# **1. What is Git?**

**Version Control System (VCS)**: This is a program that helps developers track changes in the project code. It keeps a history of all changes and allows developers to easily revert to previous versions of the code.

To designate version control systems, not only the abbreviation VCS is used, but also **SCM** (**Source Control Management** — "source code management system").

One change or group of changes in the VCS is called a **revision or version**. Each such revision contains information about what has changed, who made the changes, when it was and sometimes comments on the change.

**Versions**: VCS allows developers to create different versions of code. This is useful when a project is under active development and you need to track and compare different stages of work.

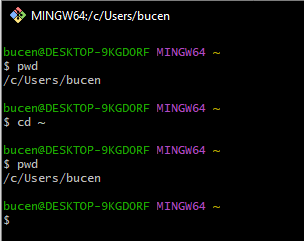
**Control**: VCS helps developers to control the development process. It allows them to see what changes were made to the code, when and by whom. This helps to avoid conflicts and confusion in the work of the team.

# **2. Getting to know the Command line**

In the command line, you are also always in some folder - it's just not visible. To find out where you are now, the **pwd** team will help (**print working directory** — "show the working folder"). It outputs the path to the current directory

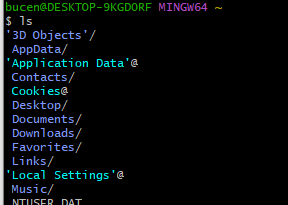
Using the terminal, you can always go to the home directory. To do this, enter the **cd** command (**change directory** — "change directory") and the **~ symbol** — the designation of the home directory. Don't forget to Enter to execute.

* Console and graphical interfaces are two ways to interact with programs.
* A console, terminal, or command line is a program that reads a user's command and executes it.
* To output the current working directory, you can use the pwd command.
* Most computer users have access to the home directory. To go to it, use the cd ~ command.

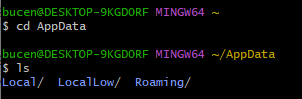


# **3. Command line navigation**

When you open a folder through the graphical interface of the operating system, you immediately see its contents. In the case of the console, the — **ls** command is used to display files and folders (**list directory contents** — "display the contents of the directory").

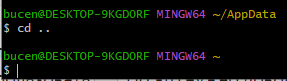


The next important command is **cd** (**change directory** — "change directory"). You met her in the last lesson. It changes the current working directory to the one specified as the parameter: cd folder name.



Please note: if there are spaces in the folder name, you need to use **quotation marks(“”)** when entering.

To return to the parent directory — that is, to a higher level — instead of the folder name, you need to write two dots: **..**



**CD** also allows you to navigate through several directories at once. To do this, you need to separate their names with a **/ sign.**

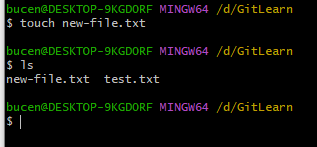
Many console commands have additional options. For example, if you call **ls**, you will see a list of regular files in the directory. But you can call ls with the **-a flag** and output an extended list. It will display all hidden files that start with the symbol . (for example, configuration files). Including two special files . and .., which denote the current and parent directories.

* change directories with the **cd command;**
* output the contents of directories **using ls**;
* view the contents along with hidden files and folders **via ls -a.**

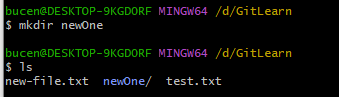
# **4. Operations with folders and files: create, copy, move**

Creating files and directories — **touch, mkdir**

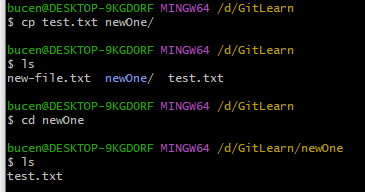
To create a file, you need to enter the **touch command** in the console. With the file name as a parameter: **touch %FILE\_NAME%**



To create directories through the terminal, another command is used — **mkdir** (**make directory** — "create a directory").

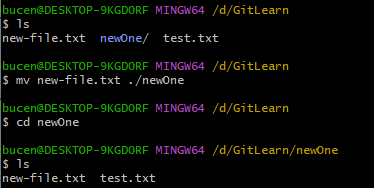


To copy files through the terminal, there is a **cp command** ("copy"). In a simple form, **cp takes two parameters: what we copy and where we copy**

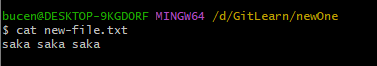
****

Copying creates a copy of a file or folder. But sometimes, instead of a copy, you need to delete a file in one place and create it in another. To do this, there is the **mv command** ("move").

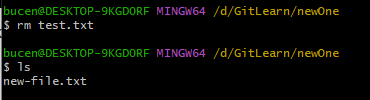
The syntax of the mv command is similar to that of cp. **After the name of the command, specify the list of files and folders to be moved, and then the folder to which you want to move.**

****

To read the file, you need to enter **cat** (**concatenate and print** — "combine and print") into the console along with the file name. The command will print out what is contained in it.



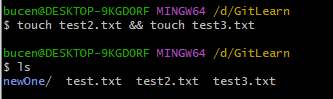
To delete a file, you need to type the **rm command** (**remove** — "delete") and pass it the file name.



You can delete a folder with the **rmdir** command (**remove directory** — "delete directory"). Don't forget to specify the folder name.

# **5. Effective work with the command line**

Commands in the terminal do not have to be typed in and executed in turn. They can be specified not one by one, but in a list at once. To do this, they need to be separated by two ampersands (**&&**).



It is not necessary to memorize all the commands by heart. If you need to find any of them, it's enough to remember which letters it starts with. You can type them in the command line and press **the Tab key twice**. The terminal will show a list of all commands that start with these characters.

# **\* Cheat sheet. Basic commands in the console**

***1) Navigation***

* **pwd** ( print working directory, "show working folder") — show me which folder I'm in;
* **ls** ( list directory contents, "display directory contents") — show files and folders in the current folder;
* **ls -a** — show also hidden files and folders whose names begin with the symbol .;
* **cd** first-project ( change directory, "change directory") — go to the first-project folder;
* **cd** first-project/html — go to the html folder, which is located in the folder first-project;
* **cd ..** — go to a higher level, to the parent folder;
* **cd ~** — go to the home directory (/Users/Username);
* **cd /** — go to the root directory.

***2) Working with files and folders***

**Creation**

* **touch** index.html — create a file index.html in the current folder;
* **touch** index.html style.css script.js — if you need to create several files at once, you can print their names in one line separated by a space;
* **mkdir** second-project (make directory, "create a directory") — create a folder named second-project in the current folder.

**Copying and moving**

* **cp** file.txt ~/my-dir ("copy") — copy the file to another location;
* **mv** file.txt ~/my-dir ("move") — move a file or folder to another location.

**Reading**

* **cat** file.txt (concatenate and print, "combine and print") — print the contents of a text file file.txt.

**Removing**

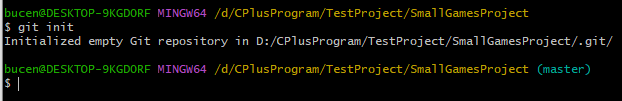
* **rm** about.html (remove, "delete") — delete the file about.html;
* **rmdir** images (remove directory, "delete directory") — delete the images folder;
* **rm -r** second-project (remove + recursive) — delete the second-project folder and everything it contains.

# **6. Let’s start with Git!**

Git is a version control system that helps track changes in a project. This tool can be used for both individual and team work.

Git allows you to save changes locally and, if necessary, return to previous versions of the project. You can also create a remote copy on a hosting platform that works with Git and share the result with others.

In order for Git to start tracking changes in the project, the folder with the files of this project must be made a Git repository. To do this, move to it and enter the **git init command** (initialize)

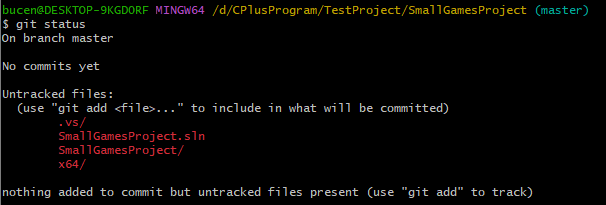


"Unload" the folder if something went wrong **rm -rf .git**

If you accidentally made the wrong folder as a Git repository, you can "unload" it. To do this, delete the hidden subfolder .git.

* **the -r key** (recursive) allows you to delete folders along with their contents;
* **the -f key** (force)will save you from questions like "Are you sure you want to delete this file? And this one? And this one too?".

Check repository status — **git status**. After initializing the first-project repository, run the **git status command** — it shows the current state of the repository.



The git status command will output:

* **the name of the current branch**: On branch master or On branch main;
* **a message that there are no commits in the repository y**et: No commits yet;
* a message that says: "in order to commit something (that is, commit), you must first create it" - nothing to commit (**create/copy files and use "git add" to track**).

**Prepare files for saving — git add**

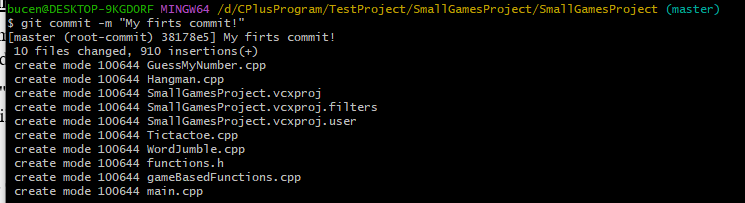
The **untracked** state means that Git does not yet store information about the versions of the file and cannot track how it has changed.

If you edit any of the "green" files in the first-project folder now, it will go into the **modified state** and will be in both the **"green" and "red"** lists.

The git add command does not save the contents of files in the repository**. Saving itself, or fixing the state of files, is called a commit**. "Commit" means to save the current version of the file.

You can make a commit with the git **commit command with the -m** key (message) which assigns a message to the commit.

Usually, such a message explains exactly what the changes were. It's like notes in the margins: thanks to them, it's easier to read and understand the text. The commit message performs the same functions — it improves understanding and simplifies navigation. It is written after the -m key in quotation marks.

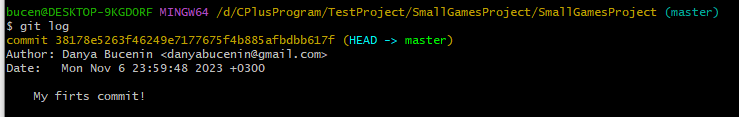


The git commit command will output the commit information.

* [master (root-commit) baa3b6e] means:
  + the commit was in the **master branch**;
  + root-commit **is the very first, or "root"**, commit in the branch, the following commits will not have such an inscription;
  + **baa3b6e is an abbreviated commit identifier** (we'll tell you more about this later).
* 10 files changed, 910 insertion(+) means:
  + 10 files have changed;
  + Nine hundred ten line was added
* Lines like create mode 100644 functions.h — this is more detailed information about new (added to Git) files.
  + create says that the file has been created. If the file was deleted, the word delete would be in this place. "delete").
  + mode 100644 reports that this is a regular file. There are also possible variants of 100755 for executable files (for example, anything.exe) and 120000 for link files in Linux. Link files do not contain data by themselves, but only link to other files — like "shortcuts" in Windows.

By the way, **my first commit!** — still not the best version of the message. The commit should be described in such a way that it is clear exactly what changes were made. For example: An important case has been added to TODO, Sorting of names has been added, an error in the loop has been fixed, or advertising text headers have been added.

In the independent task of the last lesson, you made three commits to your repository. To see them all, enter the command **git log** (log — "log [records]").



# **7. Getting to know GitHub**

GitHub is a platform for storing IT projects and collaborating on them using Git. In fact, this is a site where you can upload your project files for sharing with other people. Indeed, GitHub has become the most popular site for storing Git repositories. Many large companies, such as Google, Apple, Valve, use GitHub for their projects.

* GitHub is a platform that works with Git and simplifies team interaction.
* In addition to GitHub, there are other similar platforms, such as Github, Bitbucket, and so on.
* Git is a console tool for working with local and remote repositories. It is not directly linked to any of the platforms and develops separately from them.

## **7.1 What is SSH. Generating an SSH key**

One of the most common network protocols is SSH (from the English Secure Shell Protocol). It provides secure data exchange on the network. Using this protocol, you can receive data from a remote computer or send it to it. The traffic is encrypted, so the protocol is secure.

SSH uses a key pair for security — public and private:

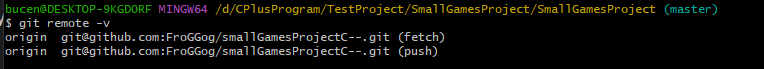
* **The private key** is stored only on your computer and should not be transferred to anyone else. It is used to decrypt data.
* **The public key** is accessible to everyone and is used to encrypt data. They can be decrypted with a paired private key.

Only you can decrypt the data using a private key, but any owner of a public key can encrypt it for you. These two keys are linked and form an **SSH pair**. In the future, you will probably use them to interact with GitHub and other remote servers.

## **7.2 Bind a remote repository to a local one — git remote add**



**origin** ("source") is a standard alias with which you can access the main remote repository (usually there is one such repository). This greatly simplifies the work.

Flag **-v** is the short form of the flag **--verbose** ("detailed"). It allows you to show more information in the output.

When you link the local and remote repository, you will be able to "**pull" (fetch)** changes from the remote repository to your local one, and **"push"** your changes back to the remote repository. This will allow all project developers to work together and see each other's changes.

## **7.3 Synchronizes local and remote repositories**

**Main branch**

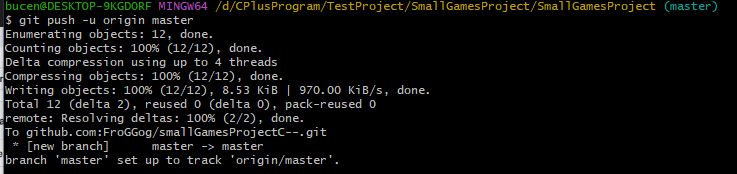
We mentioned that each commit keeps the files up-to-date. The comments themselves are stored in branches.

**If a commit is a snapshot of the status of files**, then **the branch is the timeline on which these snapshots are located**. The branch always starts from one of the commits. There may be several branches in the repository at once — parallel change histories. They can also connect to each other.

**Send changes to a remote repository — git push**

You have already gone through the entire "commit cycle": prepared the files using git add, committed them with a comment with the **git commit -m command**. It remains to upload the contents of the local repository to GitHub. The git push command is responsible for this.

For the first time, this command must be called with the **-u flag** and the parameters origin (the name of the remote repository) and main or master (the name of the current branch). The -u flag will link the local branch with the remote branch of the same name



## **7.4 File README.md**

In order for other users, as well as potential clients or employers, to understand what the project is, it needs to be described. It is customary to specify such a description in the file **README.md**

As a rule, in README.md you can find the following information about the project:

* **The name of the project and its brief description**: the cream was created for what, which solves the tasks and which closes the problems.
* **Technologies that are used in the project**. What is its difference from similar ones.
* **Project documentation** — detailed instructions on what the project is.
* **Project plans**, if any.

**How to create and issue README.md**

README.md — a text file that can be created with the touch command and then edited in the same way as any other text document. For example, in notepad.

Advantage README.md the fact is that team work tools (such as GitHub) can display its contents in the browser in the form of convenient markup. To do this, you need to not just fill in the text, but also adjust the font, headers and margins using markdown. **Markdown is a special markup language**. It allows you to format a text document beautifully.

## **7.5 Headings, paragraphs and hyphenation**

### **7.5.1 Headers of different levels are created by grids.**

# H1 — the first level header, the largest

## H2 — second level header, smaller

### H3

#### H4

##### H5

###### H6 — the sixth level header, the smallest

### **7.5.2 You can add a line under the title or paragraph.**

#### Heading 4

Text above the line

---

Text under the line

### **7.5.3 To make a line break, you need to put two spaces (in the example below they are indicated by dots ⋅⋅) or a combination of <br> characters.**

Text before the transfer⋅⋅

The text after the transfer <br>

Text after the second transfer

### **7.5.4 To start a new paragraph, there should be two hyphenation characters at the end of the previous line. To do this, press Enter twice.**

Line

another line

## **7.6 Text Selection**

### **7.6.1 To highlight the text in italics (\*text\*), it is enclosed in asterisks (asterisks) or underscores.**

Italics are \*asterisks\* or \_contraction\_.

### **7.6.2 To highlight the text in bold (\*\*text\*\*), it is surrounded by double asterisks or double underscores.**

Bold font — double \*\*asterisks\*\* or double \_\_underscores\_\_.

You can combine the selection of \*\*asterisks and \_contraction\_\*\*.

### **7.6.3 To cross out the text (~~text~~), it is surrounded by double wavy lines — tildes.**

~~Strikethrough text.~~

## **7.7 Lists**

### **7.7.1 To make a numbered list, it is enough to put numbers with a dot at the beginning of the line.**

1. The first item of the numbered list.

2. The second point.

### **7.7.2 An unnumbered list is created by an asterisk with a space at the beginning of the line or a hyphen with a space.**

\* the first item of the unnumbered list;

\* the second item of the unnumbered list

- the first item of the unnumbered list;

## **7.8 Links**

To make a part of the text a link, enclose it in square brackets, and then specify the desired address in parentheses.

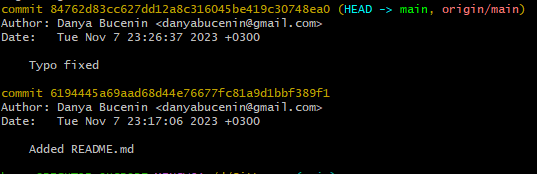
[Yandex](https://www.yandex.ru )

You can also add a title to the link. A title is a tooltip that appears when the mouse hovers over a link. The title should be enclosed in quotation marks and specified inside brackets after the address.

[Yandex](https://www.yandex.ru "I'm Yandex!")

# **8. Hash ID of the commit**

In the process of working with Git, you will often encounter the concept of a "hash commit". You could see these strange lines with a meaningless (at first glance) set of letters and numbers when you called the git log command and output the commit history.



(this yellow)

## **8.1 What is a hash. Hashing commits**

**Hashing ( "hack", "crumble", "hash") is a way to transform a set of data and get their "fingerprint".**

Commit information is a set of data: when the commit was made, the contents of the files in the repository at the time of the commit and a link to the previous, or parent, commit.

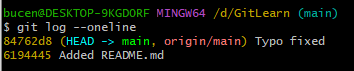
Git hashes (converts) commit information using the SHA-1 algorithm (from the English Secure Hash Algorithm — "secure hashing algorithm") and receives for each commit its own unique hash result of hashing.

Usually a hash is a short (40 characters in the case of SHA—1) string, which consists of the digits 0-9 and the Latin letters A—F (it does not matter whether uppercase or lowercase). It has the following important properties:

* if the hash is obtained twice for the same set of input data, the result is guaranteed to be the same;
* if at least something in the source data changes (at least one character), then the hash will also change (and greatly).

Git stores a table of **matches hash → commit information**. If you know the hash, you can find out everything else: the author and date of the commit and the contents of the committed files. We can say that the hash is the main identifier of the commit.

You can get a shortened log using the git log command with the **--oneline flag** ("one line"). Only the first few characters of the hash of each commit and their comments will appear in the terminal.



# **9. File statuses in Git**

Untracked/tracked, staged and modified statuses

One of the key tasks of Git is to track changes to files in the repository. To do this, each file is marked with a status. Let's consider the main ones.

* **untracked**: We said that new files in the Git repository are marked as untracked, that is, untraceable. Git "sees" that such a file exists, but does not monitor changes in it. The untracked file has no previous versions fixed in the comments or via the git add command.
* **staged** ("prepared") After executing the **git add command**, the file gets into the staging area (from the English stage — "stage", "stage [of the process]" and area — "area"), that is, in the list of files that will be included in the commit. At this moment, the file is in the staged state.
* **tracked**: The tracked state is the opposite of **untracked**. It is quite broad in meaning: it includes files that have already been committed using git commit, as well as files that have been added to the staging area by the git add command. That is, all files in which Git tracks changes in one way or another.
* **modified**: The modified state means that Git compared the contents of the file with the last saved version and found differences. For example, the file was compromised and then changed.

# **Standard Conventional Commits**

"commit agreement") it is distinguished by high-quality documentation and detailed study. It is suitable for repositories with the source code of programs. It would be inconvenient to use it for other types of projects (for example, to translate a book).

Conventional Commits offers the following commit format: <type>: <message>. The first part of type is the type of changes. There are quite a lot of such types. Here are two examples:

* feat (feature) — for new functionality;
* fix — for fixed errors.

[https://www.conventionalcommits.org/ru/v1.0.0-beta.4/#спецификация](https://www.conventionalcommits.org/ru/v1.0.0-beta.4/" \l "спецификация)

# **How to fix a commit**

Sometimes you need to change something in a newly executed commit: for example, add a couple more files or replace the message with a more informative one.

In this case, you can make edits to an already made commit using the **--amend** option (amend — "fix", "supplement") the commit command has: git commit --amend. Let's analyze how it works.

# **How to roll back if "everything is broken"**

At different stages of working with Git, similar situations may occur:

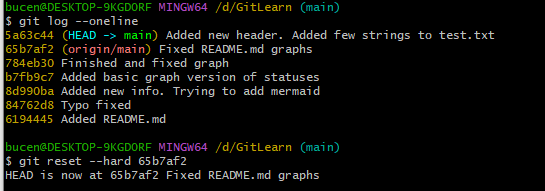
* An extra file (for example, a temporary one) got into the commit list. You need to "remove" it from the list.
* The last few commits are erroneous: for example, they did not do what was needed, or violated logic. I want to "roll back" several commits at once, return "as it was yesterday".
* Accidentally changed a file that shouldn't have changed at all. For example, you opened the wrong file in the editor and started to fix it.

## **Perform unstage of changes — git restore --staged <file>**

Let's say you created or modified some file and added it to the "on commit" list (staging area) using git add, but then changed your mind about including it there. To remove a file from staging, the git restore --staged <file> command will help.

## **"Roll back" commit — git reset --hard <commit hash>**

Sometimes you need to "roll back" what has already been committed, that is, return the repository state to an earlier one. To do this, use the **git reset --hard <commit hash> command**

****

## **"Roll back" changes that were not included in either staging or commit — git restore <file>**

It may be that you accidentally changed a file that you didn't plan on. It is now displayed in Changes not staged for commit (modified). To return everything "as it was", you can run the **git restore <file>** command.

# **Ignoring files in Git**

In order for Git to ignore such files and not try to add them to the repository, you need to create a file **.gitignore** and write the names of ignored files to it. In this lesson, we will analyze how to do this.

If the line starts with **#**, then it is a comment, and **.gitignore** will not take it into account.

Let's say you need Git to ignore all files **.DS\_Store**. To do this, it is enough to add in .gitignore a string with the file name.

**Asterisk (\*)**

The asterisk character (\*) matches any string, including an empty one. If such a symbol is used in the template in .gitignore means that the file will be ignored regardless of what will be in place of the asterisk.

**Question mark (?)**

Question mark ? matches one of any character.

file?.txt

If you save such an entry in .gitignore, then, for example, files will be ignored fileA.txt and file1.txt . And here is the file file12.txt will not be ignored because there are two characters after file in its name, not one.

**Square brackets ([...])**

Square brackets, like the question mark, correspond to a single character. In this case, the symbol is not any, but only from the list that is indicated in parentheses.

# **Clone repository — git clone**

The **git clone** command automatically links the local and remote repository. That is, if something changes in the GitHub repository (for example, commits are added), you will not need to clone it again. It will be enough to execute a command that will update your copy.

# **Executing a Fork**

Let's say you want to improve someone else's project or somehow use it in your work, but you don't have the rights to change the original repository.

We will analyze another useful operation of copying projects. Unlike cloning, it will not download the repository to a local computer, but will add it directly to your account on the GitHub server.

**What is a Fork**

Fork is a GitHub operation; it is not directly related to Git. The fork creates a copy of the repository in the GitHub account. Such a copy will be completely independent. The changes you make will not be synchronized with the original repository.

During the "fork" process, a copy of all files, commit history and branches is created. This copy is saved in your GitHub account.

Here are some of the common reasons for using "forks":

* You want to contribute to the project (for example, open source), but you do not have the rights to change the source repository. Then you can make a "fork", add the necessary edits, and then send a request to include these changes in the original project.
* You want to develop the project independently of the original one. Let's say the creators of the project decided that they would not add the functionality that you need. In this case, you can make a "fork" and add it yourself.

# **Why do we need branches**

**A branch is an isolated project development thread**. In such a flow, you can test different ideas, test new functionality, and so on.

Branches allow you to experiment with a project in Git, but at the same time keep the repository in a stable state. Each team member can work in his own branch and not interfere with others: the commits he makes will not be visible from other branches. And when the work is done, the branches can be connected.

Branches are useful even if you are working alone — for example, on a website. Before writing a new functionality, you should create a separate branch for it. Branches also allow one person to switch between several tasks at once.

## **View project branches — git branch**

## **Create a branch — git branch <branch name>**

The name of a branch in Git can consist of letters, numbers, and also include any of the four characters: ., -, \_, /. These symbols do not carry much meaning. For example, the feature/add-branch-info branch could be called feature\_add-branch-info or feature-add-branch. Note that branches do not form hierarchies like directories separated by the / symbol.

## **Switch to another branch — git checkout <branch name>**

## **Navigation suffix ~**

Comparing commit hashes can be inconvenient, because there can be many of them in one branch. Imagine: first you output the history via git log, then you search in a long list for the hashes of those commits that you want to compare, and only then you perform git diff.

To facilitate this task, Git has a navigation suffix ~N, where N is a number. It counts N commits back in time from a given commit. Numbering starts from zero: commit~0 is the commit itself, commit~1 is the previous one, commit~2 is the previous one, and so on.

For example, HEAD~1 is the next commit after the current one. And main~5 is the fifth commit in the main branch, if we count from the last commit executed.

In practice, either the current commit (HEAD) or the one following it (HEAD~1) is more often needed. For ~1 there is a special abbreviation ~ (without a number). That is, instead of HEAD~1, they usually write just HEAD~.

# **Merge and delete branches**

Imagine that you have finished developing new functionality in a separate branch and are ready to combine it with the main one — add your changes to the main version of the project. This process is called merging branches.

In this lesson, we will analyze how to “merge” branches, that is, combine them. And also, how to delete unnecessary branches from the repository.

Perform a merge — **git merge <branch name>**

# **Delete a branch after merging — git branch -D <branch name>**

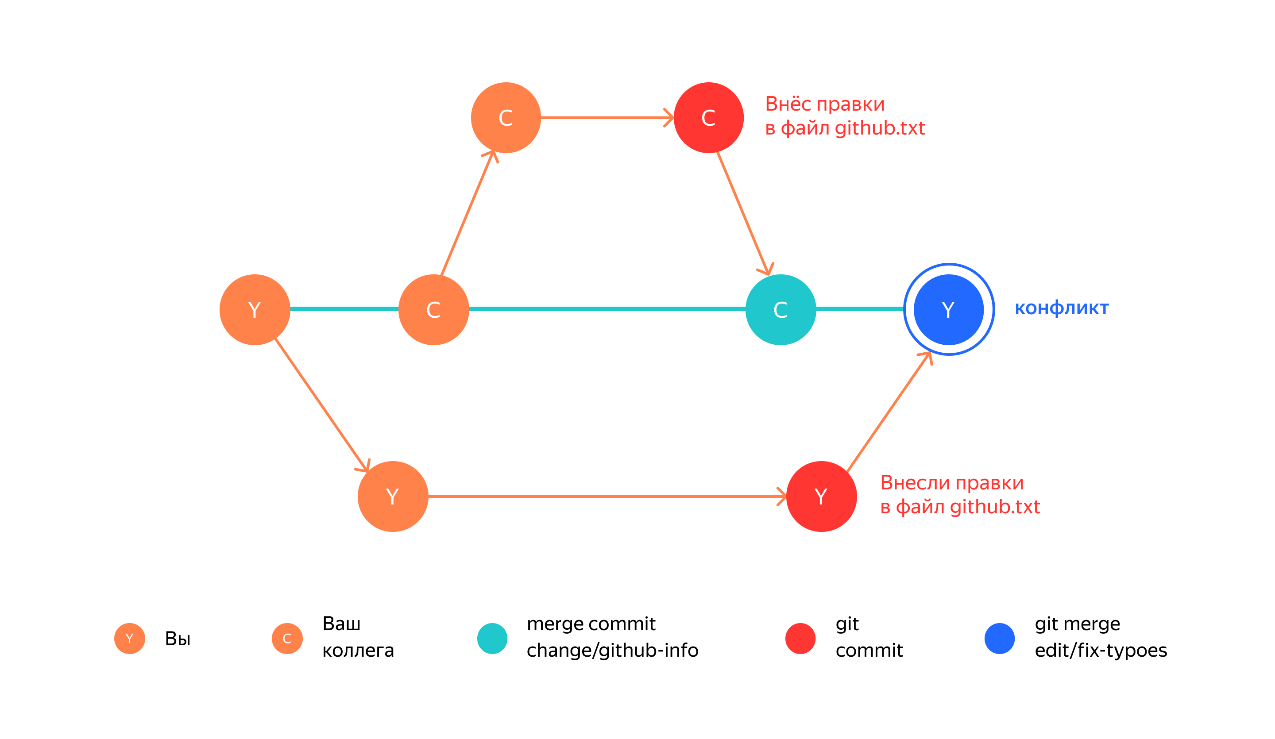
After the merge has occurred, the donor branch can be deleted. To do this, in the main branch, enter the git branch command with the -D flag ("delete") and the name of the branch.

The git branch **-D** command has a safer option with the **-d** flag. It will delete a branch only if it was completely merged with another one — that is, if two branches became (or were originally) part of the same story. For example, if you accidentally created a branch with the wrong name, you can delete it via git branch -d %branchname%.

# **What is a conflict**

When several team members are working on the same project fragment in different branches at once, conflicts may occur during the merge. Let's look at how it happens and what to do in such a situation.

Let's say you decide to edit a paragraph of the text and correct typos. At that moment, your colleague changed the same paragraph in the neighboring change/github-info branch and poured the changes into main.



# **Send a local branch to a remote repository — git push**

git push -u origin %branch\_name%

# **Pick up changes from a remote repository — git pull**