

Git and eChronos Procedures for Contributors

CONTENTS

Setup Git (for Windows)

- Downloading
- Installing
- Launching
- Built-in Help
- Standard Git Setup

Standard eChronos Procedures

- Cloning eChronos from GitHub
 - Running Clone
- Importing eChronos into Eclipse
 - needs content!!!*
- Making Changes to eChronos Files
 - Save the Changes
 - Stage the Changes
 - Commit the Changes Locally
 - Check for Changes to the Remote Repository
 - Use Log to Verify the Details
 - Pull
 - Check Status
 - Push
- Merging

Not So Standard eChronos Procedures

- Creating a Multi-Project Repository in Eclipse and Pushing to GitHub
 - Initialize the Repository in Git
 - Create a .gitignore file
 - Make the first commit
 - Upload the Repository to GitHub

++ Add these sections ++

- Needs Content2!!!*
- Needs Content3!!!

Setup Git (for Windows)

DOWNLOADING

Use one of the following links, and click on the appropriate download link in your browser.

[Git for Windows @ git-scm.com](https://git-scm.com)

[Git for Windows @ github.com](https://github.com)

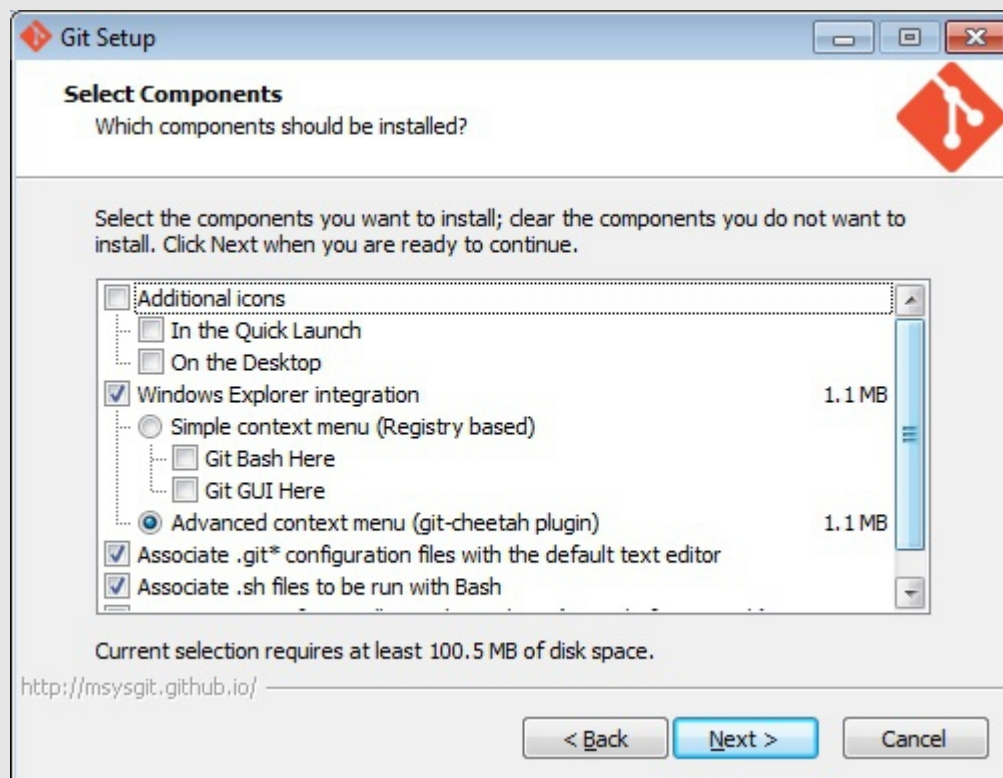
Or you can just search online for 'Git for Windows' and follow an appropriate result there.

INSTALLING

The download should have been an “EXE” file, so double-clicking will launch the installation. At this point you can choose to accept all the defaults by clicking 'Next' throughout the rest of the installation and skip to the next section. The defaults are probably fine for most users. My own installation only involved a single minor change (to add the Quick Launch button).

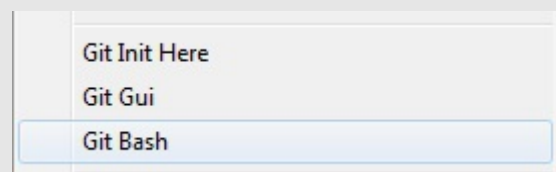
If you want to see some of the highlighted details of installation you can follow along through the rest of this section.

Select Components



I wanted the option for Quick Launch (which is unchecked in the image by mistake), but did not want the Desktop icon.

I also wanted the context menu. It shows options for Git when you right-click anywhere in Windows.

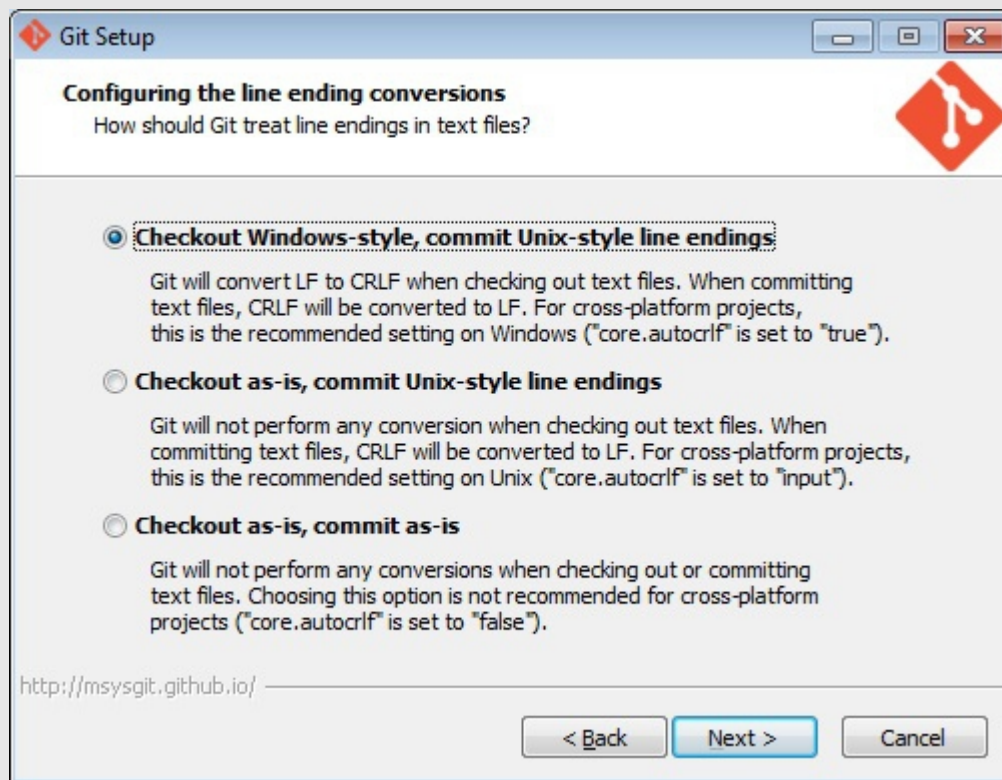


PATH Environment



The default here is good, especially if you don't know what all this means. Basically you will use the “editor” that you are currently installing to run commands instead of using the one built into Windows.

Configuring the line ending conversions



Yeah, this is gobbeldeeguk. Just accept the default. It's worked for the team so far.

LAUNCHING

After installation, both '**Git Bash**' and '**Git GUI**' will show up in your Start Menu. You want to run 'Git Bash' for the purposes of this guide, but are more than welcome to learn whatever

the heck 'Git GUI' is good for and write your own everlovin guide.

BUILT-IN HELP

If you need quick help (probably later once you've started working with Git commands) you can enter the following commands in Bash.

Warning! The following commands will launch your browser if it isn't already open!

```
$ git help git - displays 'help index'
```

```
$ git help <command> - displays help for whatever command you enter
```

STANDARD GIT SETUP

Enter your user name.

```
$ git config --global user.name "Optimus Prime"
```

Enter your email address.

```
$ git config --global user.email 1GreatLeader@cybertron.net
```

Check what you just set.

```
$ git config --list
...
user.email=1GreatLeader@cybertron.net
user.name=Optimus Prime
...
```

There are actually 3 levels of config options:

1. system (lowest precedence)
2. global
3. repository specific (highest)

As you can see, the global setting is in the middle. You can set the same information at the lowest level by calling: `$ git config --system user.name "Megatron"`, but it will be overridden by the setting at the global level. Likewise you can navigate to any specific repository on your system and call `$ git config user.name "Ultra Magnus"`, so that commits for that repository will have a different user name than any that do not set it specifically. Entering just the global settings should be sufficient for most users.

Calling `$ git config --list` shows you the settings for all 3 levels, so it may look like there is redundant information once you begin adding info to the other levels of settings.

HINT: This is a handy technique for testing what it is like for users in both situations: one that sets up the initial repository on GitHub and one that clones it. This way you can have multiple repositories on your computer with the same data, but different user names, each with the ability to make separate push/pull executions on the remote repository.

Standard eChronos Procedures

CLONING ECHRONOS FROM GITHUB

Running Clone

Typically the command looks something like this:

```
$ git clone git@github.com:Carolla/Adventurer.git eChronos
```

The 'eChronos' on the end is the name of a new folder that will be created in the current directory. The files from the repository on GitHub will be copied there. You can change this name to whatever you like. You can also create the directory manually in the file system, navigate to it in git, then run the clone command without the folder name at the end.

Note that if you don't specify a name at the end, git will use the name of the remote repository and create your local repository in a folder called 'Adventurer'.

Navigate into the new repository:

```
USER@COMPUTER ~/git  
$ cd eChronos
```

For verification you can look at the branches in the repository:

```
USER@COMPUTER ~/git/eChronos  
$ git branch -v  
* master 8cf49fb First commit
```

You can also look at the remotes that were set up automatically:

```
USER@COMPUTER ~/git/eChronos  
$ git remote -v  
origin git@github.com:Carolla/Adventurer.git (fetch)  
origin git@github.com:Carolla/Adventurer.git (push)
```

IMPORTING ECHRONOS INTO ECLIPSE

needs content!!!

Once you've cloned the remote repository to your computer, you can look for the files using your system's file browser.

[pic]

You can see that a working directory has been made from the repository files. These are the files you will actually modify in Eclipse. The repository files are contained in the .git folder. These will only be modified by Git using the Git Bash console.

There are 3 separate project folders for eChronos. They will have to be imported into Eclipse one at a time. The folders will also need some setup inside Eclipse to behave properly as projects and to cooperate with each other.

MAKING CHANGES TO ECHRONOS FILES

Save the Changes

Save your files using Eclipse (or other editor) to update the file in your Working Directory.

Stage the Changes

Using Git, Stage the changed file or files. This prepares the files to be committed permanently to your local repository. Type `git add ..` Make sure to include the dot! Git will automatically scan for changes and stage them all.

[pic]

You can also specify files. `git add ~/path/path/filename`

[pic]

Commit the Changes Locally

Commit to submit permanent changes to your local repository. Make sure you add a commit message using `-m` "commit message here".

[pic]

Otherwise you will end up in the dreaded VIM editor!!

[pic]

If this happens you can:

[type your message]

AND/OR

exit by typing "q" for "quit"

Check for Changes to the Remote Repository

You must do a fetch BEFORE checking status for this to work! Calling status without first calling fetch will not update info from the remote repository.

To see what I mean, do a `git status` first:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean
```

Now do the fetch:

```
$ git fetch origin
remote: Counting objects: 372, done.
remote: Compressing objects: 100% (230/230), done.
remote: Total 372 (delta 165), reused 264 (delta 125), receiving objects: 85% (31
Receiv
Receiving objects: 100% (372/372), 4.13 MiB | 146.00 KiB/s, done.
Resolving deltas: 100% (165/165), done.
From github.com:Carolla/Adventurer
 20cd66f..3de67cc master -> origin/master
```

Now when you call status, there's an obvious difference!

```
$ git status
On branch master
Your branch is behind 'origin/master' by 26 commits, and can be fast-forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working directory clean
```

Use Log to Verify the Details

Git log shows you local commits

```
$ git log
commit 20cd66f66fde270ac081cb3e625a374f14543e0d
Author: Dave C <mail.com>
Date: Fri Dec 5 15:45:19 2014 +1000

    hunted down some errors, corrected some file paths with AI

commit f6fbfae3ca19d592db679fd8f05b78e48acb2ea3
Merge: a31ce78 4c655c8
Author: Al Cline <mail.com>
Date: Mon Dec 1 19:46:58 2014 -0500

    Removed Data Shuttle and related classes and references
```

Git log origin shows remote commits

```
$ git log origin
commit 3de67cc65ee72bc524aa321436bc871e06fa2bf6
Merge: 4108650 1175786
Author: Al Cline <mail.com>
Date: Wed Jan 21 21:01:45 2015 -0500

    Merge branch 'master' of github.com:Carolla/Adventurer

commit 4108650d8b215f15eb0771cac99e727081d66555
Author: Al Cline <mail.com>
Date: Wed Jan 21 20:56:43 2015 -0500

    Moved some methods from BaseCiv into Utilities, and added tests for them
```

Pull

Using pull will do a combined fetch/merge. We already did a fetch, but you can use it just as easily (or more?) as merge.

```
$ git pull
Updating 20cd66f..3de67cc
Fast-forward
 .../DevDocs/Agile Practices for eChronos.docx | Bin 0 -> 27431 bytes
 Adventurer/DevDocs/AgileAssessment_Meyer.pdf | Bin 0 -> 4344134 bytes
 Adventurer/src/civ/Adventurer.java           | 64 +-
 Adventurer/src/civ/CommandParser.java        | 33 +-
 Adventurer/src/civ/HeroDisplayCiv.java       | 188 ++--
 Adventurer/src/civ/MainframeCiv.java         | 6 +-
 Adventurer/src/civ/NewHeroCiv.java           | 1141 ++++++++-----
 Adventurer/src/civ/OccupationDisplayCiv.java | 339 +++--
 Adventurer/src/civ/SkillDisplayCiv.java      | 240 ++--
```

Check Status

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean
```

Push

This will copy your latest local commit to the remote repository on GitHub.

```
$ git push
Counting objects: 10, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 855.24 KiB | 0 bytes/s, done.
Total 6 (delta 3), reused 0 (delta 0)
```

Finally! All that work, but now our changes are saved to the remote repository!

MERGING

“Merging” in Git must be done locally. This means that you can't push your changes up to GitHub when others have committed new changes that you don't yet have. To push up your changes in this case you must first do a fetch, then merge locally, and then you can push.

The Quick Version

Let's do a quick demo to see how this is done. We've got a repository set up with a remote on GitHub. It's always a good idea to do a `$ git status` so we'll start with that.

```
USER@COMPUTER ~/git/MyRepo (master)
$ git status
On branch master
Your branch is ahead of 'to_gh_myrepo/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working directory clean
```

Performing a fetch will sync us with the remote repository.

```
USER@COMPUTER ~/git/MyRepo (master)
$ git fetch
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 6 (delta 2), reused 6 (delta 2)
Unpacking objects: 100% (6/6), done.
From github.com:GH_User/2015_JAN>Hello_GitHub_Repo
 21e8a66..7ae7c72  master    -> to_gh_myrepo/master
```

Do another status if you want to see how it compares to your local branch (probably 'master').

```
USER@COMPUTER ~/git/MyRepo (master)
$ git status
On branch master
Your branch and 'to_gh_myrepo/master' have diverged,
and have 1 and 1 different commit each, respectively.
  (use "git pull" to merge the remote branch into yours)

nothing to commit, working directory clean
```

Let's continue with our merge. We can do so by providing the name of the remote branch. Any of the following will work:

- `git merge origin/master` (assuming you didn't change the default names here)
- `git merge @{upstream}`
- `git merge @{u}`

Note: After running one of these commands, it should be possible to run a merge on this branch with no arguments, simply calling `$ git merge`

```
USER@COMPUTER ~/git/MyRepo (master)
$ git merge to_gh_myrepo/master
Auto-merging HelloProject_01/src/hellopackage/HelloWorld.java
CONFLICT (content): Merge conflict in HelloProject_01/src/hellopackage/HelloWorld.java
Automatic merge failed; fix conflicts and then commit the result.
```

So now Git has created a new file containing all the data from both of the files it just tried to combine. It also placed marks in the document to help us identify which edits came from which file originally. We need to open the file and edit it to keep only the changes we want. Then we can save, stage, commit and finally push back to the server.

Here's how the file looks now:

```
HelloWorld.java x
1  package hellopackage;
2
3  public class HelloWorld {
4
5      private String name;
6
7      public HelloWorld(String name) {
8          this.name = name;
9      }
10
11     public void mrSpyriusMessage() {
12         <<<<<<< HEAD
13         String message = "Big 'ol shucks, howdy!";
14         =====
15         String message = "My new hello message!";
16         >>>>>>> gh_hello_repo/master
17         System.out.println(name + " says: " + message);
18     }
19
20     public static void main(String[] args) {
21         HelloWorld hw = new HelloWorld("Mr. Spyrius");
22         hw.mrSpyriusMessage();
23     }
24
25 }
```

What appears between '`<<<<<<< Head`' and '`=====`' is the code from our local repository. Between '`=====`' and '`>>>>>>> gh_hello_repo/master`' is from the remote. I'll clean it up and keep the message from my local commit.

Now it looks like this:

```
HelloWorld.java x
1  package hellopackage;
2
3  public class HelloWorld {
4
5      private String name;
6
7      public HelloWorld(String name) {
8          this.name = name;
9      }
10
11     public void mrSpyriusMessage() {
12         String message = "Big 'ol shucks, howdy!";
13         System.out.println(name + " says: " + message);
14     }
15
16     public static void main(String[] args) {
17         HelloWorld hw = new HelloWorld("Mr. Spyrius");
18         hw.mrSpyriusMessage();
19     }
20
21 }
```

So we finish up by saving, staging the edited file (using `$ git add .`), and checking the status

```
USER@COMPUTER ~/git/MyRepo (master)
$ git status
On branch master
Your branch and 'to_gh_myrepo/master' have diverged,
and have 1 and 1 different commit each, respectively.
(use "git pull" to merge the remote branch into yours)

All conflicts fixed but you are still merging.
(use "git commit" to conclude merge)

nothing to commit, working directory clean
```

We run the commit

```
USER@COMPUTER ~/git/MyRepo (master)
$ git commit -m"Use this hello message instead"
[master f484e2b] Use this hello message instead
```

Check status

```
USER@COMPUTER ~/git/MyRepo (master)
$ git status
On branch master
Your branch is ahead of 'to_gh_myrepo/master' by 2 commits.
(use "git push" to publish your local commits)

nothing to commit, working directory clean
```

And finally it's time to push

```
USER@COMPUTER ~/git/MyRepo (master)
$ git push
Counting objects: 17, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (7/7), 599 bytes | 0 bytes/s, done.
Total 7 (delta 3), reused 0 (delta 0)
To git@github.com:GH_User/2015_JAN_Hello_GitHub_Repo.git
7ae7c72..f484e2b  master -> master
```

The Not Remotely Quick Version

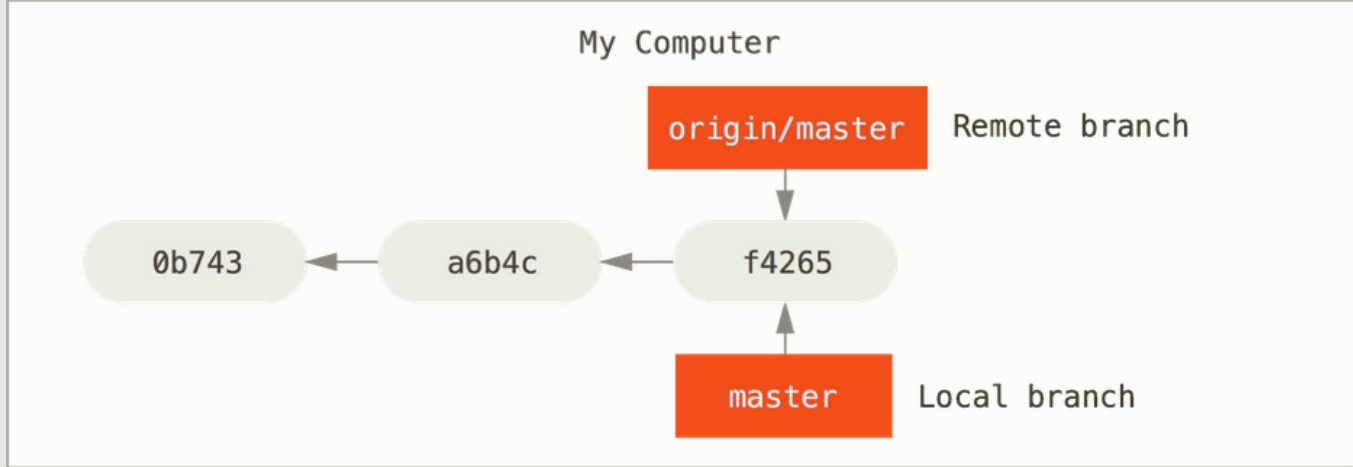
Lets say hypothetically there's a repository called "Adventurer" on GitHub with a branch called 'master'.

Suppose this 'master' branch has three commits in it. We can represent it something like this:

GitHub Adventurer/master



Let's also say that you've cloned the repository and now you have on your computer something like this:



You can see that there are now two branches on your local machine that have been set up for you, 'master' and 'origin/master'. So currently there are 3 branches and 2 repositories in play.

Generally you will treat 'origin/master' as if it's the same as 'Adventurer/master'. Though the two branches can get out of sync (when someone else pushes), when you try to do a push or a pull, the two will re-sync and prevent you from making changes if a discrepancy exists.

The best way to understand the difference between these entities is to see what happens when changes are made.

Not So Standard eChronos Procedures

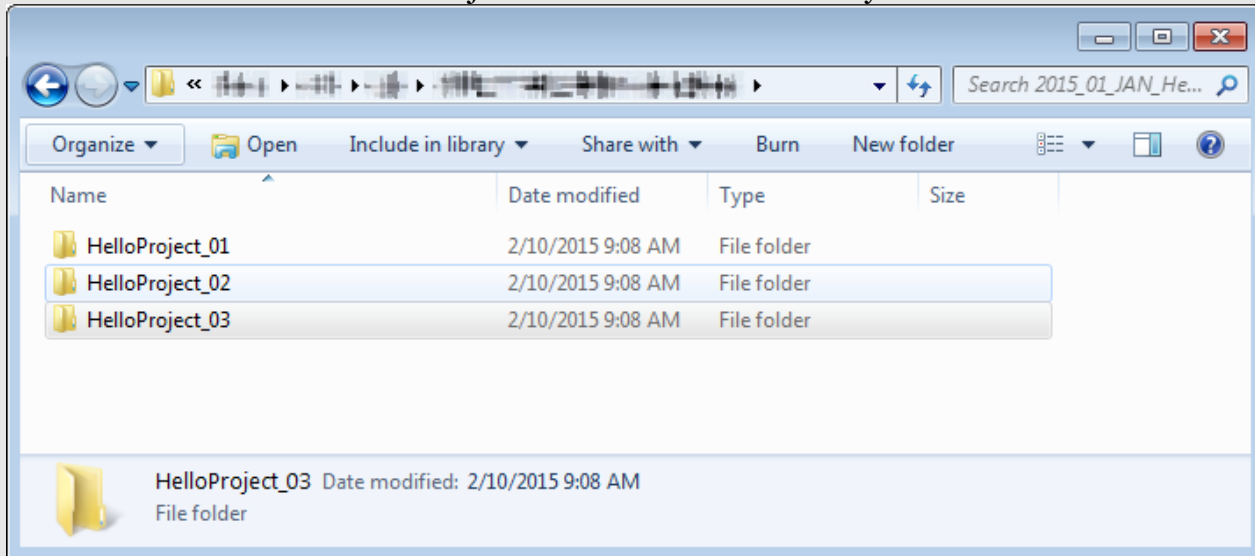
CREATING A MULTI-PROJECT REPOSITORY IN ECLIPSE AND PUSHING TO GITHUB

Navigate to the Git folder in your user directory

Create a new folder to be the root of your repository

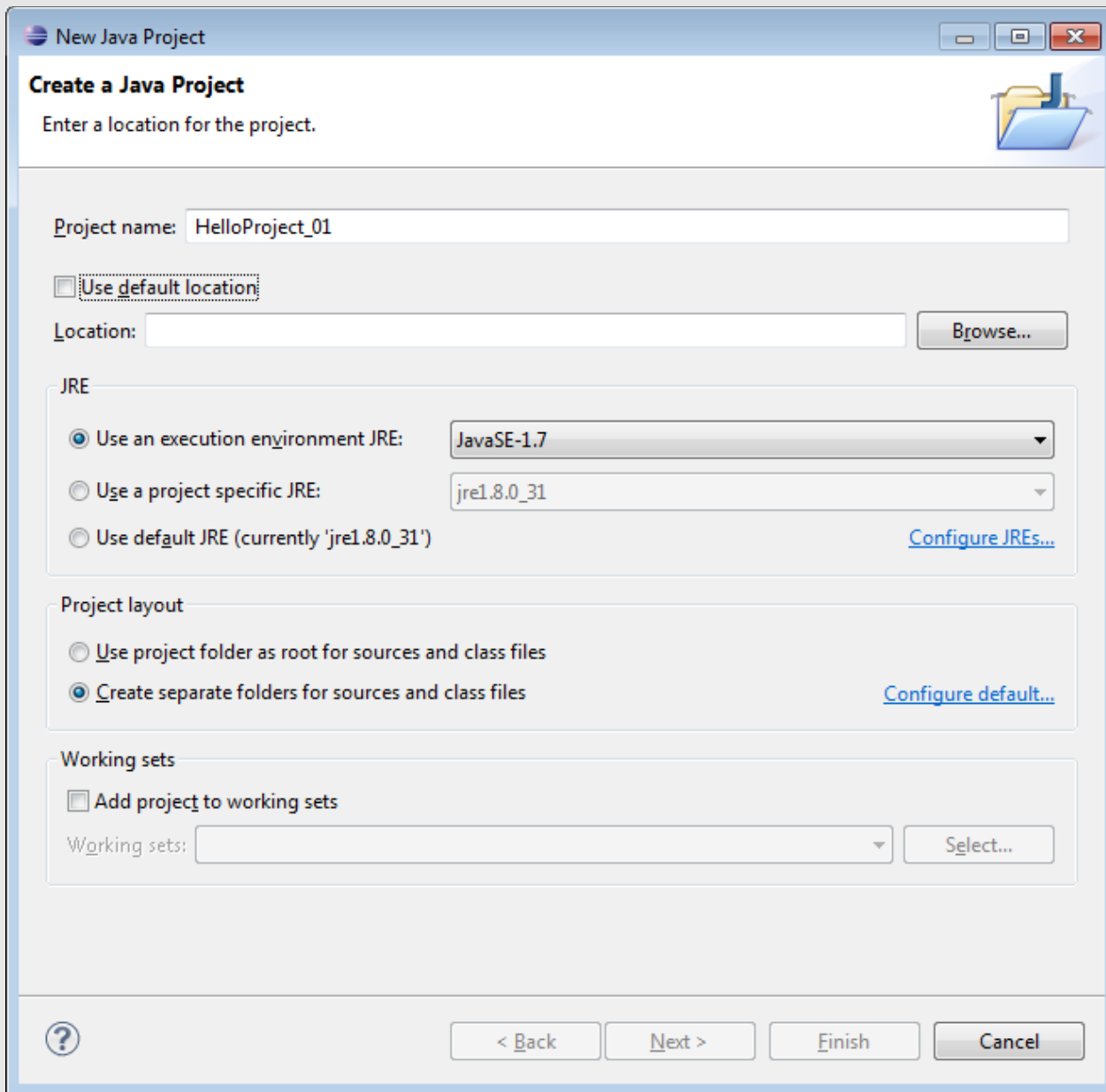
Name it appropriately (Examples: eChronos, HelloProject, My_SecretProj_Repo)

Create folders for each of the Projects inside the new directory

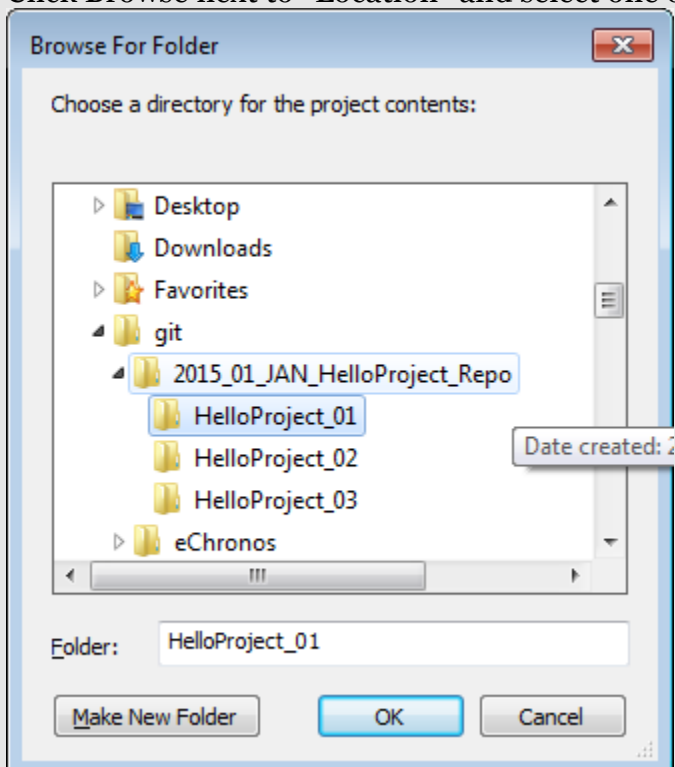


In Eclipse, create projects by right-clicking and selecting New > Java Project

Make sure to un-check the "Use default location" checkbox



Click Browse next to “Location” and select one of the project folders you just created



Initialize the Repository in Git

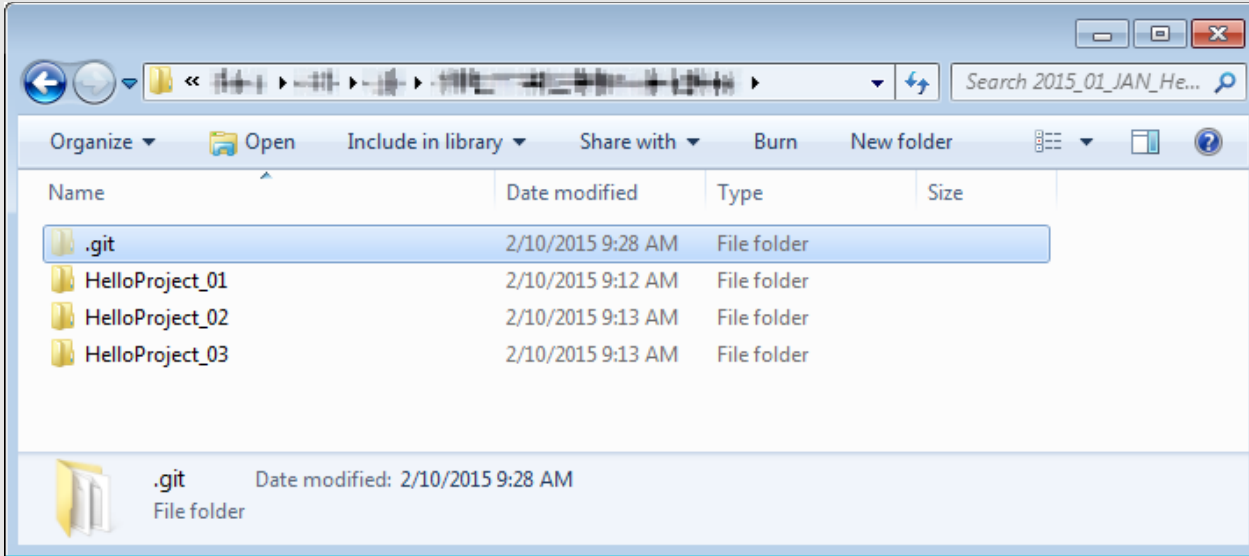
Open Bash

Navigate to the directory containing the Project Folders

Use git init to set up the empty repository

```
USER@COMPUTER ~/git/MyRepo  
$ git init
```

There will now be a ".git" folder next to your project folders (assuming you are setup to see hidden folders).

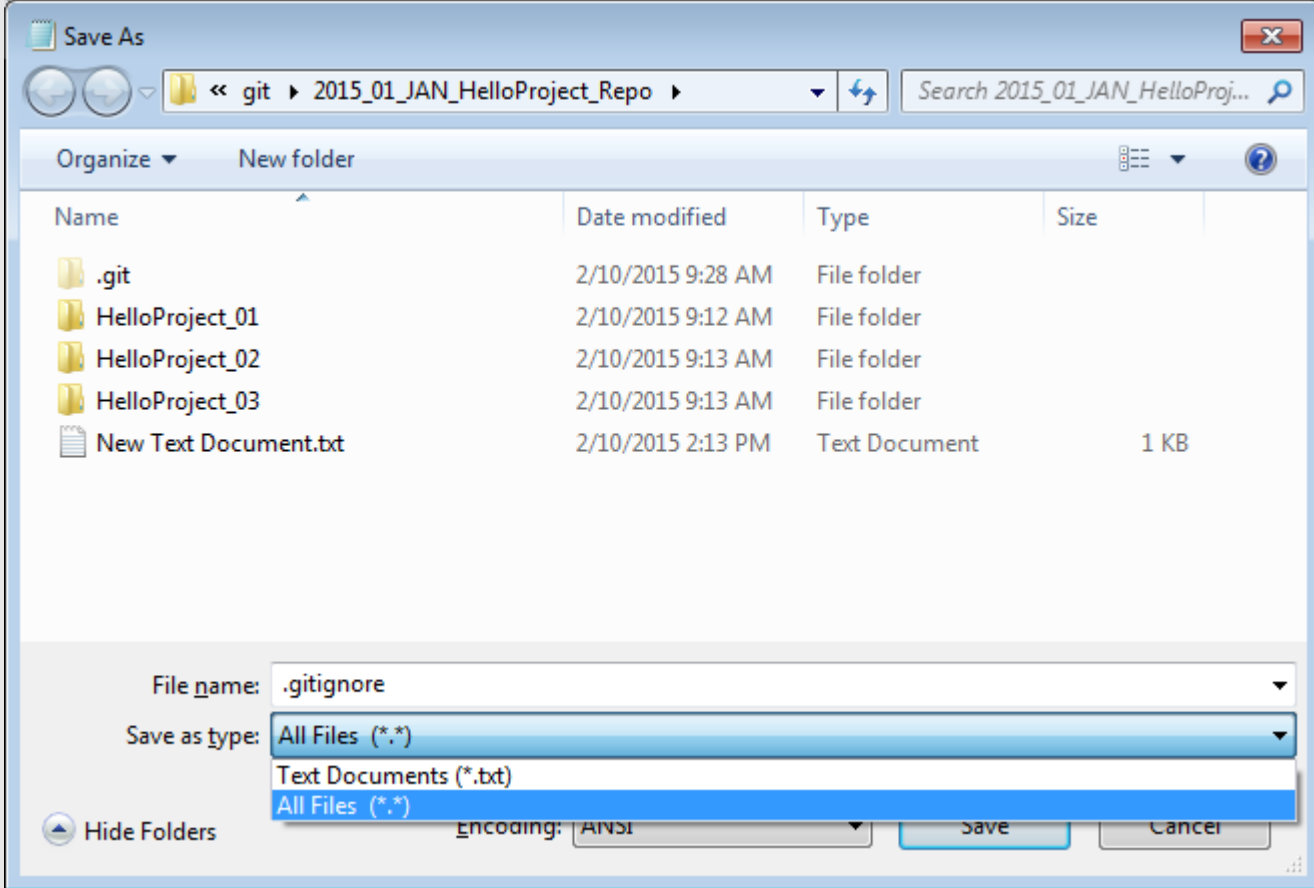


What's in the .git folder constitutes your "local repository". The project folders are your "working directory". When you do your first commit to the local repository, it will then contain an effective "backup" of the files in your working directory.

It is worth noting that a "branch" has also been automatically been created for you called "master". Until you create or clone and afterwards switch to another branch, all your work will be done on master.

Create a .gitignore file

Use a text editor to manually create a .gitignore file in this root repository directory.



Make sure to choose file type "All Files (*.*)". Otherwise Notepad will add .txt to the end.

Special characters:

- Start a line with # for a comment
- Use * as a wildcard for zero or more characters
- End with / to specify directory rather than file
- Use ! to ignore a pattern
- Use \ as escape (i.e. \!important!.txt)

Sample .gitignore:

```
# ignore all bin folders #
#####
bin/

# ignore all .class files #
#####
*.class

# ignore some eclipse files #
#####
.classpath
.project

# ignore the batman folder, but not the bruce folder inside it #
#####
batman/
!/bruce
```

Make the first commit

In this case you have to stage your files first.

```
USER@COMPUTER ~/git/MyRepo (master)
$ git add .
```

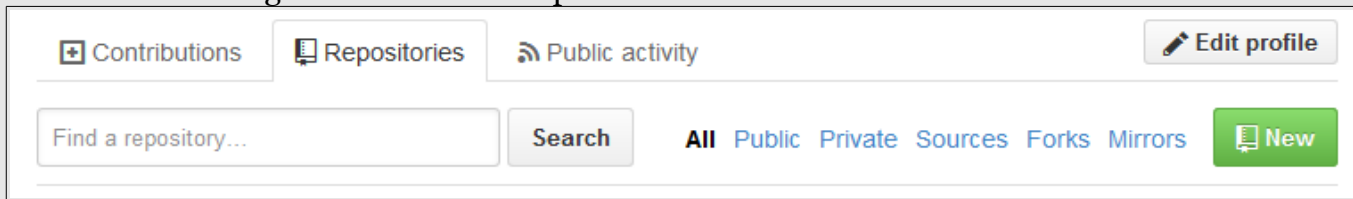
Do the commit and add a commit message (-m “message”).

```
USER@COMPUTER ~/git/MyRepo (master)
$ git commit -m "First commit"
```

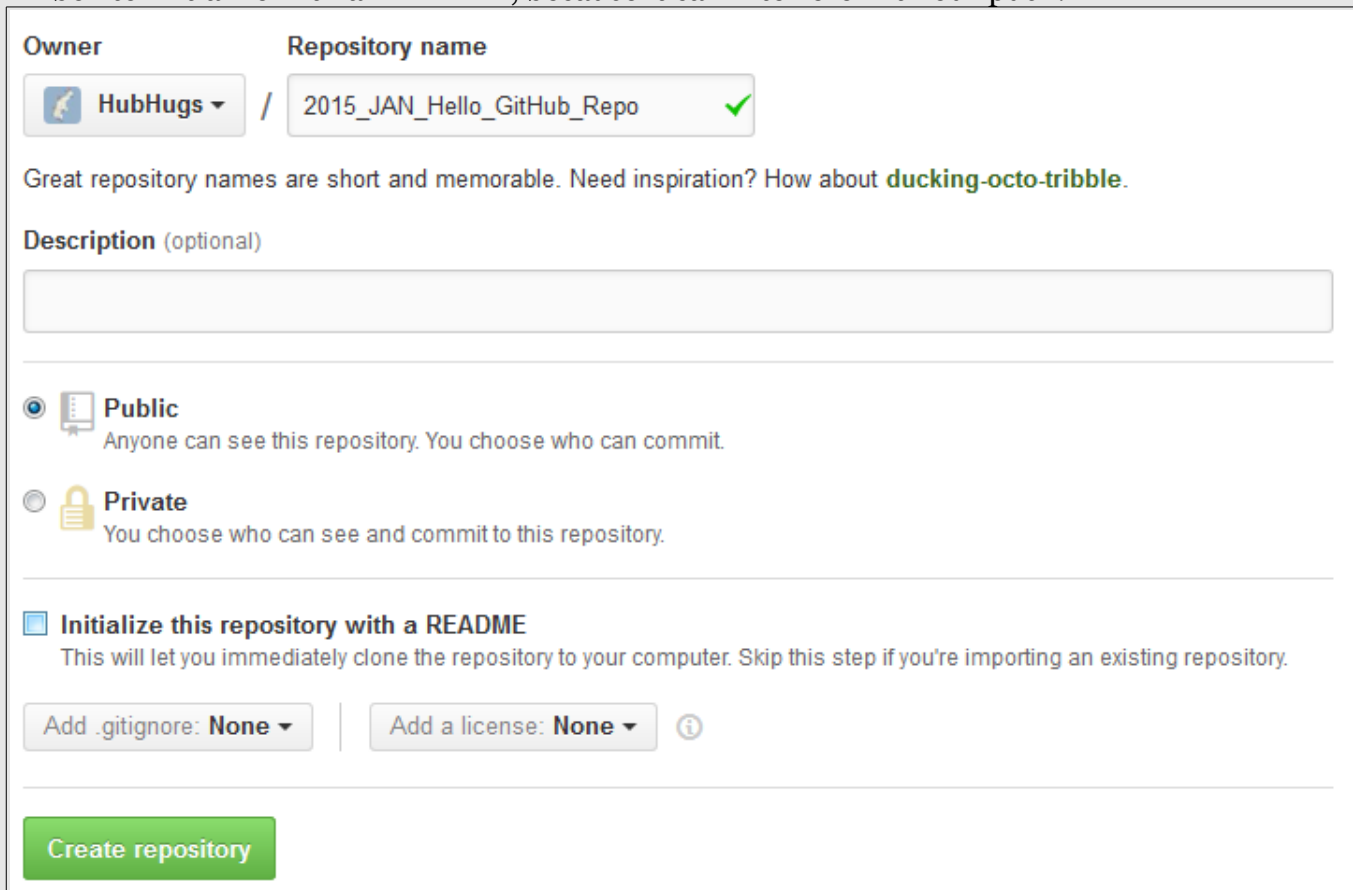
Upload the Repository to GitHub

Login to GitHub and create a new Repository there on the website. There are several ways to do this.

You can use the drop-down next to your user name, or the handy green button. Both are available from the main page after logging in. You can also navigate to the Repositories page and create it there. The image shows the latter option.

A screenshot of the GitHub web interface. At the top, there are three tabs: 'Contributions', 'Repositories' (which is selected), and 'Public activity'. To the right of these tabs is a button labeled 'Edit profile'. Below the tabs is a search bar with the placeholder text 'Find a repository...' and a 'Search' button. To the right of the search bar are several filters: 'All', 'Public', 'Private', 'Sources', 'Forks', and 'Mirrors'. Further right is a green button labeled 'New' with a plus icon.

Fill in a name for the repository and hit the *Create repository* button. Note that we're NOT checking the box to initialize with a README, because it can interfere with our push.

A screenshot of the 'Create repository' form on GitHub. The form has two main sections: 'Owner' and 'Repository name'. The 'Owner' section shows a dropdown menu with 'HubHugs' selected. The 'Repository name' section shows a text input with '2015_JAN_Hello_GitHub_Repo' and a green checkmark. Below these sections is a text input for 'Description (optional)'. Under the description is a section for 'Visibility' with two radio buttons: 'Public' (selected) and 'Private'. Below the visibility section is a checkbox labeled 'Initialize this repository with a README'. At the bottom of the form are two dropdown menus: 'Add .gitignore: None' and 'Add a license: None'. A green 'Create repository' button is at the bottom left.

Once the repository is created, GitHub suggests ways to populate it. For the most part, we will be following their second suggestion.

**Quick setup — if you've done this kind of thing before**

Set up in Desktop

or

HTTPS

SSH

git@github.com:HubHugs/2015_JAN_Hello_GitHub_Repo.git

We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo # 2015_JAN_Hello_GitHub_Repo >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin git@github.com:HubHugs/2015_JAN_Hello_GitHub_Repo.git
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin git@github.com:HubHugs/2015_JAN_Hello_GitHub_Repo.git
git push -u origin master
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

[Import code](#)

First we'll connect our local “master” branch to the remote repository. GitHub uses the nickname “origin”. I'll use “gh_hello_repo”. Copy the repository address from the top of the page.



Set up in Desktop

or

HTTPS

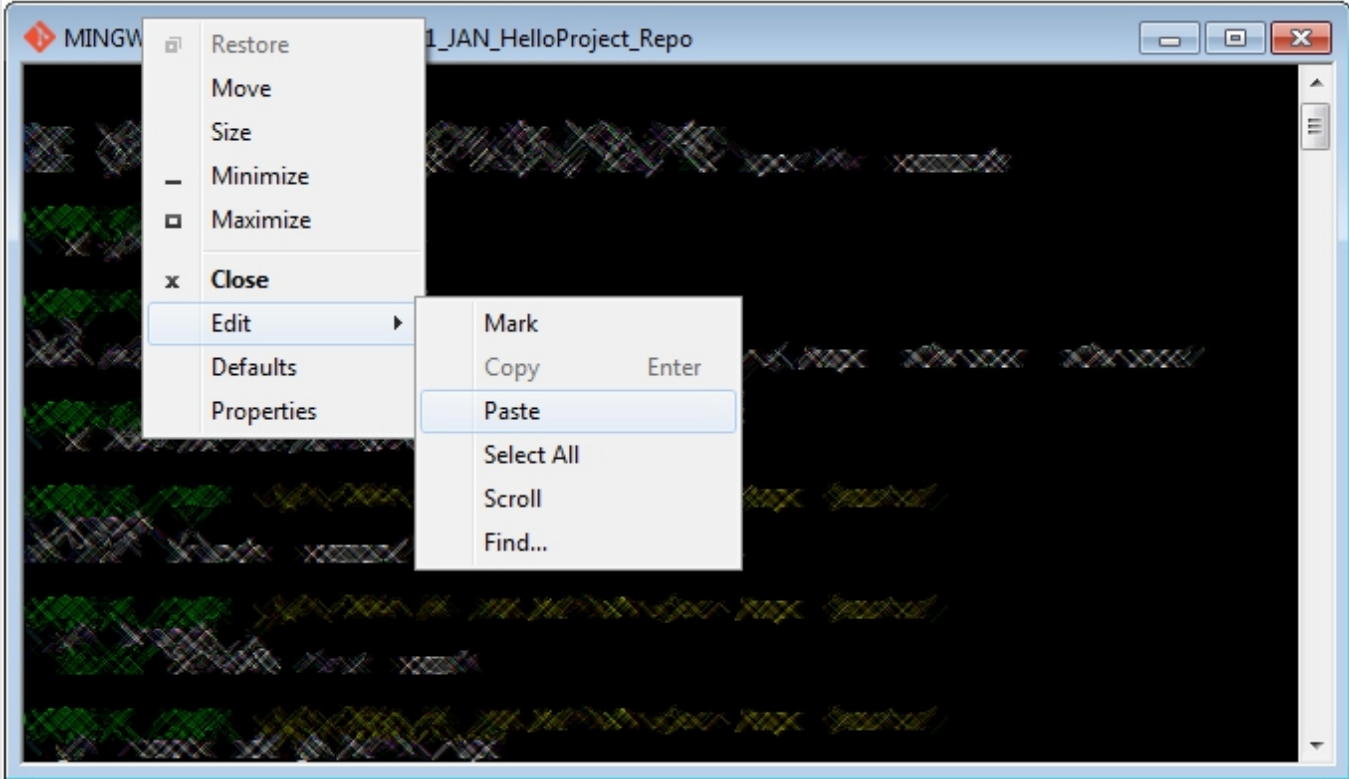
SSH

git@github.com:HubHugs/2015_JAN_Hello_GitHub_Repo.git

In Bash type in the first part of the command. Use your own short name for the repository instead of gh_hello_repo:

```
USER@COMPUTER ~/git/MyRepo (master)
$ git remote add gh_hello_repo
```

Then paste in the address from GitHub. You can do so in Bash by right-clicking the top of the window and selecting Edit > Paste.



So the final command should look something like this:

```
USER@COMPUTER ~/git/MyRepo (master)
$ git remote add gh_hello_repo git@github.com:HubHugs/2015_JAN_Hello_GitHub_Rep
o.git
```

Afterwards we can check for remote branches by using: `git remote -v`

```
USER@COMPUTER ~/git/MyRepo (master)
$ git remote -v
gh_hello_repo    git@github.com:HubHugs/2015_JAN_Hello_GitHub_Repo.git (fetch)
gh_hello_repo    git@github.com:HubHugs/2015_JAN_Hello_GitHub_Repo.git (push)
```

And now we can do our push. Make sure to use the `-u` option. This will setup a “tracking” relationship between the two branches. Tracking allows push and pull operations without the need for additional command input.

```
USER@COMPUTER ~/git/MyRepo (master)
$ git push -u gh_hello_repo master
```

So we are telling the push command to use the “gh_hello_repo” repository and the “master” branch on that repository; not to be confused with the “master” branch of our local repository. The format is:
`push -u [remote repository] [remote branch]`

Below is the command with the full output:

```
USER@COMPUTER ~/git/MyRepo (master)
$ git push -u gh_hello_repo master
Counting objects: 15, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
writing objects: 100% (15/15), 958 bytes | 0 bytes/s, done.
Total 15 (delta 2), reused 0 (delta 0)
To git@github.com:HubHugs/2015_JAN_Hello_GitHub_Repo.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from gh_hello_repo.
```


Now if we refresh the repository page in our browser it shows the updated info:

The screenshot shows the GitHub repository page for '2015_JAN_Hello_GitHub_Repo'. At the top, there are buttons for 'Unwatch', 'Star' (0), and 'Fork' (0). Below this is a 'Description' section with a text input field and a 'Website' section with another text input field. There are 'Save' and 'Cancel' buttons. Below the inputs, it shows '1 commit', '1 branch', '0 releases', and '0 contributors'. A 'branch: master' dropdown is visible. The main content area shows a list of files and their commit history:

File	Commit	Time
First commit		
Dave C authored 2 days ago	latest commit 8cf49fbfa0	
HelloProject_01/src/hellopackage	First commit	2 days ago
HelloProject_02/src/hellopackage	First commit	2 days ago
HelloProject_03/src/hellopackage	First commit	2 days ago
.gitignore	First commit	2 days ago

At the bottom, there is a button 'Add a README' and a message: 'Help people interested in this repository understand your project by adding a README!'. On the right side, there are links for 'Code', 'Issues' (0), 'Pull Requests' (0), 'Wiki', 'Pulse', 'Graphs', and 'Settings'. There is also a section for 'HTTPS clone URL' with the URL 'https://github.com/HubHug' and buttons for 'Clone in Desktop' and 'Download ZIP'.

As a final note it is probably a good idea to go into the settings for your repository on GitHub and in the “Features” section set it to “Restrict editing to collaborators only” unless you want the general public to be able to make changes as they please.

The screenshot shows the 'Features' section of the GitHub repository settings. It contains three sections:

- Wikis**: GitHub Wikis is a simple way to let others contribute content. Any GitHub user can create and edit pages to use for documentation, examples, support, or anything you wish.
- Restrict editing to collaborators only** (checked with a green checkmark): Public wikis will still be readable by everyone.
- Issues**: GitHub Issues adds lightweight issue tracking tightly integrated with your repository. Add issues to milestones, label issues, and close & reference issues from commit messages.

++ Add these sections ++

Merging two files that have conflicts.

Deleting Files

Un-deleting Files

Refreshing a previously deleting file, that is, restoring a backed up version.

Contents_NeedsContent2

Contents_NeedsContent3

Needs Content2!!!

Needs Content3!!!