Lab: Reverting Commits and Rewriting History

In this lab, we will cover the following recipes:

- Undo – Remove a commit completely
- Undo – Remove a commit and retain changes to files
- Undo – Remove a commit and retain changes in the staging area
- Undo – Working with a dirty area
- Redo – Recreate the latest commit with new changes
- Revert – Undo the changes introduced by a commit

Tutorial Overview

Pre-reqs:

- Google Chrome (Recommended)

Lab Environment

There is no requirement for any setup.

Important: Instructions for this lab are written in such a way that it also shows expected output from the git cli. Only run commands that start with **$** as shown below.

Undo – Remove a commit completely

In this example, we'll learn how we can undo a commit as if it had never happened. We'll learn how we can use the reset command to effectively discard the commit and thereby reset our branch to the desired state.

Getting ready

In this example, we'll use the example of the `github_helloworld` repository, clone the repository, and change our working directory to the cloned one:

```
$ git clone https://github.com/fenago/github_helloworld.git
```

```
$ cd github_helloworld.git
```

How to do it...

First, we'll try to undo the latest commit in the repository as though it never happened:

1. We'll make sure that our working directory is clean, no files are in the modified state, and nothing is added to the index:

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

nothing to commit, working directory clean
```

2. Also, check what is in our working tree:

```
$ ls
HelloWorld.java Makefile        hello_world.c
```

3. If all works well, we'll check the log to see the history of the repository. We'll use the `--oneline` switch to limit the output:

```
$ git log --oneline
3061dc6 Adds Java version of 'hello world'
9c7532f Fixes compiler warnings
5b5d692 Initial commit, K&R hello world
```

4. The most recent commit is the `3061dc6 Adds Java version of 'hello world'` commit. We will now undo the commit as though it never happened, and the history won't show it:

```
$ git reset --hard HEAD^

HEAD is now at 9c7532f Fixes compiler warnings
```

5. Check the log, status, and filesystem, so that you can see what actually happened:

```
$ git log --oneline
9c7532f Fixes compiler warnings
5b5d692 Initial commit, K&R hello world

$ git status
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.

  (use "git pull" to update your local branch)

nothing to commit, working directory clean

$ ls
hello_world.c
```

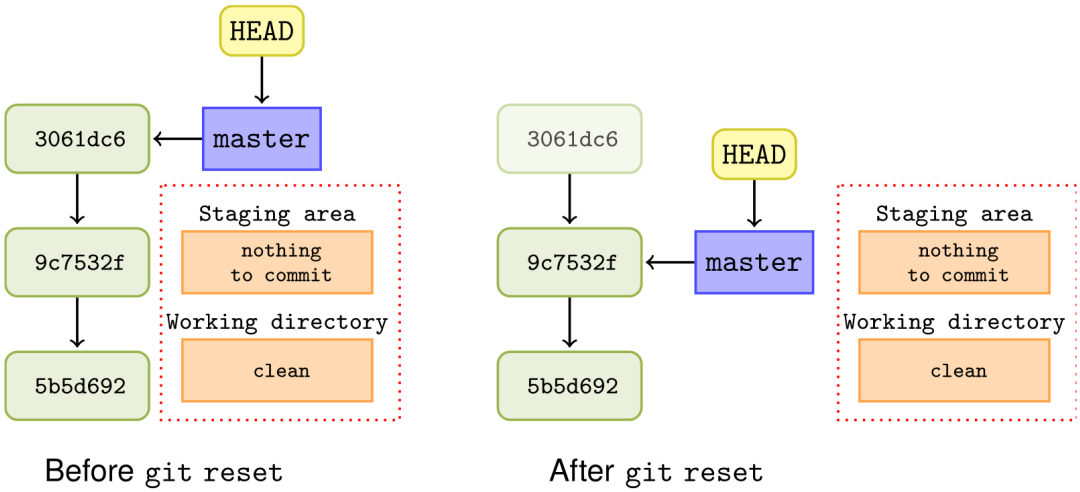
6. The commit is now gone, along with all the changes it introduced (`Makefile` and `HelloWorld.java`).

Note

In the last output of the `git status` command, you can see that our master branch is one behind `origin/master` . This is similar to what we mentioned at the beginning of the lab, because we are removing and undoing commits that are already published. Also, as mentioned, you should only perform the undo and redo (`git reset`) operations on commits that are not shared yet. Here, we only show it on the published commits to make the example easy to reproduce.

How it works...

Effectively, we are just changing the pointer of the masterbranch to point to the previous commit **HEAD**, which means the first parent of **HEAD**. Now, the branch will point to **9c7532f**, instead of the commit we removed, **35b29ae**. This is shown in the following diagram:



The preceding diagram also shows that the original **3061dc6** commit is still present in the repository, but new commits on themaster branch will start from **9c7532f**; the **3061dc6** commit is called a dangling commit.

Note

You should only perform this undo operation on commits you haven't shared (pushed) yet, since when you create new commits following undo or reset, those commits form a new history that will diverge from the original history of the repository.

When the reset command is executed, Git looks at the commit pointed to by **HEAD** and finds the parent commit from this. The current branch,master, and the **HEAD** pointer, are then reset to the parent commit, as are the staging area and working tree.

Undo – Remove a commit and retain changes to files

Instead of performing the hard reset and thereby losing all the changes the commit introduced, the reset can be performed so that the changes are retained in the working directory.

Getting ready

We'll again use the example of the hello world repository. Make a fresh clone of the repository, or reset the master branch if you have already cloned one.

You can make a fresh clone as follows:

```
$ git clone https://github.com/fenago/github_helloworld.git
$ cd github_helloworld
```

You can reset the existing clone as follows:

```
$ git checkout master
$ git reset --hard origin/master

HEAD is now at 3061dc6 Adds Java version of 'hello world'
```

How to do it...

1. First, we'll check whether we have made any changes to files in the working tree (just for the clarity of the example) and the history of the repository:

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

nothing to commit, working directory clean

$ git log --oneline
3061dc6 Adds Java version of 'hello world'
9c7532f Fixes compiler warnings
5b5d692 Initial commit, K&R hello world
```

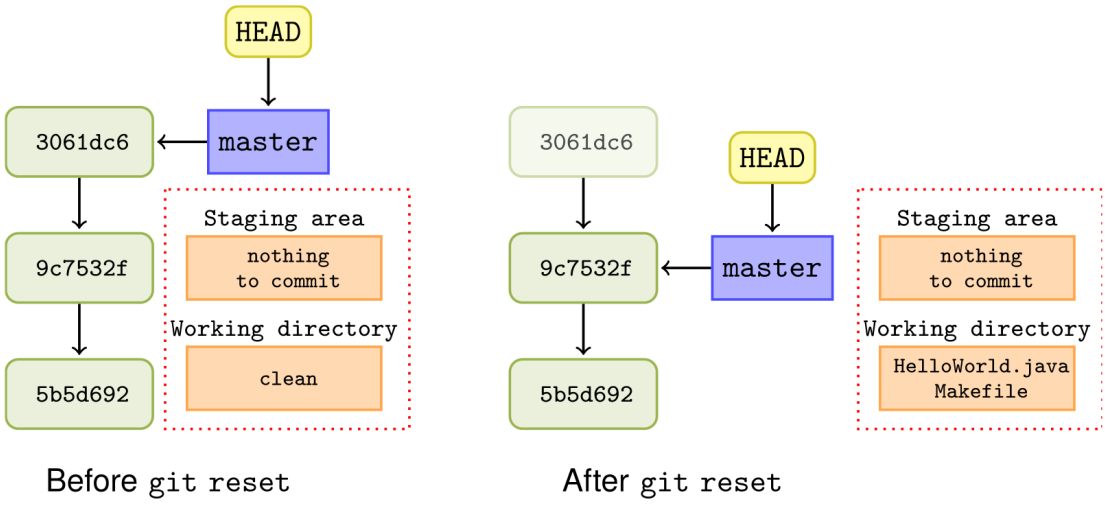
2. Now, we'll undo the commit and retain the changes introduced to the working tree:

```
$ git reset --mixed HEAD^

$ git log --oneline
9c7532f Fixes compiler warnings
5b5d692 Initial commit, K&R hello world

$ git status
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.

  (use "git pull" to update your local branch)

Untracked files:

  (use "git add <file>..." to include in what will be committed)

    HelloWorld.java
    Makefile

nothing added to commit but untracked files present (use "git add" to track)
```

We can see that our commit has been undone, but the changes to the file are preserved in the working tree, so more work can be done in order to create a proper commit.

How it works...

From the parent commit pointed to by the commit at **HEAD**, Git resets the branch pointer and **HEAD** to point to the parent commit. The staging area is reset, but the working tree is kept as it was before the reset, so the files affected by the `undone` commit will be in a modified state. This is illustrated in the following diagram:



Note

The `--mixed` option is the default behavior of `git reset`, so it can be omitted: `git reset HEAD^`

Undo – Remove a commit and retain changes in the staging area

Of course, it is also possible to undo the commit, but keep the changes to the files in the index or the staging area so that you are ready to recreate the commit with, for example, some minor modifications.

Getting ready

We'll still use the example of the hello world repository. Make a fresh clone of the repository, or reset the master branch if you have already cloned one.

Create a fresh clone as follows:

```
$ git clone https://github.com/fenago/github_helloworld.git
$ cd github_helloworld
```

We can reset the existing clone as follows:

```
 $ git checkout master
 $ git reset --hard origin/master

 HEAD is now at 3061dc6 Adds Java version of 'hello world'
```

How to do it...

1. Check whether we have any files in the modified state and check the log:

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

nothing to commit, working directory clean
```

```
$ git log --oneline
3061dc6 Adds Java version of 'hello world'
9c7532f Fixes compiler warnings
5b5d692 Initial commit, K&R hello world
```

2. Now, we can undo the commit, while retaining the changes in the index:

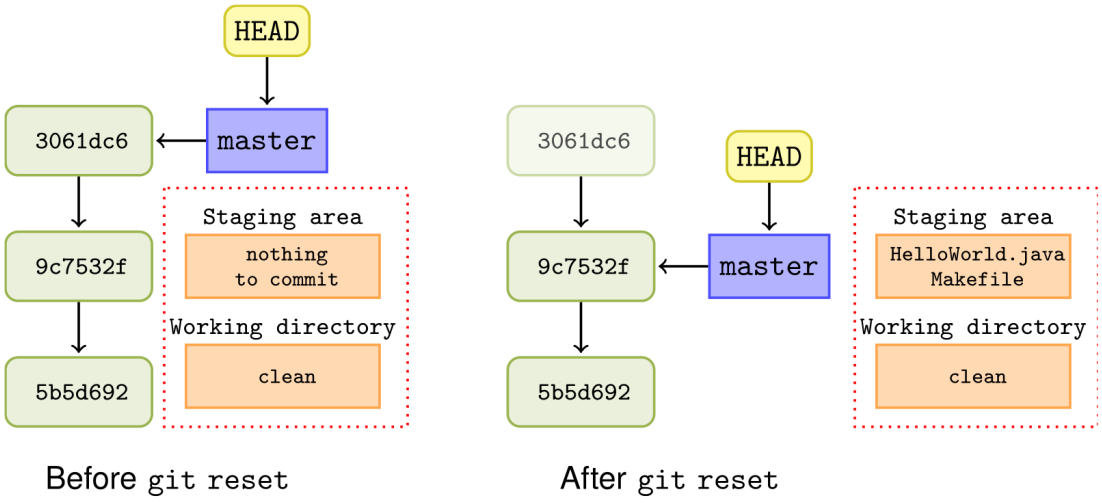
```
$ git reset --soft HEAD^

$ git log --oneline
9c7532f Fixes compiler warnings
5b5d692 Initial commit, K&R hello world

$ git status
On branch master

Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.

  (use "git pull" to update your local branch)

Changes to be committed:

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

    new file:   HelloWorld.java
    new file:   Makefile
```

You can now make minor (or major) changes to the files you need, add them to the staging area, and create a new commit.

How it works...

Again, Git will reset the branch pointer and **HEAD** to point to the previous commit. However, with the `--soft` option, the index and working directories are not reset, that is, they have the same state as they had before we created the now undone commit.

The following diagram shows the Git state before and after the undo:



Undo – Working with a dirty area

In the previous examples, we assumed that the working tree was clean, that is, no tracked files were in the modified state. However, this is not always the case, and if a hard reset is carried out, the changes to the modified files will be lost. Fortunately, Git provides a smart way to quickly put stuff away so that it can be retrieved later using the `git stash` command.

Getting ready

Again, we'll use the example of the hello world repository. Make a fresh clone of the repository, or reset the master branch if you have already cloned one.

We can create the fresh clone as follows:

```
$ git clone https://github.com/fenago/github_helloworld.git
$ cd github_helloworld
```

We can reset the existing clone as follows:

```
$ git checkout master
$ git reset --hard origin/master

HEAD is now at 3061dc6 Adds Java version of 'hello world'
```

We'll also need to have some files in the working condition, so we'll change `hello_world.c` to the following:

```
#include <stdio.h>

void say_hello(void) {
  printf("hello, worldn");
}

int main(void){
  say_hello();
```

```
  return 0;
}
```

How to do it...

In order to not accidentally delete any changes you have in your working tree when you are about to undo a commit, you can have a look at the current state of your working directory with `git status` command (as we already saw). If you have changes and you want to keep them, you can stash them away before undoing the commit and retrieve them afterward. Git provides a stash command that can put unfinished changes away, so it is easy to make quick context switches without losing work.

With the `hello_world.c` file in the working directory modified to the preceding state, we can try to do a hard reset on the `HEAD` commit, keeping our changes to the file by stashing them away before the reset and applying them again later:

1. First, check the history:

```
$ git log --oneline
3061dc6 Adds Java version of 'hello world'
9c7532f Fixes compiler warnings
5b5d692 Initial commit, K&R hello world
```

2. Then, check the status:

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

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

3. As expected, `hello_world.c` was in the modified state; so, stash it away, check the status, and perform the reset:

```
$ git stash
Saved working directory and index state WIP on master: 3061dc6 Adds Java version of
'hello world'

HEAD is now at 3061dc6 Adds Java version of 'hello world'

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean

$ git reset --hard HEAD^
```

```
HEAD is now at 9c7532f Fixes compiler warnings

$ git log --oneline
9c7532f Fixes compiler warnings
5b5d692 Initial commit, K&R hello world
```

4. The reset is done, and we got rid of the commit we wanted. Let's resurrect the changes we stashed away and check the file:

```
$ git stash pop
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.

  (use "git pull" to update your local branch)

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_world.c
no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (e56b68a1f5a0f72afcfd064ec13eefcda7a175ca)

$ cat hello_world.c

#include <stdio.h>
void say_hello(void) {
  printf("hello, worldn");
}
int main(void){
  say_hello();
  return 0;
}
```

So, the file is back to the state it was in before the reset, and we got rid of the unwanted commit.

How it works...

The reset command works as explained in the previous examples but, combined with thestashcommand, it forms a very useful tool that corrects mistakes even though you have already starting working on something else. Thestash command works by saving the current state of your working directory and the staging area. Then, it reverts your working directory to a clean state.

Undo – Working with a dirty area

In the previous examples, we assumed that the working tree was clean, that is, no tracked files were in the modified state. However, this is not always the case, and if a hard reset is carried out, the changes to the modified files will be lost. Fortunately, Git provides a smart way to quickly put stuff away so that it can be retrieved later using the `git stash` command.

Getting ready

Again, we'll use the example of the hello world repository. Make a fresh clone of the repository, or reset the master branch if you have already cloned one.

We can create the fresh clone as follows:

```
$ git clone https://github.com/fenago/github_helloworld.git
$ cd github_helloworld
```

We can reset the existing clone as follows:

```
$ git checkout master
$ git reset --hard origin/master

HEAD is now at 3061dc6 Adds Java version of 'hello world'
```

We'll also need to have some files in the working condition, so we'll change `hello_world.c` to the following:

```
#include <stdio.h>

void say_hello(void) {
  printf("hello, worldn");
}

int main(void){
  say_hello();
  return 0;
}
```

How to do it...

In order to not accidentally delete any changes you have in your working tree when you are about to undo a commit, you can have a look at the current state of your working directory with `git status` command (as we already saw). If you have changes and you want to keep them, you can stash them away before undoing the commit and retrieve them afterward. Git provides a stash command that can put unfinished changes away, so it is easy to make quick context switches without losing work.

With the `hello_world.c` file in the working directory modified to the preceding state, we can try to do a hard reset on the `HEAD` commit, keeping our changes to the file by stashing them away before the reset and applying them again later:

1. First, check the history:

```
$ git log --oneline
3061dc6 Adds Java version of 'hello world'
9c7532f Fixes compiler warnings
5b5d692 Initial commit, K&R hello world
```

2. Then, check the status:

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

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

3. As expected, `hello_world.c` was in the modified state; so, stash it away, check the status, and perform the reset:

```
$ git stash
Saved working directory and index state WIP on master: 3061dc6 Adds Java version of
'hello world'

HEAD is now at 3061dc6 Adds Java version of 'hello world'

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean

$ git reset --hard HEAD^
HEAD is now at 9c7532f Fixes compiler warnings

$ git log --oneline
9c7532f Fixes compiler warnings
5b5d692 Initial commit, K&R hello world
```

4. The reset is done, and we got rid of the commit we wanted. Let's resurrect the changes we stashed away and check the file:

```
$ git stash pop
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.

  (use "git pull" to update your local branch)

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_world.c
no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (e56b68a1f5a0f72afcfd064ec13eefcda7a175ca)

$ cat hello_world.c

#include <stdio.h>
void say_hello(void) {
  printf("hello, worldn");
```

```
}
int main(void){
  say_hello();
  return 0;
}
```

So, the file is back to the state it was in before the reset, and we got rid of the unwanted commit.

How it works...

The reset command works as explained in the previous examples but, combined with thestashcommand, it forms a very useful tool that corrects mistakes even though you have already starting working on something else. Thestash command works by saving the current state of your working directory and the staging area. Then, it reverts your working directory to a clean state.

Redo – Recreate the latest commit with new changes

As with undo, redo can mean a lot of things. In this context, redoing a commit will mean creating almost the same commit again with the same parent(s) as the previous commit, but with different content and/or different commit messages. This is quite useful if you've just created a commit, but have perhaps forgotten to add a necessary file to the staging area before you committed, or if you need to reword the commit message.

Getting ready

Again, we'll use the hello worldrepository. Make a fresh clone of the repository, or reset themaster branch if you have already cloned one.

We can create a fresh clone as follows:

```
$ git https://github.com/fenago/github_helloworld.git
$ cd github_helloworld
```

We can reset an existing clone as follows:

```
$ git checkout master
$ git reset --hard origin/master

HEAD is now at 3061dc6 Adds Java version of 'hello world'
```

How to do it...

Let's pretend we need to redo the latest commit because we need to reword the commit message to include a reference to the issue tracker.

1. Let's first take a look at the latest commit and make sure the working directory is clean:

```
$ git log -1

commit 3061dc6cf7aeb2f8cb3dee651290bfea85cb4392
Author: John Doe <john.doe@example.com>
Date:   Sun Mar 9 14:12:45 2014 +0100
    Adds Java version of 'hello world'
    Also includes a makefile
```

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

2. Now, we can redo the commit and update the commit message with the git commit `--amend` command. This will bring up the default editor, and we can add a reference to the issue tracker in the commit message (`Fixes: RD-31415`):

```
$ git commit --amend
Adds Java version of 'hello world'
Also includes a makefile
Fixes: RD-31415

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Author:    John Doe <john.doe@example.com>
#
# On branch master
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
#       new file:   HelloWorld.java
#       new file:   Makefile
#

~
~
[master 75a41a2] Adds Java version of 'hello world'
 Author: John Doe <john.doe@example.com>
 2 files changed, 19 insertions(+)
 create mode 100644 HelloWorld.java
 create mode 100644 Makefile
```

3. Now, let's check the log again to see whether everything worked:

```
$ git log -1

commit 75a41a2f550325234a2f5f3ba41d35867910c09c

Author:  John Doe <john.doe@example.com>
Date: Sun Mar 9 14:12:45 2014 +0100  Adds Java version of 'hello world'  Also includes
a makefile  Fixes: RD-31415
```

4. We can see that the commit message has changed, but we can't verify from the log output that the parent of the commit is the same as in the original commit, and so on, as we saw in the first commit we did. To check this, we can use the `git cat-file` command. First, let's see how the original commit looked:

```
$ git cat-file -p 3061dc6

tree d3abe70c50450a4d6d70f391fcbda1a4609d151f
```

```
parent 9c7532f5e788b8805ffd419fcf2a071c78493b23

author John Doe <john.doe@example.com> 1394370765 +0100
committer John Doe <john.doe@example.com> 1394569447 +0100 Adds Java version of 'hello
world' Also includes a makefile
```

The parent commit is `b8c39bb35c4c0b00b6cfb4e0f27354279fb28866`, and the root tree is `d3abe70c50450a4d6d70f391fcbda1a4609d151f`.

5. Let's check the data from the new commit:

```
$ git cat-file -p HEAD

tree d3abe70c50450a4d6d70f391fcbda1a4609d151f
parent 9c7532f5e788b8805ffd419fcf2a071c78493b23
author John Doe <john.doe@example.com> 1394370765 +0100
committer John Doe <john.doe@example.com> 1394655225 +0100

Adds Java version of 'hello world'
Also includes a makefile
Fixes: RD-31415
```

The parent is the same, that is, `9c7532f5e788b8805ffd419fcf2a071c78493b23` and the root tree is also the same, that is, `d3abe70c50450a4d6d70f391fcbda1a4609d151f`. This is what we expected as we only changed the commit message. If we had added some changes to the staging area and executed `git commit--amend`, we would have included those changes in the commit and the root-tree SHA1 ID would have been different, but the parent commit ID still the same.

How it works...

The `--amend` option to git commit is roughly equivalent to performing `git reset --soft HEAD^`, followed by fixing the files needed and adding those to the staging area. Then, we will run git commit reusing the commit message from the previous commit (`git commit -c ORIG_HEAD`).

There's more...

We can also use the `--amend` method to add missing files to our latest commit. Let's say you needed to add the `README.md` file to your latest commit in order to get the documentation up to date, but you have already created the commit, though you have not pushed it yet.

You then add the file to the index as you would while starting to craft a new commit. You can check with git status that only the `README.md` file is added:

```
$ git add README.md

$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commit each, respectively.
  (use "git pull" to merge the remote branch into yours)

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
```

```
  new file:   README.md
```

Now, you can amend the latest commit with `git commit --amend` . The command will include files in the index in the new commit and you can, as with the last example, reword the commit message if needed. It is not needed in this example, so we'll pass the `--no-edit` option to the command:

```
$ git commit --amend --no-edit

[master f09457e] Adds Java version of 'hello world'
 Author: John Doe <john.doe@example.com>
 3 files changed, 20 insertions(+)
 create mode 100644 HelloWorld.java
 create mode 100644 Makefile
 create mode 100644 README.md
```

You can see from the output of the commit command that three files were changed and `README.md` was one of them.

Note

You can also reset the author information (name, email, and timestamp) with the commit `--amend` command. Just pass along the `--reset-author` option and Git will create a new timestamp and read author information from the configuration or environment, instead of the using information from the old commit object.

Revert – Undo the changes introduced by a commit

Revert can be used to undo a commit in history that has already been published (pushed), whereas this can't be done with the amend or reset options without rewriting history.

Revert works by applying the anti-patch introduced by the commit in question. A revert will, by default, create a new commit in history with a commit message that describes which commit has been reverted.

Getting ready

Again, we'll use the hello worldrepository. Make a fresh clone of the repository, or reset themaster branch if you have already cloned one.

We can create a fresh clone as follows:

```
$ git clone https://github.com/fenago/github_helloworld.git
$ cd github_helloworld
```

We can reset the existing clone as follows:

```
$ cd github_helloworld
$ git checkout master
$ git reset --hard origin/master
HEAD is now at 3061dc6 Adds Java version of 'hello world'
```

How to do it...

1. First, we'll list the commits in the repository:

```
$ git log --oneline
3061dc6 Adds Java version of 'hello world'
9c7532f Fixes compiler warnings
5b5d692 Initial commit, K&R hello world
```

2. We'll revert the second commit, `9c7532f` :

```
$ git revert 9c7532f

Revert "Fixes compiler warnings"
This reverts commit 9c7532f5e788b8805ffd419fcf2a071c78493b23.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
#       modified:   hello_world.c
#
~
~
~
"~/john.doe/fenago/repos/github_helloworld/.git/COMMIT_EDITMSG" 12L, 359C [master
9b94515] Revert "Fixes compiler warnings"  1 file changed, 1 insertion(+), 5
deletions(-)
```

3. When we check the log, we can see that a new commit has been made:

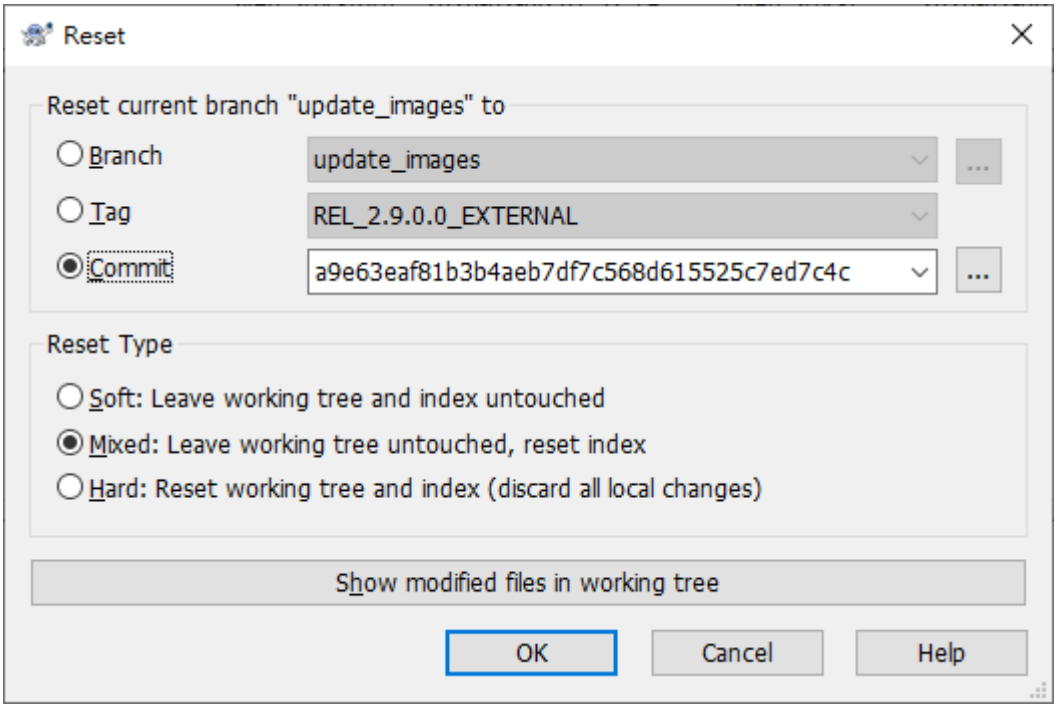
```
$ git log --oneline
9b94515 Revert "Fixes compiler warnings"
3061dc6 Adds Java version of 'hello world'
9c7532f Fixes compiler warnings
5b5d692 Initial commit, K&R hello world
```

We can take a closer look at the two commits with `git show` if we want a closer investigation of what happened.

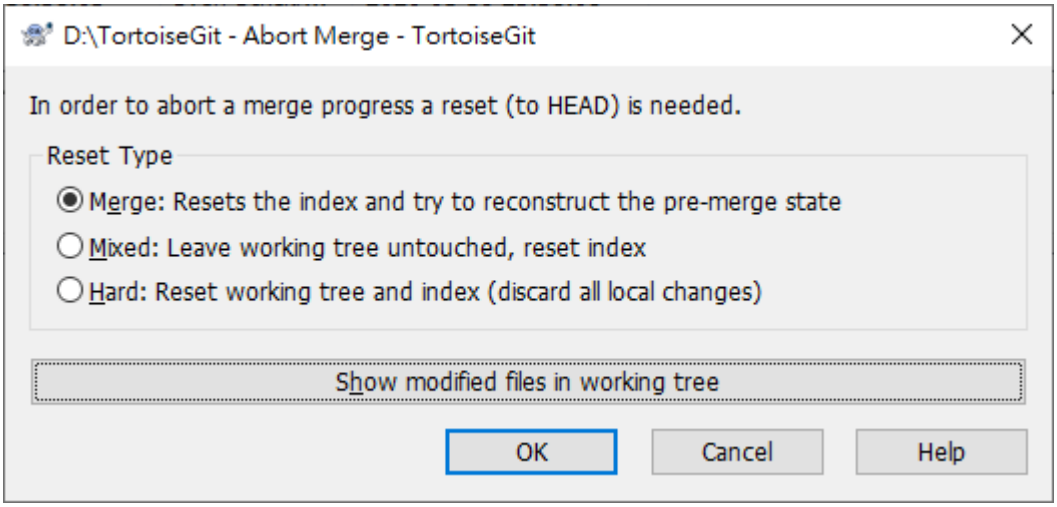
Git reset with TortoiseGit

The reset dialog can be used to reset the current HEAD to the specified state and optionally also the index and the working tree. This can also be used to abort a merge.

The Reset dialog



The Abort Merge dialog



On the Reset dialog, you can click `...` to browse the log and choose a specific version. In Abort merge dialog, you can only reset to HEAD.

`Soft: Leave working tree and index untouched` Does not touch the index file nor the working tree at all (but resets the head to the selected commit, just like all modes do). This leaves all your changed files "Changes to be committed" as before. This option is not available in Abort Merge dialog.

`Mixed: Leave working tree untouched`, reset index Resets the index but not the working tree (i.e., the changed files are preserved but not marked for commit) and reports what has not been updated. This is the git default action. This option can abort a merge.

`Hard: Reset working tree and index (discard all local changes)` Resets the index and working tree. Any changes to tracked files in the working tree since the selected commit are discarded. This option can abort a merge, and it is the default action in Abort Merge dialog.