

Presentation Week 1 - Day 1

Introduction to Git

Makers Institute 

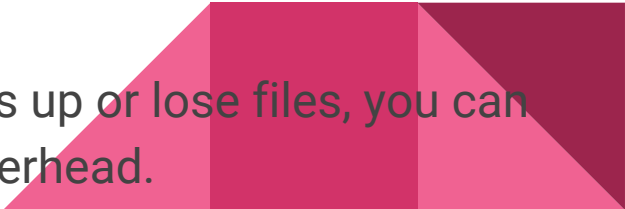
Getting Started

About Version Control

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.

If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to revert files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more.

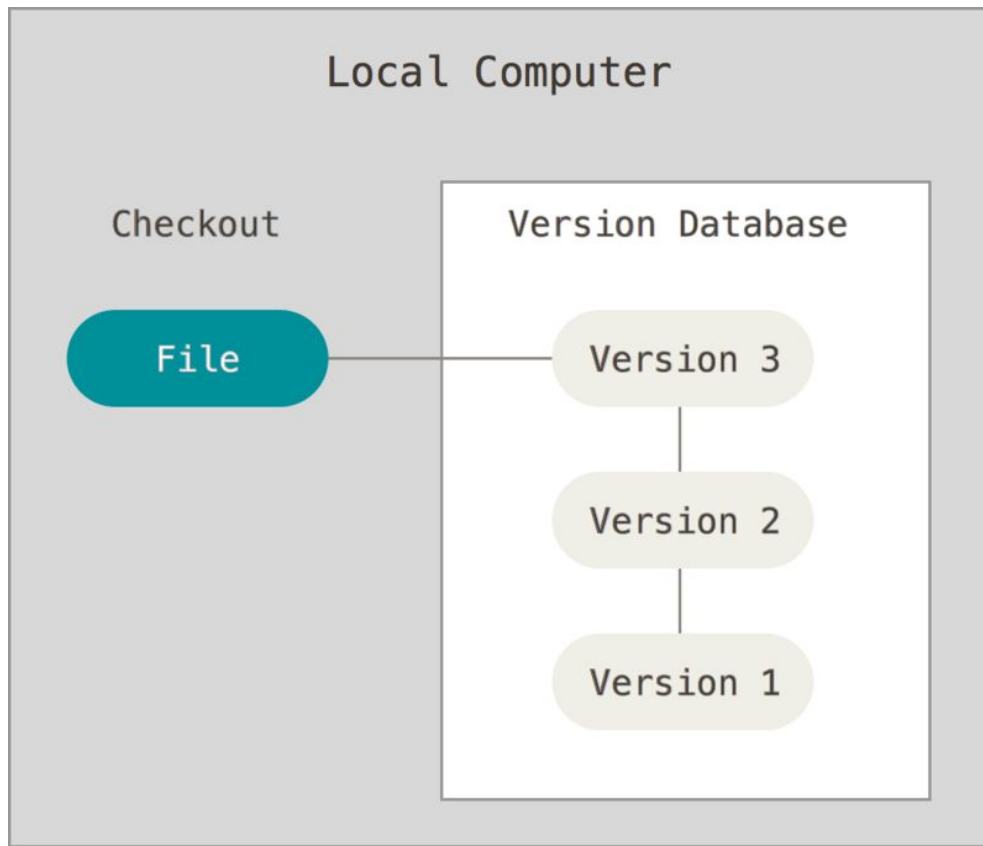
Using a VCS also generally means that if you screw things up or lose files, you can easily recover. In addition, you get all this for very little overhead.



About Version Control

Local Version Control Systems

Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

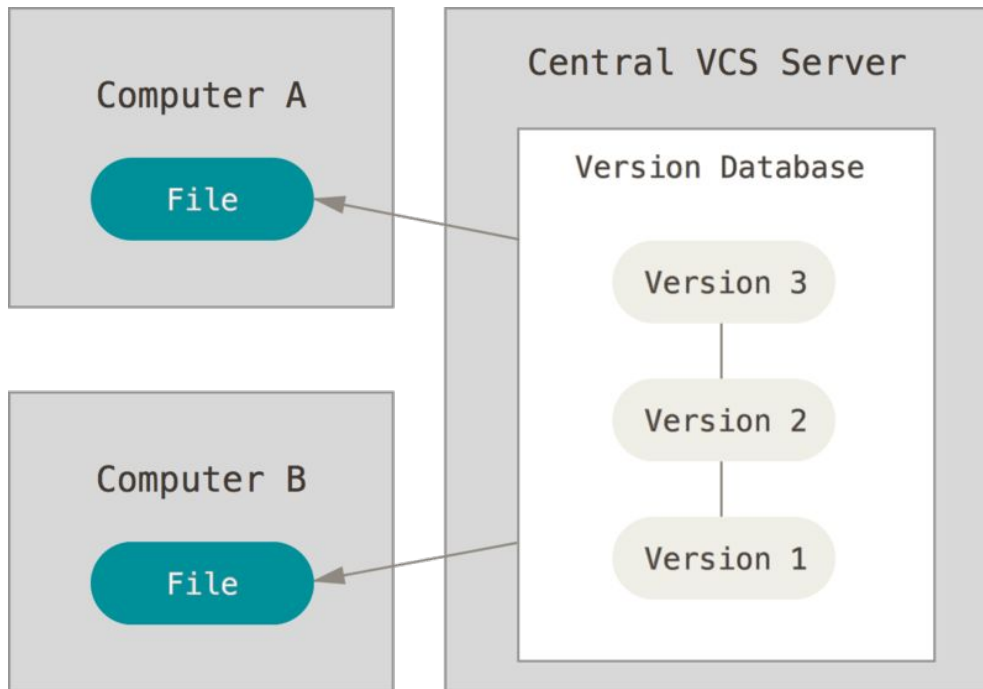


About Version Control

Centralized Version Control Systems

Advantages: Everyone knows to a certain degree what everyone else on the project is doing. Administrators have fine-grained control over who can do what; and it's far easier to administer a CVCS than it is to deal with local databases on every client.

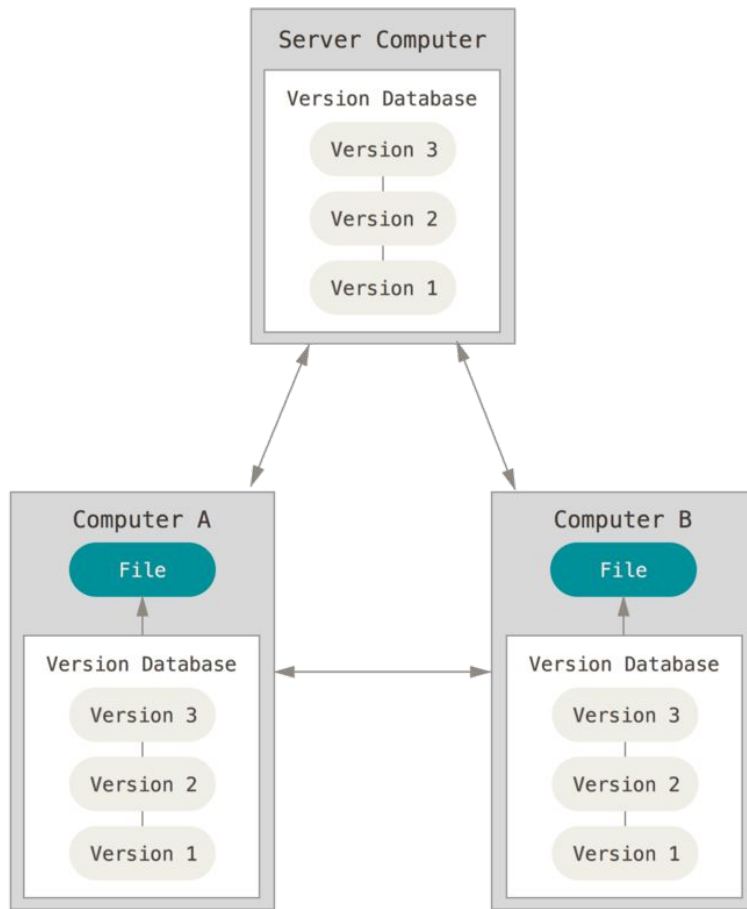
Downsides: If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they're working on.



About Version Control

Distributed Version Control Systems

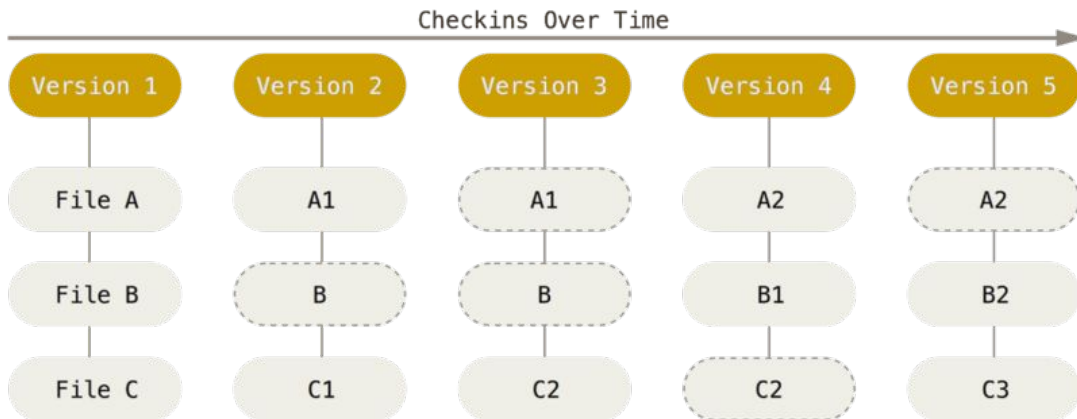
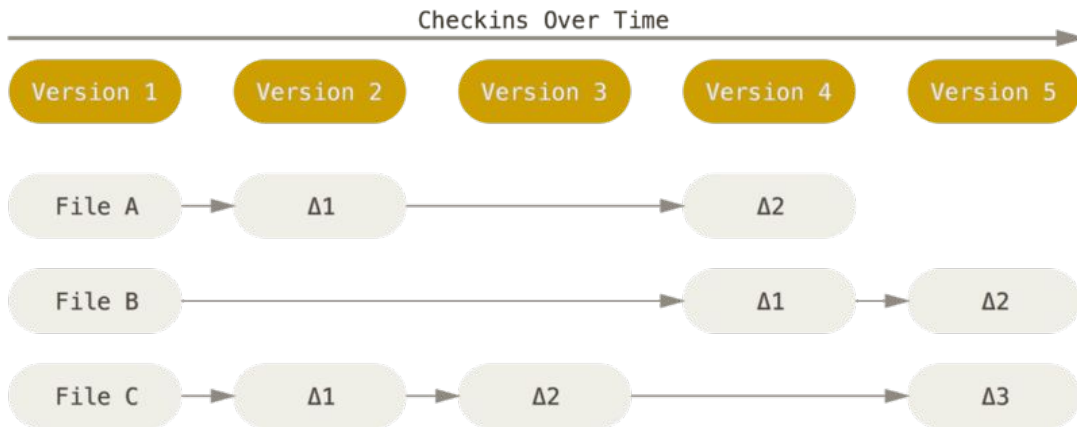
This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files: they fully mirror the repository. Thus if any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.



Git Basics

Snapshots, Not Differences

Git doesn't think of or store its data this way. Instead, Git thinks of its data more like a set of snapshots of a miniature filesystem. Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a **stream of snapshots**.



Git Basics

Nearly Every Operation Is Local

Most operations in Git only need local files and resources to operate – generally no information is needed from another computer on your network. If you're used to a CVCS where most operations have that network latency overhead, this aspect of Git will make you think that the gods of speed have blessed Git with unworldly powers. Because you have the entire history of the project right there on your local disk, most operations seem almost instantaneous.

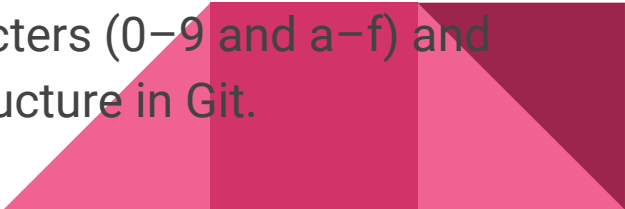


Git Basics

Git Has Integrity

Everything in Git is check-summed before it is stored and is then referred to by that checksum. This means it's impossible to change the contents of any file or directory without Git knowing about it. This functionality is built into Git at the lowest levels and is integral to its philosophy. You can't lose information in transit or get file corruption without Git being able to detect it.

The mechanism that Git uses for this checksumming is called a SHA-1 hash. This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git.



Git Basics

Git Generally Only Adds Data

When you do actions in Git, nearly all of them only add data to the Git database. It is hard to get the system to do anything that is not undoable or to make it erase data in any way. As in any VCS, you can lose or mess up changes you haven't committed yet; but after you commit a snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository.



Git Basics

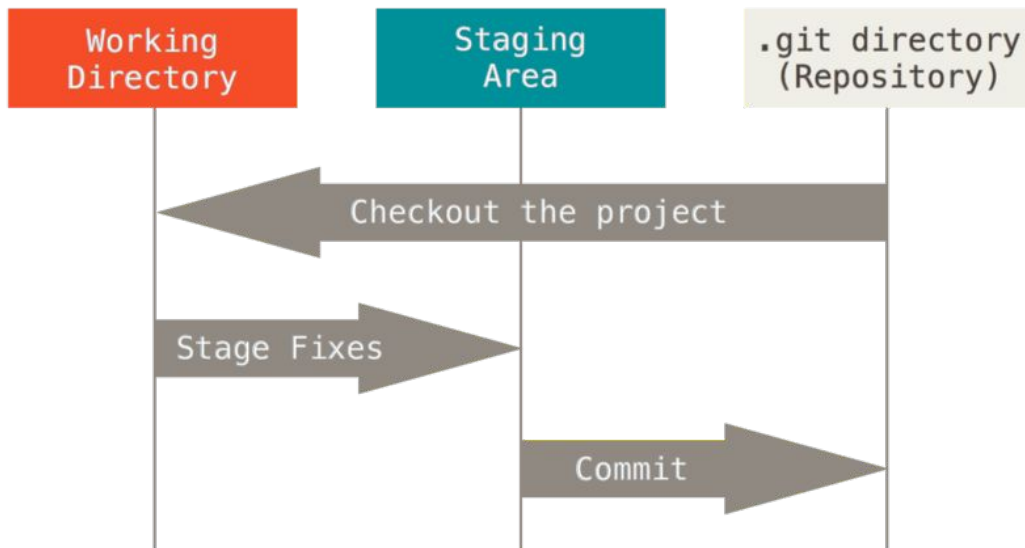
The Three States

Now, pay attention. This is the main thing to remember about Git if you want the rest of your learning process to go smoothly. Git has three main states that your files can reside in: committed, modified, and staged.

Committed means that the data is safely stored in your local database.

Modified means that you have changed the file but have not committed it to your database yet.

Staged means that you have marked a modified file in its current version to go into your next commit snapshot.



The Basics

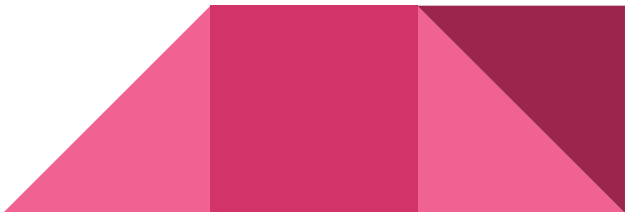


Git is a collection of command line utilities that track and record changes in files. This process is referred to as **version control**.

Git is decentralized, it works fully locally by storing this data as a folder on your hard drive, which we call a **repository**. However you can store a copy of your repository online.

1. Installing Git

Installing Git on your machine is straightforward:

- Linux – Simply open up a new terminal and install git via your distribution's package manager. For Ubuntu the command is: `sudo apt-get install git`
 - Windows – we recommend [git for windows](#) as it offers both a GUI client and a BASH command line emulator.
 - OS X – The easiest way is to install [homebrew](#), and then just run `brew install git` from your terminal.
- 

2. Configuring Git

Now that we've installed git on our computer, we will need to add some quick configurations. There are a lot of options that can be fiddled with, but we are going to set up the most important ones: our username and email. Open a terminal and run these commands:

```
$ git config --global user.name "My Name"  
$ git config --global user.email myEmail@example.com
```



3. Creating a new repository – `git init`

To set up a new repository, we need to open a terminal, navigate to our project directory and run `git init`. This will enable Git for this particular folder and create a hidden `.git` directory where the repository history and configuration will be stored.

Create a folder on your Desktop called `git_exercise`, open a new terminal and enter the following:

```
$ cd Desktop/git_exercise/  
$ git init
```

Now create a simple text file called *hello.txt* and save it in the *git_exercise* folder.

4. Checking the status – `git status`

Git status is another must-know command that returns information about the current state of the repository: is everything up to date, what's new, what's changed, and so on. Running `git status` in our newly created repo should return the following:



4. Checking the status – `git status`

```
$ git status
```

```
On branch master
```

```
Initial commit
```

```
Untracked files:
```

```
(use "git add ..." to include in what will be committed)
```

```
hello.txt
```



5. Staging – `git add`

Git has the concept of a “*staging area*”. You can think of this like a blank canvas, which holds the changes which you would like to commit. It starts out empty, but you can add files to it (or even single lines and parts of files) with the `git add` command, and finally commit everything (create a snapshot) with `git commit`.

In our case we have only one file so let's add that:

```
$ git add hello.txt
```



5. Staging – `git add`

If we want to add everything in the directory, we can use:

```
$ git add -A
```

Checking the status again should return a different response from before.



5. Staging – `git add`

```
$ git status
```


On branch master

Initial commit

Changes to be committed:

(use "git rm --cached ..." to unstage)

new file: hello.txt



6. Committing – `git commit`

A commit represents the state of our repository at a given point in time. It's like a snapshot, which we can go back to and see how things were when we took it.

To create a new commit we need to have at least one change added to the staging area (we just did that with `git add`) and run the following:

```
$ git commit -m "Initial commit."
```



Remote repositories



Right now our commit is local – it exist only in the `.git` folder. Although a local repository is useful by itself, in most cases we will want to share our work and deploy it to a server or a repository hosting service.

1. Connecting to a remote repository – `git remote add`

In order to upload something to a remote repo, we first have to establish a connection with it. We create our own empty repository at [GitHub](#). The registration and setup may take a while, but all services offer good step-by-step guides to help you.

To link our local repository with the one on GitHub, we execute the following line in the terminal:

```
$ git remote add origin  
https://github.com/repo-name/proj-name.git
```

2. Uploading to a server – `git push`

Now it's time to transfer our local commits to the server. This process is called a **push**, and is done every time we want to update the remote repository.

The Git command to do this is `git push` and takes two parameters – the name of the remote repo (we called ours *origin*) and the branch to push to (*master* is the default branch for every repo).



2. Uploading to a server – `git push`

```
$ git push origin master
```

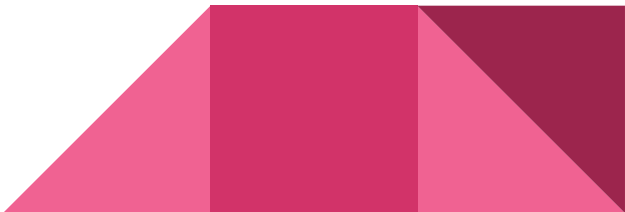
```
Counting objects: 3, done.
```

```
Writing objects: 100% (3/3), 212 bytes | 0 bytes/s, done.
```

```
Total 3 (delta 0), reused 0 (delta 0)
```

```
To https://github.com/tutorialzine/awesome-project.git
```

```
    * [new branch]      master -> master
```



3. Cloning a repository – `git clone`

At this point, people can see and browse through your remote repository on Github. They can download it locally and have a fully working copy of your project with the `git clone` command:

```
$ git clone https://github.com/repo-name/proj-name.git
```

A new local repository is automatically created, with the github version configured as a remote.



4. Getting changes from a server – `git pull`

If you make updates to your repository, people can download your changes with a single command – **pull**:

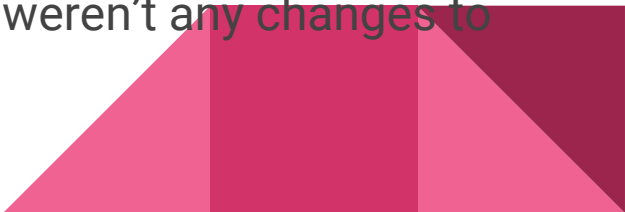
```
$ git pull origin master
```

```
From https://github.com/tutorialzine/awesome-project
```

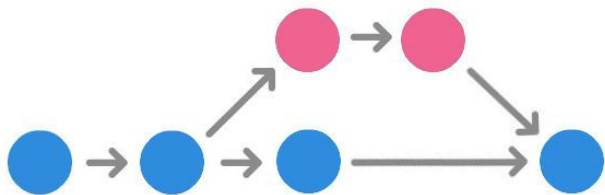
```
* branch                master      -> FETCH_HEAD
```

```
Already up-to-date.
```

Since nobody else has committed since we cloned, there weren't any changes to download.



Branches



This is done for a couple of reasons:

- An already working, stable version of the code won't be broken.
- Many features can be safely developed at once by different people.
- Developers can work on their own branch, without the risk of their codebase changing due to someone else's work.
- When unsure what's best, multiple versions of the same feature can be developed on separate branches and then compared.

1. Creating new branches – `git branch`

The default branch of every repository is called **master**. To create additional branches use the `git branch <name>` command:

```
$ git branch amazing_new_feature
```

This just creates the new branch, which at this point is exactly the same as our *master*.



2. Switching branches – `git checkout`

Now, when we run `git branch`, we will see there are two options available:

```
$ git branch
  amazing_new_feature
* master
```

Master is the current branch and is marked with an asterisk. However, we want to work on our new amazing features, so we need to switch to the other branch. This is done with the `git checkout` command, expecting one parameter – the branch to switch to.

```
$ git checkout amazing_new_feature
```

3. Merging branches – `git merge`

Our “amazing new feature” is going to be just another text file called *feature.txt*. We will create it, add it, and commit it.

```
$ git add feature.txt  
$ git commit -m "New feature complete."
```

The new feature is complete, we can go back to the master branch.

```
$ git checkout master
```



3. Merging branches – `git merge`

Now, if we open our project in the file browser, we'll notice that *feature.txt* has disappeared. That's because we are back in the master branch, and here *feature.txt* was never created. To bring it in, we need to `git merge` the two branches together, applying the changes done in *amazing_new_feature* to the main version of the project.

```
$ git merge amazing_new_feature
```

The master branch is now up to date. The *awesome_new_feature* branch is no longer needed and can be removed.

```
$ git branch -d awesome_new_feature
```


Advanced



In the last section, we are going to take a look at some more advanced techniques that are very likely to come in handy.

1. Checking difference between commits

Every commit has its unique id in the form of a string of numbers and symbols. To see a list of all commits and their ids we can use `git log`:

```
$ git log
```

```
commit ba25c0ff30e1b2f0259157b42b9f8f5d174d80d7
```

```
Author: Tutorialzine
```

```
Date:   Mon May 30 17:15:28 2016 +0300
```

```
    New feature complete
```

```
commit b10cc1238e355c02a044ef9f9860811ff605c9b4
```

```
Author: Tutorialzine
```

```
Date:   Mon May 30 16:30:04 2016 +0300
```

```
    Added content to hello.txt
```

```
commit 09bd8cc171d7084e78e4d118a2346b7487dca059
```

```
Author: Tutorialzine
```

```
Date:   Sat May 28 17:52:14 2016 +0300
```

```
    Initial commit
```

1. Checking difference between commits

As you can see the ids are really long, but when working with them it's not necessary to copy the whole thing – the first several symbols are usually enough.

To see what was new in a commit we can run `git show [commit]`:

```
$ git show b10cc123
```

```
commit b10cc1238e355c02a044ef9f9860811ff605c9b4
```

```
Author: Tutorialzine
```

```
Date:   Mon May 30 16:30:04 2016 +0300
```

```
    Added content to hello.txt
```

```
diff --git a/hello.txt b/hello.txt
```

```
index e69de29..b546a21 100644
```

```
--- a/hello.txt
```

```
+++ b/hello.txt
```

```
@@ -0,0 +1 @@
```

```
+Nice weather today, isn't it?
```

1. Checking difference between commits

To see the difference between any two commits we can use `git diff` with the `[commit-from]..[commit-to]` syntax:

```
$ git diff 09bd8cc..ba25c0ff

diff --git a/feature.txt b/feature.txt
new file mode 100644
index 0000000..e69de29
diff --git a/hello.txt b/hello.txt
index e69de29..b546a21 100644
--- a/hello.txt
+++ b/hello.txt
@@ -0,0 +1 @@
+Nice weather today, isn't it?
```

We've compared the first commit to the last one, so we see all the changes that have ever been made. Usually it's easier to do this using the `git difftool` command which brings up a graphical client showing all differences side to side.

2. Reverting a file to a previous version

Git allows us to return any selected file back to the way it was in a certain commit. This is done via the familiar `git checkout` command, which we used earlier to switch branches, but can also be used to switch between commits (it's pretty common in Git for one command to be used for multiple seemingly unrelated tasks).

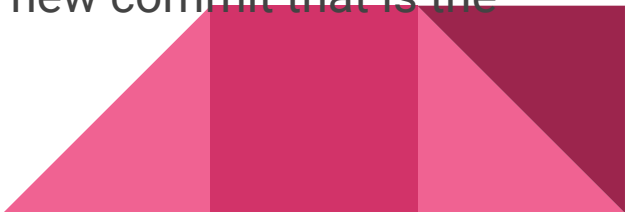
In the following example we will take `hello.txt` and reverse everything we've done to it since the initial commit. To do so we have to supply the id of the commit we want to go back to, as well as the full path to our file.

```
$ git checkout 09bd8cc1 hello.txt
```

3. Fixing a commit

If you notice that you've made a typo in your commit message, or you've forgotten to add a file and you see right after you commit, you can easily fix this with `git commit --amend`. This will add everything from the last commit back to the staging area, and attempt to make a new commit. This gives you a chance to fix your commit message or add more files to the staging area.

For more complex fixes that aren't in the last commit (or if you've pushed your changes already), you've got to use `git revert`. This will take all the changes that a commit has introduced, reverse them, and create a new commit that is the exact opposite.



3. Fixing a commit

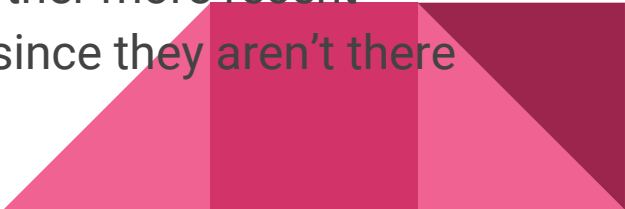
The newest commit can be accessed by the HEAD alias.

```
$ git revert HEAD
```

For other commits it's best to use an id.

```
$ git revert b10cc123
```

When reverting older commits, keep in mind that merge conflicts are very likely to appear. This happens when a file has been altered by another more recent commit, and now Git cannot find the right lines to revert, since they aren't there anymore.



4. Resolving Merge Conflicts

Apart from the scenario depicted in the previous point, conflicts regularly appear when merging branches or pulling someone else's work. Sometimes conflicts are handled automatically by git, but other times the person dealing with them has to decide (and usually handpick) what code stays and what is removed.

Let's look at an example where we're trying to merge two branches called `john_branch` and `tim_branch`. Both John and Tim are writing in the same file a function that displays all the elements in an array.



4. Resolving Merge Conflicts

John is using a for loop:

```
// Use a for loop to console.log contents.  
for(var i=0; i<arr.length; i++) {  
    console.log(arr[i]);  
}
```

Tim prefers forEach:

```
// Use forEach to console.log contents.  
arr.forEach(function(item) {  
    console.log(item);  
});
```

4. Resolving Merge Conflicts

They both commit their code on their respective branch. Now if they try to merge the two branches they will see the following error message:

```
$ git merge tim_branch
```

```
Auto-merging print_array.js
```

```
CONFLICT (content): Merge conflict in print_array.js
```

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

Git wasn't able to merge the branches automatically, so now it's up to the devs to manually resolve the conflict. If they open the file where the conflict resides, they'll see that Git has inserted a marker on the conflicting lines.

4. Resolving Merge Conflicts

```
<<<<<<< HEAD
// Use a for loop to console.log contents.
for(var i=0; i<arr.length; i++) {
    console.log(arr[i]);
}
=====
// Use forEach to console.log contents.
arr.forEach(function(item) {
    console.log(item);
});
>>>>>>> Tim's commit.
```

4. Resolving Merge Conflicts

Above the ===== we have the current HEAD commit, and below the conflicting one. This way we can clearly see the differences, and decide which is the better version, or write a new one all together. In this situation we go for the latter and rewrite the whole thing, removing the markers to let Git know we're done.

```
// Not using for loop or forEach.  
// Use Array.toString() to console.log contents.  
console.log(arr.toString());
```

When everything is set, a merge commit has to be done to finish the process.

```
$ git add -A  
$ git commit -m "Array printing conflict resolved."
```

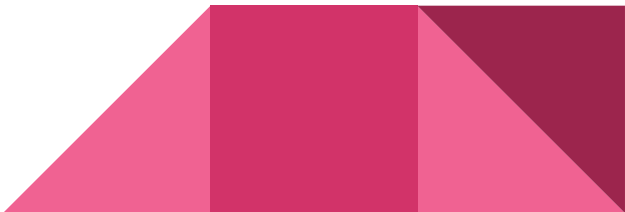
4. Resolving Merge Conflicts

As you can see this process is quite tiresome and can get extremely hard to deal with in large projects. Most developers prefer to resolve conflicts with the help of a GUI client, which makes things much easier. To run a graphical client use `git mergetool`.



5. Setting up .gitignore

In most projects there are files or entire folders that we don't want to ever commit. We can make sure that they aren't accidentally included in our `git add -A` by creating a .gitignore file:

1. Manually create a text file called .gitignore and save it in your project's directory.
 2. Inside, list the names of files/directories to be ignored, each on a new line.
 3. The .gitignore itself has to be added, committed and pushed, as it is just like any other file in the project.
- 

5. Setting up .gitignore

Good examples for files to be ignored are:

- log files
- task runner builds
- the node_modules folder in node.js projects
- folders created by IDEs like Netbeans and IntelliJ
- personal developer notes



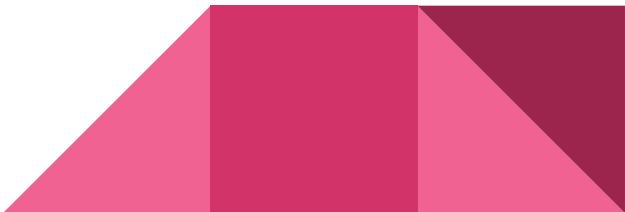
5. Setting up .gitignore

A .gitignore banning all of the above will look like this:

```
*.log  
build/  
node_modules/  
.idea/  
my_notes.txt
```

The slash at the end of some of the lines signals that this is a folder and we are ignoring everything inside it recursively. The asterisk serves its usual function as a wild card.

What we have learned so far:

- How to initialize git repository
 - How to check statuses on our repository
 - How to add files and commit changes
 - How to create online repository and push our local commits to the server
 - How to use clone to copy a full working project
 - How to pull an update on our remote repository
 - How to create, switching, and merging branches
 - How to check difference between commits
 - How to revert file to a previous version
 - How to fix a commit
 - How to resolve merge conflicts
- 

Questions?