Let’s add the `hello.git` repository to our original repository.

RUN:

```
cd hello

git remote add shared ../hello.git
```

NOTE: We are now in the hello repo.

48. Submitting changes

Goals

- To learn how to submit changes to the remote repository.

Since a clean repository is usually shared on some network server, we need to send our changes to other repositories. Start by creating a change to be sent. Edit the README file and do a commit

FILE: ***README***

```
This is the Hello World example from the git_G.

(Changed in the original and pushed to shared)
```

RUN:

```
git checkout master

git add README

git commit -m "Added shared comment to readme"
```

Now send changes to the shared repository.

RUN:

```
git push shared master
```

The shared repository is the one receiving changes sent by us. (Remember, we added it as a remote repository in the previous lesson).

RESULT:

```
$ git push shared master

To ../hello.git

   2faa4ea..79f507c  master -> master
```

Note: We had to explicitly specify the master branch to submit changes. It can be configured automatically, but I always forget the command. For easy management of remote branches switch to «Git Remote Branch».

49. Removing common changes

Goals

- To learn how to extract changes from the common repository.

Quickly switch to the cloned repository and pull the changes just sent to the common repository.

RUN:

```
cd ../cloned_hello
```

Note: We are now in the *cloned_hello* repository.
Continue with …

RUN:

```
git remote add shared ../hello.git

git branch --track shared master

git pull shared master

cat README
```

50. Placing your git repository

Goals

- To learn how to configure a git server for sharing repos.

There are different ways to share a git repository on the network. Here's the quickest way.

01Run git server

RUN:

```
# (From the work directory)

git daemon --verbose --export-all --base-path=.
```

Now, go to your working directory in a separate terminal window.

RUN:

```
# (From the work directory)

git clone git://localhost/hello.git network_hello

cd network_hello

ls
```

You will find a copy of the hello project.

02Sending to Git Daemon

If you want to allow push to the repository Git Daemon, add `--enable=receive-pack` tag to git daemon command. Be attentive, this server does not perform authentication, so anyone can push changes to your repository.

51. Sharing repositories

Goals

- To learn to share repositories via WIFI.

Check, whether your neighbor runs a git daemon. Exchange your IP-addresses, then check whether you can extract changes from each other’s repos.