

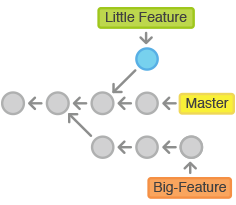
<http://www.atlassian.com/git/tutorial/git-branches#!branch>

***The git branch Command***

A branch represents an independent line of development. You can think of them as a way to request a brand new working directory, staging area, and project history. New commits are recorded in the history for the current branch, which results in a fork in the history of the project.

***Discussion***

In Git, branches are a part of your everyday development process. When you want to add a new feature or fix a bug—no matter how big or how small—you spawn a new branch to encapsulate your changes. This makes sure that unstable code is never committed to the main code base, and it gives you the chance to clean up your feature’s history before merging it into the main branch.



For example, the above diagrams visualizes a repository with two isolated lines of development, one for a little feature, and one for a longer-running feature. By developing them in branches, it’s not only possible to work on both of them in parallel, but it also keeps the main master branch free from questionable code.

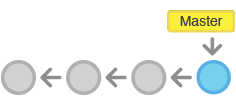
**Branch Tips**

The implementation behind Git branches is much more lightweight than SVN’s model. Instead of copying files from directory to directory, Git stores a branch as a reference to a commit. In this sense, a branch represents the tip of a series of commits—it's not a container for commits. The history for a branch is extrapolated through the commit relationships.

This has a dramatic impact on Git's merging model. Whereas merges in SVN are done on a file-basis, Git lets you work on the more abstract level of commits. You can actually see merges in the project history as a joining of two independent commit histories.

***Creating Branches***

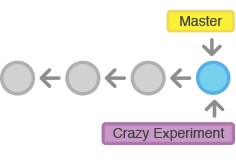
*It's important to understand that branches are just pointers to commits. When you create a branch, all Git needs to do is create a new pointer—it doesn’t change the repository in any other way. So, if you start with a repository that looks like this:*



Then, you create a branch using the following command:

**git branch crazy-experiment**

The repository history remains unchanged. All you get is a new pointer to the current commit:



Note that this only *creates* the new branch. To start adding commits to it, you need to select it with git checkout, and then use the standard git add and git commit commands. Please see the git checkout section of this module for more information.

#### Deleting Branches

Once you’ve finished working on a branch and have merged it into the main code base, you’re free to delete the branch without losing any history:

**git branch -d crazy-experiment**

However, if the branch hasn’t been merged, the above command will output an error message:

**error: The branch 'crazy-experiment' is not fully merged.  
If you are sure you want to delete it, run 'git branch -D crazy-experiment'.**

This protects you from losing your reference to those commits, which means you would effectively lose access to that entire line of development. If you *really* want to delete the branch (e.g., it’s a failed experiment), you can use the capital -D flag:

**git branch -D crazy-experiment**

This deletes the branch regardless of its status and without warnings, so use it judiciously.

## The git checkout Command

The git checkout command serves three distinct functions: checking out files, checking out commits, and checking out branches. In this module, we’re only concerned with the first two configurations.

Checking out a commit makes the entire working directory match that commit. This can be used to view an old state of your project without altering your current state in any way. Checking out a file lets you see an old version of that particular file, leaving the rest of your working directory untouched.

### Usage

**git checkout master**

Return to the master branch. Branches are covered in depth in the next module, but for now, you can just think of this as a way to get back to the “current” state of the project.

**git checkout <commit> <file>**

Check out a previous version of a file. This turns the <file> that resides in the working directory into an exact copy of the one from <commit> and adds it to the staging area.

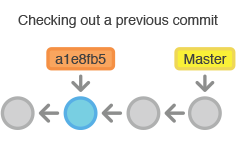
**git checkout <commit>**

Update all files in the working directory to match the specified commit. You can use either a commit hash or a tag as the <commit> argument. This will put you in a detached HEAD state.

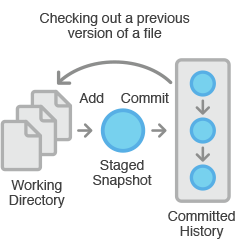
### Discussion

The whole idea behind any version control system is to store “safe” copies of a project so that you never have to worry about irreparably breaking your code base. Once you’ve built up a project history, git checkout is an easy way to “load” any of these saved snapshots onto your development machine.

Checking out an old commit is a read-only operation. It’s impossible to harm your repository while viewing an old revision. The “current” state of your project remains untouched in the master branch (see the [Branches Module](http://www.atlassian.com/git/tutorial/git-branches) for details). During the normal course of development, the HEAD usually points to master or some other local branch, but when you check out a previous commit, HEAD no longer points to a branch—it points directly to a commit. This is called a “detached HEAD” state, and it can be visualized as the following:



On the other hand, checking out an old file does affect the current state of your repository. You can re-commit the old version in a new snapshot as you would any other file. So, in effect, this usage of git checkout serves as a way to revert back to an old version of an individual file.



### Example

#### Viewing an Old Revision

This example assumes that you’ve started developing a crazy experiment, but you’re not sure if you want to keep it or not. To help you decide, you want to take a look at the state of the project before you started your experiment. First, you’ll need to find the ID of the revision you want to see.

**git log --oneline**

Let’s say your project history looks something like the following:

*b7119f2 Continue doing crazy things  
872fa7e Try something crazy  
a1e8fb5 Make some important changes to hello.py  
435b61d Create hello.py  
9773e52 Initial import*

You can use git checkout to view the “Make some import changes to hello.py” commit as follows:

**git checkout a1e8fb5**

This makes your working directory match the exact state of the a1e8fb5 commit. You can look at files, compile the project, run tests, and even edit files without worrying about losing the current state of the project. *Nothing* you do in here will be saved in your repository. To continue developing, you need to get back to the “current” state of your project:

**git checkout master**

This assumes that you're developing on the default master branch, which will be thoroughly discussed in the Branches Module.

Once you’re back in the master branch, you can use either git revert or git reset to undo any undesired changes.

#### Checking Out a File

If you’re only interested in a single file, you can also use git checkout to fetch an old version of it. For example, if you only wanted to see the hello.py file from the old commit, you could use the following command:

**git checkout a1e8fb5 hello.py**

Remember, unlike checking out a commit, this *does* affect the current state of your project. The old file revision will show up as a “Change to be committed,” giving you the opportunity to revert back to the previous version of the file. If you decide you don’t want to keep the old version, you can check out the most recent version with the following:

**git checkout HEAD hello.py**

**The git merge Command**

Merging is Git's way of putting a forked history back together again. The git merge command lets you take the independent lines of development created by [git branch](http://www.atlassian.com/git/tutorial/git-branches#%21branch) and integrate them into a single branch.

Note that all of the commands presented below merge *into* the current branch. The current branch will be updated to reflect the merge, but the target branch will be completely unaffected. Again, this means that git merge is often used in conjunction with [git checkout](http://www.atlassian.com/git/tutorial/git-branches#%21checkout) for selecting the current branch and git branch -d for deleting the obsolete target branch.

**Usage**

**git merge <branch>**

Merge the specified branch into the current branch. Git will determine the merge algorithm automatically (discussed below).

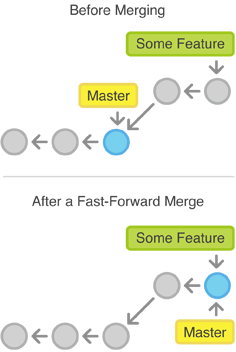
**git merge --no-ff <branch>**

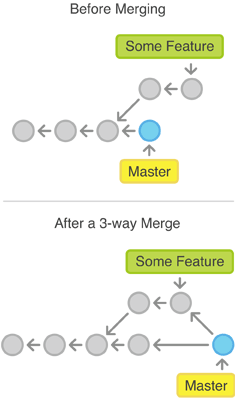
Merge the specified branch into the current branch, but *always* generate a merge commit (even if it was a fast-forward merge). This is useful for documenting all merges that occur in your repository.

**Discussion**

Once you’ve finished developing a feature in an isolated branch, it's important to be able to get it back into the main code base. Depending on the structure of your repository, Git has several distinct algorithms to accomplish this: a fast-forward merge or a 3-way merge.

A **fast-forward merge** can occur when there is a linear path from the current branch tip to the target branch. Instead of “actually” merging the branches, all Git has to do to integrate the histories is move (i.e., “fast forward”) the current branch tip up to the target branch tip. This effectively combines the histories, since all of the commits reachable from the target branch are now available through the current one. For example, a fast forward merge of some-feature into master would look something like the following:



However, a fast-forward merge is not possible if the branches have diverged. When there is not a linear path to the target branch, Git has no choice but to combine them via a **3-way merge**. 3-way merges use a dedicated commit to tie together the two histories. The nomenclature comes from the fact that Git uses *three* commits to generate the merge commit: the two branch tips and their common ancestor. 

While you can use either of these merge strategies, many developers like to use fast-forward merges (facilitated through [rebasing](http://www.atlassian.com/wac/landing/git/training/rewriting-git-history#%21rebase)) for small features or bug fixes, while reserving 3-way merges for the integration of longer-running features. In the latter case, the resulting merge commit serves as a symbolic joining of the two branches.

**Resolving Conflicts**

If the two branches you're trying to merge both changed the same part of the same file, Git won't be able to figure out which version to use. When such a situation occurs, it stops right before the merge commit so that you can resolve the conflicts manually.

The great part of Git's merging process is that it uses the familiar edit/stage/commit workflow to resolve merge conflicts. When you encounter a merge conflict, running the [git status](http://www.atlassian.com/wac/landing/git/training/git-basics#%21status) command shows you which files need to be resolved. For example, if both branches modified the same section of hello.py, you would see something like the following:

# On branch master  
# Unmerged paths:  
# (use "git add/rm ..." as appropriate to mark resolution)  
#  
# both modified: hello.py  
#

Then, you can go in and fix up the merge to your liking. When you're ready to finish the merge, all you have to do is run git add on the conflicted file(s) to tell Git they're resolved. Then, you run a normal git commit to generate the merge commit. It’s the exact same process as committing an ordinary snapshot, which means it’s easy for normal developers to manage their own merges.

Note that merge conflicts will only occur in the event of a 3-way merge. It’s not possible to have conflicting changes in a fast-forward merge.

**Example**

**Fast-Forward Merge**

Our first example demonstrates a fast-forward merge. The code below creates a new branch, adds two commits to it, then integrates it into the main line with a fast-forward merge.

# Start a new feature  
git checkout -b new-feature master  
  
# Edit some files  
git add   
git commit -m "Start a feature"  
  
# Edit some files  
git add   
git commit -m "Finish a feature"  
  
# Merge in the new-feature branch  
git checkout master  
git merge new-feature  
git branch -d new-feature

This is a common workflow for short-lived topic branches that are used more as an isolated development than an organizational tool for longer-running features.

Also note that Git should not complain about the git branch -d, since new-feature is now accessible from the master branch.

**3-Way Merge**

The next example is very similar, but requires a 3-way merge because master progresses while the feature is in-progress. This is a common scenario for large features or when several developers are working on a project simultaneously.

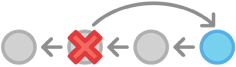
# Start a new feature  
git checkout -b new-feature master  
  
# Edit some files  
git add   
git commit -m "Start a feature"  
  
# Edit some files  
git add   
git commit -m "Finish a feature"  
  
# Develop the master branch  
git checkout master  
  
# Edit some files  
git add   
git commit -m "Make some super-stable changes to master"  
  
# Merge in the new-feature branch  
git merge new-feature  
git branch -d new-feature

Note that it’s impossible for Git to perform a fast-forward merge, as there is no way to move master up to new-feature without backtracking.

For most workflows, new-feature would be a much larger feature that took a long time to develop, which would be why new commits would appear on master in the meantime. If your feature branch was actually as small as the one in the above example, you would probably be better off rebasing it onto master and doing a fast-forward merge. This prevents superfluous merge commits from cluttering up the project history.

## The git revert Command

The git revert command undoes a committed snapshot. But, instead of removing the commit from the project history, it figures out how to undo the changes introduced by the commit and appends a *new* commit with the resulting content. This prevents Git from losing history, which is important for the integrity of your revision history and for reliable collaboration.



### Usage

**git revert <commit>**

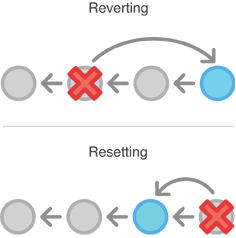
Generate a new commit that undoes all of the changes introduced in <commit>, then apply it to the current branch.

### Discussion

Reverting should be used when you want to remove an entire commit from your project history. This can be useful, for example, if you’re tracking down a bug and find that it was introduced by a single commit. Instead of manually going in, fixing it, and committing a new snapshot, you can use git revert to automatically do all of this for you.

#### Reverting vs. Resetting

It's important to understand that git revert undoes a single commit—it does not "revert" back to the previous state of a project by removing all subsequent commits. In Git, this is actually called a [reset](http://www.atlassian.com/git/tutorial/undoing-changes#%21reset), not a revert.



Reverting has two important advantages over resetting. First, it doesn’t change the project history, which makes it a “safe” operation for commits that have already been published to a shared repository. For details about why altering shared history is dangerous, please see the [git reset](http://www.atlassian.com/git/tutorial/undoing-changes#%21reset) page.

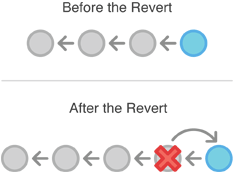
Second, git revert is able to target an individual commit at an arbitrary point in the history, whereas git reset can only work backwards from the current commit. For example, if you wanted to undo an old commit with git reset, you would have to remove all of the commits that occurred after the target commit, remove it, then re-commit all of the subsequent commits. Needless to say, this is not an elegant undo solution.

### Example

The following example is a simple demonstration of git revert. It commits a snapshot, then immediately undoes it with a revert.

# Edit some tracked files  
  
# Commit a snapshot  
git commit -m "Make some changes that will be undone"  
  
# Revert the commit we just created  
git revert HEAD

This can be visualized as the following:



Note that the 4th commit is still in the project history after the revert. Instead of deleting it, git revert added a new commit to undo its changes. As a result, the 3rd and 5th commits represent the exact same code base, and the 4th commit is still in our history just in case we want to go back to it down the road.

## The git reset Command

If git revert is a “safe” way to undo changes, you can think of git reset as the *dangerous* method. When you undo with git reset(and the commits are no longer referenced by any ref or the reflog), there is no way to retrieve the original copy—it is a *permanent* undo. Care must be taken when using this tool, as it’s one of the only Git commands that has the potential to lose your work.

Like [git checkout](http://www.atlassian.com/git/tutorial/undoing-changes#%21checkout), git reset is a versatile command with many configurations. It can be used to remove committed snapshots, although it’s more often used to undo changes in the staging area and the working directory. In either case, it should only be used to undo *local* changes—you should never reset snapshots that have been shared with other developers.

### Usage

**git reset <file>**

Remove the specified file from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes.

**git reset**

Reset the staging area to match the most recent commit, but leave the working directory unchanged. This unstages *all* files without overwriting any changes, giving you the opportunity to re-build the staged snapshot from scratch.

**git reset --hard**

Reset the staging area and the working directory to match the most recent commit. In addition to unstaging changes, the --hard flag tells Git to overwrite all changes in the working directory, too. Put another way: this *obliterates* all uncommitted changes, so make sure you really want to throw away your local developments before using it.

**git reset <commit>**

Move the current branch tip backward to <commit>, reset the staging area to match, but leave the working directory alone. All changes made since <commit> will reside in the working directory, which lets you re-commit the project history using cleaner, more atomic snapshots.

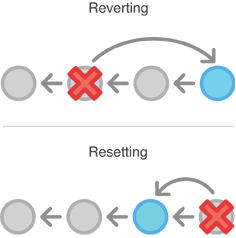
**git reset --hard <commit>**

Move the current branch tip backward to <commit> and reset both the staging area and the working directory to match. This obliterates not only the uncommitted changes, but all commits after <commit>, as well.

### Discussion

All of the above invocations are used to remove changes from a repository. Without the --hard flag, git reset is a way to clean up a repository by unstaging changes or uncommitting a series of snapshots and re-building them from scratch. The --hard flag comes in handy when an experiment has gone horribly wrong and you need a clean slate to work with.

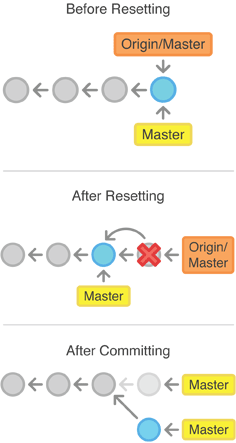
Whereas reverting is designed to safely undo a *public* commit, git reset is designed to undo *local* changes. Because of their distinct goals, the two commands are implemented differently: resetting completely removes a changeset, whereas [reverting](http://www.atlassian.com/git/tutorial/undoing-changes#%21revert) maintains the original changeset and uses a new commit to apply the undo.



#### Don’t Reset Public History

You should never use git reset <commit> when any snapshots after <commit> have been pushed to a public repository. After publishing a commit, you have to assume that other developers are reliant upon it.

Removing a commit that other team members have continued developing poses serious problems for collaboration. When they try to sync up with your repository, it will look like a chunk of the project history abruptly disappeared. The sequence below demonstrates what happens when you try to reset a public commit. The origin/master branch is the central repository’s version of your local master branch.



As soon as you add new commits after the reset, Git will think that your local history has diverged from origin/master, and the merge commit required to synchronize your repositories is likely to confuse and frustrate your team.

The point is, make sure that you’re using git reset <commit> on a local experiment that went wrong—not on published changes. If you need to fix a public commit, the git revert command was designed specifically for this purpose.

### Examples

#### Unstaging a File

The git reset command is frequently encountered while preparing the staged snapshot. The next example assumes you have two files called hello.py and main.py that you’ve already added to the repository.

# Edit both hello.py and main.py  
  
# Stage everything in the current directory  
git add .  
  
# Realize that the changes in hello.py and main.py  
# should be committed in different snapshots  
  
# Unstage main.py  
git reset main.py  
  
# Commit only hello.py  
git commit -m "Make some changes to hello.py"  
  
# Commit main.py in a separate snapshot  
git add main.py  
git commit -m "Edit main.py"

As you can see, git reset helps you keep your commits highly-focused by letting you unstage changes that aren’t related to the next commit.

#### Removing Local Commits

The next example shows a more advanced use case. It demonstrates what happens when you’ve been working on a new experiment for a while, but decide to completely throw it away after committing a few snapshots.

# Create a new file called `foo.py` and add some code to it  
  
# Commit it to the project history  
git add foo.py  
git commit -m "Start developing a crazy feature"  
  
# Edit `foo.py` again and change some other tracked files, too  
  
# Commit another snapshot  
git commit -a -m "Continue my crazy feature"  
  
# Decide to scrap the feature and remove the associated commits  
git reset --hard HEAD~2

The git reset HEAD~2 command moves the current branch backward by two commits, effectively removing the two snapshots we just created from the project history. Remember that this kind of reset should only be used on *unpublished* commits. Never perform the above operation if you’ve already pushed your commits to a shared repository.

## The git clean Command

The git clean command removes untracked files from your working directory. This is really more of a convenience command, since it’s trivial to see which files are untracked with git status and remove them manually. Like an ordinary rm command, git clean is *not* undoable, so make sure you really want to delete the untracked files before you run it.

The git clean command is often executed in conjunction with git reset --hard. Remember that resetting only affects tracked files, so a separate command is required for cleaning up untracked ones. Combined, these two commands let you return the working directory to the exact state of a particular commit.

### Usage

**git clean -n**

Perform a “dry run” of git clean. This will show you which files are going to be removed without actually doing it.

**git clean -f**

Remove untracked files from the current directory. The -f (force) flag is required unless the clean.requireForce configuration option is set to false (it's true by default). This will *not* remove untracked folders or files specified by .gitignore.

**git clean -f <path>**

Remove untracked files, but limit the operation to the specified path.

**git clean -df**

Remove untracked files *and* untracked directories from the current directory.

**git clean -xf**

Remove untracked files from the current directory as well as any files that Git usually ignores.

### Discussion

The git reset --hard and git clean -f commands are your best friends after you’ve made some embarrassing developments in your local repository and want to burn the evidence. Running both of them will make your working directory match the most recent commit, giving you a clean slate to work with.

The git clean command can also be useful for cleaning up the working directory after a build. For example, it can easily remove the .o and .exe binaries generated by a C compiler. This is occasionally a necessary step before packaging a project for release. The -x option is particularly convenient for this purpose.

Keep in mind that, along with git reset, git clean is one of the only Git commands that has the potential to permanently delete commits, so be careful with it. In fact, it’s so easy to lose important additions that the Git maintainers *require* the -f flag for even the most basic operations. This prevents you from accidentally deleting everything with a naive git clean call.

### Example

The following example obliterates all changes in the working directory, including new files that have been added. It assumes you’ve already committed a few snapshots and are experimenting with some new developments.

# Edit some existing files  
# Add some new files  
# Realize you have no idea what you're doing  
  
# Undo changes in tracked files  
git reset --hard  
  
# Remove untracked files  
git clean -df

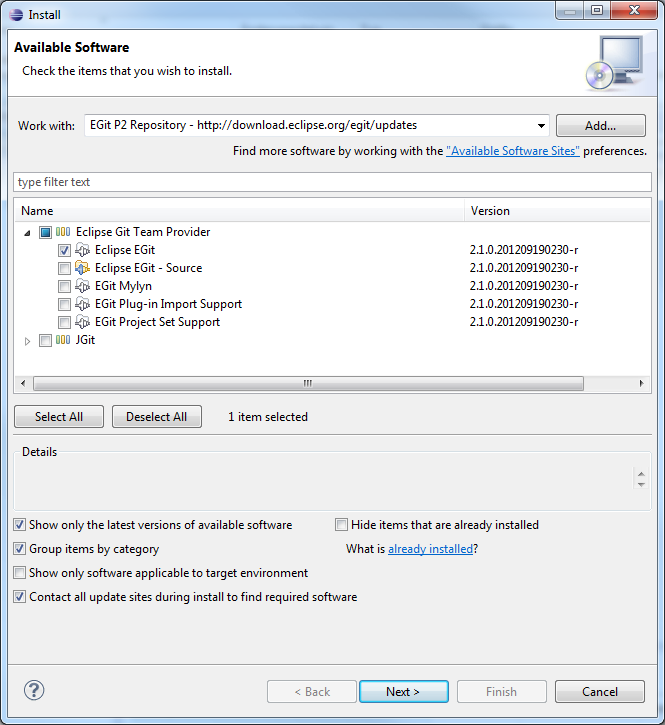
After running this reset/clean sequence, the working directory and the staging area will look exactly like the most recent commit, and git status will report a clean working directory. You're now ready to begin again.

Note that, unlike the second example in [git reset](http://www.atlassian.com/git/tutorial/undoing-changes#%21reset), the new files were *not*added to the repository. As a result, they could not be affected by git reset --hard, and git clean was required to delete them.

**Tutorial taken from http://eclipsesource.com**

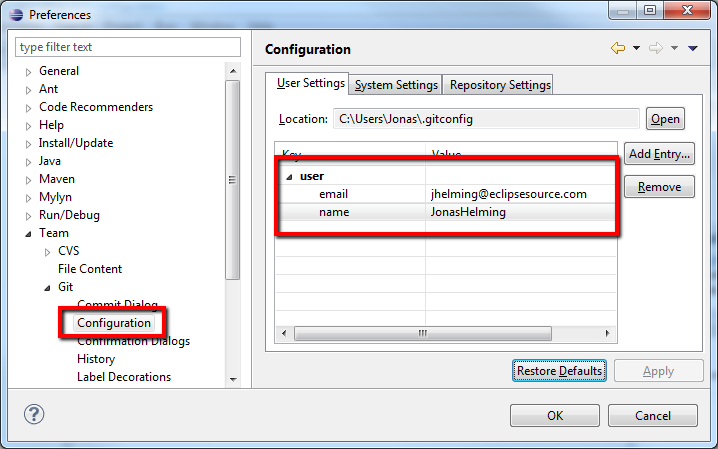
**Installing EGit in Eclipse**

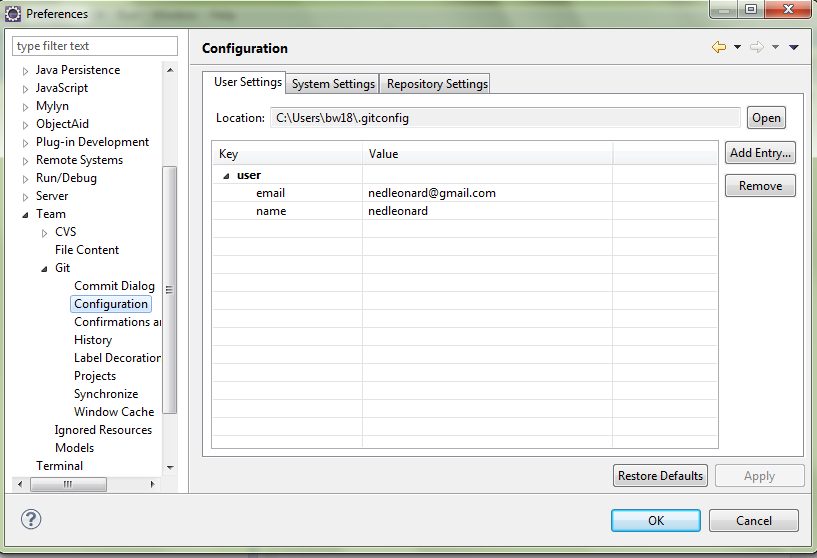
EGit is already included in the Eclipse Juno Release, so you do not need to install it. If you use an older version of Eclipse, open the Eclipse Wizard to install new software *Help => Install New Software*. Insert <http://download.eclipse.org/egit/updates> after *Work with:* and hit Return. Select Eclipse EGit as a child from Eclipse Team Git Provider. You don’t have to install any other plugins. Click *Next* and confirm your selection in the following window by pressing *Next* again. Finally, accept the terms of use and the license agreement and click *Finish* to start the installation. After the installation has finished restart Eclipse.



**EGit Configuration**

Every commit in EGit will include the user’s name and his email-address. These attributes can be set in the Preferences-window *Window => Preferences*. Navigate to *Team => Git => Configuration* and hit the *New Entry…* Button. Enter *user.name* as *Key* and your name as *Value* and confirm. Repeat this procedure with *user.email* and your email address and click *OK* in the Preferences window. The username and email should be the same you use for your Git account, ie. your GitHub account.

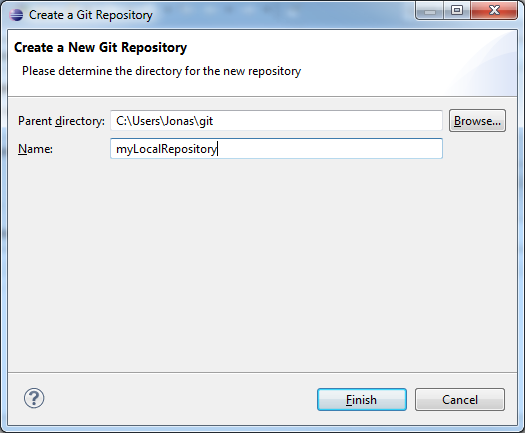




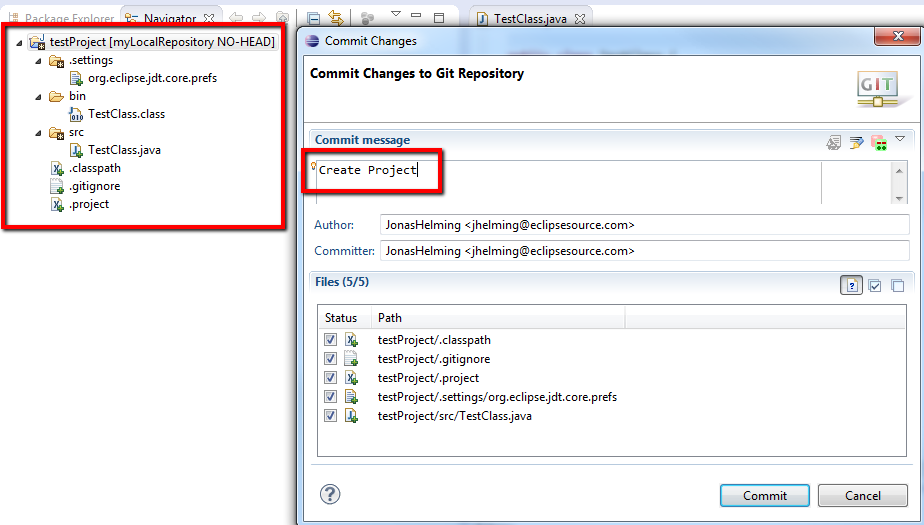
**Creating Local Repositories**

One major advantage of Git compared to SVN or CVS is that you can easily create local repositories, even before you share them with other people. In this way, you can version your work locally. First, you have to create a project in Eclipse that you want to share via your local repository. For later purposes it would be useful to add some files, e.g. a Java class to your project.

After you have created your project, select the context menu by right clicking it and navigate to *Team => Share Project…* . Select *Git* as the repository type and hit *Next*. In the following window select your project, hit the *Create Repository*-button and click *Finish*.

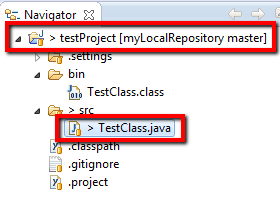


The newly created repository will be empty, although the project is assigned to it.  (Note the changed icons: the project node will have a repository icon, the child nodes will have an icon with a question mark, ignored files, e.g. the bin directory, won’t have any icons at all.) Before you can commit the files to your repository, you need to add them. Simply right click the shared project’s node and navigate to *Team => Add*. After this operation, the question mark should change to a plus symbol. To set certain folders or files to be ignored by Git, e.g. the bin folder, right click them and select *Navigate => Ignore*. The ignored items will be stored in a file called *gitignore*, which you should add to the repository. The last thing to do is commit the project by right clicking the project node and selecting *Team => Commit…* from the context menu. In the Commit wizard, all files should be selected automatically. Enter a commit message (the first line should be headline-like, as it will appear in the history view) and hit the *Commit* button. If the commit was successful, the plus symbols will have turned into repository icons.

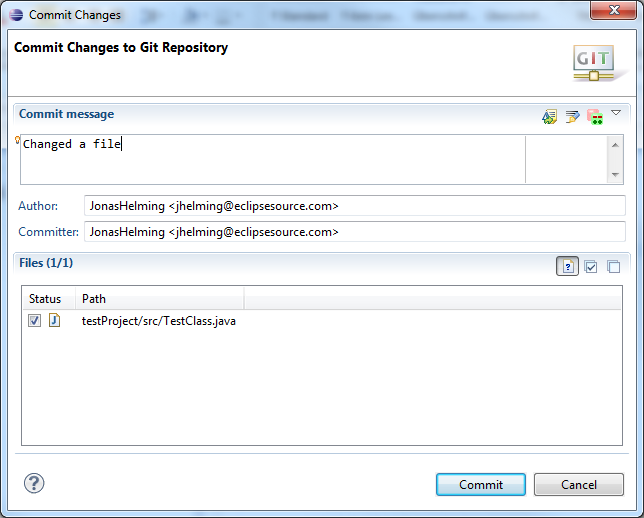


**Commit**

Now you can start to modify files in your project. To save changes made in your workspace to your repository, you will have to commit them. After changing files in your project, a “>” sign will appear right after the icon, telling you the status of these files is dirty. Any parent folder of this file will be marked as dirty as well.



If you want to commit the changes to your repository, right click the project (or the files you want to commit) and select *Team => Commit…* . This will open a new window, allowing you to select the files you want to commit. Before you can commit the files, you will have to enter a commit message in the upper textbox. After you’re done, click *Commit* to commit the selected files to your repository.



***You do not have to create your own project, all you need to do is import the project folder from my Repository on Github. Also, set up your name and e-mail in the Github Configuration screen as shown above.***