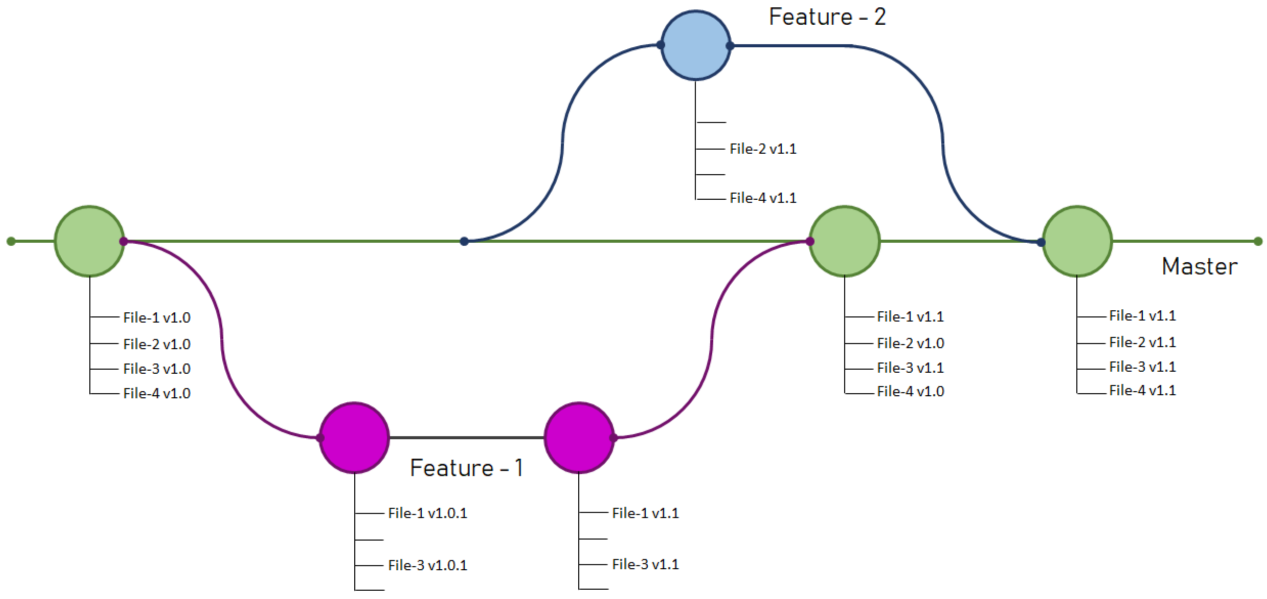
Git: branches and switching between versions

So, you've been working on your project for some time and you find yourself in one of the following situations:

- you would like to implement a new big **feature**, but you want to keep a clean working version of your code while you're developing and testing it.
- you have a **new idea** about your project that you're not sure about yet and just want to test it without breaking anything else.
- your **colleague** is also working on some new feature, and you don't want to mix your codes until they are properly tested.



If you run `git status` in your repository, git will probably return `on branch master`, which means that you're on the default branch of your code. Usually, you want to keep the `master` as clean as possible from bugs and raw code, because `master` is what runs your robot, product, important computations, etc. Also, `master` is what others usually see and use on your remote repository, so that's another piece of motivation to keep your `master` clean!

Now you want to introduce some changes to your project. In programming, changes almost always come with errors and bugs, until you properly debug and test your program. How do you keep this process away from `master`? Answer is: you create a new branch.

Branching

Create a new branch using `git branch [name]`

In terminal, in your repository folder, run `git status` to make sure that you are on `master` branch.

Now, run

```
git branch [name]
```

This will create a branch with the specified name. However, if you repeat `git status`, you'll find out that you're still on `master`. So, let us switch to the new branch!

Switch to the new branch using `git checkout [name]`

To switch from current branch to another branch, use:

```
git checkout [name]
```

For example, if you called your branch 'segmentation_fix' using `git branch segmentation_fix`, you switch to it using `git checkout segmentation_fix`. Once you pressed enter, you should see:

```
Switched to branch 'segmentation_fix'
```

Note: Personally, I don't think that the word *checkout* really fits to the function of command `checkout`. However, if you want to memorise it, note that one of the meanings of *check out* in English is *to take a look*.

Note 2: You can create a new branch and switch to it using one single command `git checkout -b [name]`.

List existing branches using `git branch`

If you forgot the name of the branch you want to switch to, run

```
git branch
```

without the name, it will simply list the available branches and mark the active one with an asterisk `*`.

Work on the branch the way you worked on `master`

On the new branch, you can do everything you did on master: create and edit files, `git commit`, `git push`, and even create new branches.

Merge branch using `git merge [name] -m "Message"`

Every branch is meant to finally get back to the `master` (or to be forgotten forever, if you stop working on it). Once you have made and tested your changes in the branch, it is time to *merge* it to the `master`. That is, you will apply changes made in the brach to the main version of the code.

First, `git checkout master` to switch to `master`, then:

```
git merge [name] -m "Your comment"
```

Sometimes, `git merge` also automatically runs `git commit`, secretly from you. This is why we specify a message in this command. Why and when `git merge` runs `git commit` is out of the scope of this class, but if you want you may learn it here.

Sometimes, branches that you try to merge may have conflicting versions of the same files. Git has mechanisms of resolving them automatically, but sometimes it cannot handle it all by itself. You probably won't run into this issue working on your projects at ECN. But in the future, when extensively using git, you will need to learn how to resolve the conflicts.

Delete branch using `git branch -d [name]`

Once you have merged your branch to `master`, you may want to delete it, to keep your repository clean of unnecessary branches.

To delete an existing branch, switch to `master` and run

```
git branch -d [name]
```

Don't worry, if you have successfully merged your branch before deleting it, you won't lose any of your code. If the branch wasn't merged, git just won't let you delete it that easily.

.gitignore

Untrack files using `git rm --cached` and *.gitignore*

If, by mistake, you have `git add`-ed a local file that you don't want to appear on the remote server (like file with your private SHA keys or passwords), you can use `git rm --cached` to fix it:

```
git rm --cached [filename]
```

Note: it is important to use `--cached` option, because plain `git rm` will also delete the file itself.

Now, if you run `git status`, you will see that the file is not staged for the commit anymore. But seeing it in red every time we run `git status` is not what we want, especially if we have many files like that. To make git completely ignore this file, we will use a special document called *.gitignore* (yes, with a dot at the beginning). Let us create it:

```
touch .gitignore
atom .gitignore
```

and, supposing that the file that you want to be ignored by git is called *secret.txt*, we write the following inside of *.gitignore*:

```
secret.txt
```

So, *.gitignore* is simply a list of files and folders that you don't want git to look at. If you now run `git status`, you will see that *secret.txt* is not mentioned at all. However, new file *.gitignore* is now detected by git, which is completely normal. Now, just `git add .gitignore` and commit using `git commit -m "Added .gitignore"`.

Similarly, you can make git ignore entire folders, for example when you have a dataset in your project folder and you don't want it to be uploaded to the remote server. Simply, to hide a folder called *secret_folder*, in *.gitignore* add a line like:

```
secret_folder/
```

To hide a specific file within the folder, but not the folder itself:

```
folder/secret_file.txt
```

Switching to previous versions

Check a certain version of your repository using `git checkout`

You can switch to a certain snapshot of your code, just like to a saved game, the same way you switched branches.
To start, first run `git log`, it will list to you all the commits made in the repository:

```
usr % git log
commit 77350516a6225985ddb87b3a187f44b02a3e5f47 (HEAD -> master)
Author: name <email@gmail.com>
Date:   Fri Mar 12 10:45:54 2021 +0300

    Added user input parser

commit 1d3864f259bf30c91abb799c1146ddc25a4556d3
Author: name <email@gmail.com>
Date:   Wed Mar 10 12:13:21 2021 +0300

    Added .gitignore
```

Here, you can see two commits with their dates and messages, each having a long unique identifier. Just copy the ID of the required commit and run, for example:

```
git checkout 1d3864f259bf30c91abb799c1146ddc25a4556d3
```

This way, your folder and your files will look as they looked in that specific snapshot. This change **is revertible**. You can always go back to the latest snapshot, the same way you just did it.

If you went back in time with `git checkout`, you may see that git warns you about so-called *detached HEAD* state. It basically means that your commits from here won't be saved, unless you create a new branch. This is quite logical, think of the movies on time travelling, where changing the past (committing to past snapshots) creates an alternative timeline (branch).

Reset your repository to a certain version using `git reset --hard [sha]`

If have found that you did some bad coding and want your repository reset to the last commit, `git checkout` won't be enough, because it only shows you another version of your repository, without discarding your latest bad code. To actually discard it , use this command:

```
git reset --hard HEAD
```

As you may see, your folder and your files are now in the state of the latest commit. If you want to go further back in time, use:

```
git reset --hard HEAD~
```

where `~` stands for the parent of the latest commit. You may repeat this operation *n* times to go *n* commits back.

Attention: `git reset --hard HEAD` will discard your latest changes *irreversibly*. Running this command is actually a rare case, and you probably won't use it in the nearest future. But if you do, use it with caution. In case when you did mess up your local repository, *don't* `git push`. Instead, `git pull` its intact version from the remote server.

Homework

1. Fork this repository on GitHub and get a local clone of it on your computer. The following tasks should be done locally on your computer.
2. Make a hotfix:
   a. Create and switch to branch 'hotfix', fix the coding error in file 'main.py';
   b. Merge 'hotfix' to 'master';
3. Implement a feature:
   a. From 'master', create a new branch called 'feature';
   b. In branch 'feature', add a function that solves the problem described in the main.py. Test it and commit.
   c. Merge 'feature' to master.
4. Create file 'ignore.me' and add it to .gitignore. Make sure it's properly ignored by running `git status`, then `git add .gitignore`, then `git commit -m`.
5. Push your local repo to Github.
6. Send me a link to your repository via personal channel in Slack by 20/03.

Literature

Official git tutorial and some of its chapters:

- Branching
- Reset Demystified

A short story on git wisdom

This is a short ironic story on git commands.

`A UNIX programmer was working in the cubicle farms. As she saw Master Git traveling down the path, she ran to meet him.`

`"It is an honor to meet you, Master Git!" she said. "I have been studying the UNIX way of designing programs that each do one thing well. Surely I can learn much from you."`

`"Surely," replied Master Git.`

`"How should I change to a different branch?" asked the programmer.`

`"Use git checkout."`

`"And how should I create a branch?"`

`"Use git checkout."`

`"And how should I update the contents of a single file in my working directory, without involving branches at all?"`

`"Use git checkout."`

`After this third answer, the programmer was enlightened.`