If you create websites, web applications or mobile apps, or in general if you write any type of code, then it's always good to use a version control system. This course will teach you about one of the most popular and widely used version control systems today. Welcome to The Complete Git & GitHub course. Hi, I'm George and this is Lasha. We are the creators and designers of this course, and we represent CodeAndCreate. Nowadays, any of the professional developer or designer needs to know how to work with Git and GitHub. Git allows you to record different versions of your project and let's you go back in time and check previous states of the project. As for the GitHub, it is an online service where you can share your code to the world and also it allows us to collaborate with different developers. So, if you want to learn those things, then this is the right course for you. Let's talk about the course content. We will start from Git Basics where you can learn about how to install Git. We will go through the basics of the terminal, we'll create our first local repository, and you will see how to save different versions of the project as the snapshots using commits. Next we will move on and learn about how to go back in time and check the previous states of the project, update them or even delete it. Once we get some solid knowledge about Git, then we will start learning about the Basics of GitHub. We will go through the topics such as creating the repository on GitHub, connecting the local repository to a remote one, making commits, and much more. Next we will move on and see how to work with one of the greatest features of Git, which is branching. We will learn about how to deal with branches locally and also remotely. After that, we will refer to kind of complex stuff. We will show you how to merge branches with different methods and also how to deal with merge conflicts. We will go through those topics both in Git and GitHub as well. Next it should be followed by Forking and Contributing. You will learn about how to copy someone else's remote repository on your GitHub account, how to clone it to your local computer, and how to contribute to the original project. Finally, we will discuss how to collaborate on GitHub. We will go through some really important topics such as how to set up a collaboration team, how to define different rules for collaborators, and much other stuff. So, join us, see you inside. [Outro music]

[No audio] Alright. So before we start to discuss any specific topic regarding Git, the first question that comes to mind, and that we have to answer is the following, What is Git? So as the definition, we can say that Git is a Distributed Version Control System. Most of you who is not familiar with Git yet, those words are not quite clear. So let's just first talk about what version control itself is. Version control, or maybe sometimes you may hear source control, is a system that records changes of your file or entire project, and then you're able to recall any specific version of your project later. In other words, the version control system allows you to go back in time and get previous states of your project. Also you can compare changes over time. Alright, let's use an example to make even clearer what version control system is, and why should we use it. Suppose that you're working on a project, I mean, you're building a website, you have done a couple of things and you already have your first version of the website. Then suppose that you want to change something, and also you don't want to lose your first version of the project. So, what can you do? The first thing that comes to mind is to save the first version, then make a copy of it and work on the second version. Then suppose you need to make another copy, then another, and so on. Finally, you might have lots of different versions of the project, and also your hard drive might be full. You agree that it's not the perfect and efficient way when you work, especially on large projects. So, in order to avoid such painful and insufficient way, the version control system was created. So it allows you to make snapshots of the current state of your project, which means that it saves all of those different versions of the project in just one directory. Nowadays, there are a couple of different version control systems, but the most popular one, which is widely used today, is Git. Those days when web development is extremely fast-growing industry in the world, and you have to be a good and professional web developer, then you should definitely know how to work with Git. Because you might be working on different projects at the same time and without a version control system, your working process might be hard and painful. Also one of the advantages of using Git is collaboration. It means that different developers can work on the same project. For that today we use mostly GitHub, which is online service. You can host your project on GitHub. It is a shareable place, and everyone can work on that project. Regarding GitHub, we will discuss it later in the course. At the beginning of this video, when we defined what Git is, we said that it's a distributed version control system. In general, there are two types of version control systems, Centralized and Distributed or Decentralized. Centralized version control systems have a single server that contains all the different versions of projects, so that developers can collaborate, but actually, this type of version control has some downsides. If you do not have network connection, then you won't be able to get access on versions. And also, if for some reason, server corrupts, then you will lose all your data. But in case of distributed version control systems, you save versions of your projects locally, on different computers, and also on server, and each copy of project is a full backup of all your data. So if something happens with server, then you still have the full backup of entire project locally. And also you don't need to have network connection in order to save versions of your project. Alright. So that was a little introduction of version control systems and actually Git. For now it might not be quite clear for you, especially when you're not familiar with Git, but it's not a problem. Step by step, you will get solid knowledge and experience in it. Okay, so, bear with me and move on to the next lecture. [No audio]

[No audio] Alright. So in the previous lecture, we had a brief overview about what Git and generally what version control system is. And before we dive deeper into any particular topic regarding Git, I'm going to talk a little about how Git actually works. In order to use Git for your project, first of all, you have to initialize it. Let's say we have a folder called My-Website, in which we store a couple of files, index.html style.css, and also we have a folder for images. So after initialization of Git, there will be created a folder named .git. Actually, how to initialize Git, you will see it in practice in the coming lectures. For now I just want to show you the idea how the process is going, and how Git works. So, once this folder is created, then the entire directory where that folder is placed will become a repository. Sometimes it's called just repo. So the repository is a container for a project, which you want to track with Git. Alright, so after that, we are able to track all changes of our project. It doesn't matter if we change something inside those two files, or inside images folder. All the changes will be recognized by Git. By the way, as a quick note, I want to tell you that if git initialization happened inside images folder, then changes outside of that folder would not be tracked. Because in that case, the repository is going to be images folder, and not the entire project. Okay, let's consider again the previous example. Suppose that we added a new file to our project, let's say script.js, and we want to save this version of our project in history. We can do that using commit, which actually allows you to take a snapshot and save the current state of the project. In order to make it clear, let's consider some examples using a little diagram. Let's say this line represents History, or sometimes it's called a Timeline of the development of the project. So imagine that before we take any snapshots or commit something, we are at the starting point. After doing some work, suppose that we want to take a snapshot and save a current state of the project. For that we have to make a commit. After that, let's say we added some JavaScript to our project and made the website dynamic and created another commit. Now we have already saved two versions of our project, then assume that we added some backend functionality to our project, and took another snapshot. So, overall, we created the history of the development of the project, and anytime in the future, we're able to go back in time and get a previous date of the project we want. Now, I showed just a couple of examples, but in practice, in the real world, you can make as many snapshots as you wish. Again, how to make commits, you will see it in details in coming videos. One little thing that I want to mention before we finish this lecture is that this line is called the Branch. To be more precise, it is a master branch. In general, you're able to work on different branches, but regarding that, let's discuss later on. Alright. See you in the next video in which we're going to set up our Git working environment. [No audio]

[No audio] Okay, so it's time to set up the tools, I mean to install Git and actually prepare our working environment. Throughout the course we will need terminal and text editor. I'm going to use VS code as the text editor. In order to download VS code, you have to visit a website called code.visualstudio.com, and then download the installer from here. The installation process is really simple, so I'm not going to go through it. As for terminal, we're going to use Git Bash, which you can get automatically once you install Git itself. In order to install Git, we have to go to a website called git-scm.com. Then we have to go to Downloads, and we have to select the operating system which we're currently using. Besides that, on the right side, you can see here Download button, which automatically finds out which operating system we are using, and it allows us to download the proper installer. For now we use Windows. So, that's why you see here, Download for Windows. If you're using a Mac, then instead of Windows you will see here Mac and the same for Linux. I will show you how to install Git on Windows, and if you're using different operating system, and if you have some trouble with the installation process, which I think won't happen, feel free to reach out in QA section and we will try to help. Okay, let's click here and download the installer. [No audio] Once it is downloaded, let's go ahead and start the installation process. We have to click here Next, then we need to select the location of Git folder. I'm going to leave it as it is. So click Next. Then check the first box. It will allow us to create the icon, On the Desktop. Then again Next. Now we have to select the default editor for Git. As I already said, throughout the course we're going to use VS Code. So I'm going to select it, but if you have your favorite code editor, you can go ahead and select it. So here Git offers us to work by default from the command line and also from 3rd-party software, so I'm not going to change it. Then click Next again. Next, Next, and then finally click the Install button. Okay, it will take a little amount of time. [No audio] Once Git is installed, let's click the icon. As you see, Git Bash is opened. It means that Git is successfully installed. And if we check its version using the command, git --version, then it will give us the current version of Git. Alright, that's it. We are ready to start working with Git. So let's move on to the next lecture. [No audio]

[No audio] Alright. So once we introduced to some basic terminal commands, finally it's time to create our very first git repository. First of all, let's open Git Bash and navigate to the Desktop using cd Desktop. I'm going to create a new folder. For that we need mkdir. Let's call this folder gitProject. [No audio] Now I want to open this folder with VS code. We can either open it manually by right click, like so, or we can use the terminal. For that we need command code and the name of the folder, gitProject. [No audio] So, as you can see, our folder is opened with VS code. By the way, if you're inside the folder, and you want to open it with VS code, then you just need to type code space and dot. Alright. Let's place editor and terminal side by side in order to make our working process more convenient. [No audio] Next, I'm going to create a new file. We can do that from the terminal. Let's enter the folder, write cd gitProject and create a new file using touch and indicate name of the file, index.html. [No audio] Also we are able to create a new file from the editor. We can either click the icon here, or we can create a new file using right click and then Create File. Let's call this file style.css. [No audio] Alright. So our project consists of two different files, but at this point it's not a git repository. As you remember from previous lectures, the first thing that we have to do in order to create git repository is git initialization. For that we need to use our very first Git command, git init. [No audio] So, as you can see, there is created a new folder with the name .git, and also our files have changed their color in our editor, for now they are green. If you don't see .git folder in VS code by default, then you should fix it from the settings. So you need to go to Settings, then you have to click curly braces, and then JSON file will be opened. From here we can add the following code which will allow you to display .git folder. So let's go ahead and insert it in your settings file. If for some reason for there still is not displayed, then just restart your VS code and it should be displayed. Okay. If we check our gitProject from File Explorer, then the newly-created git folder will be hidden by default. In order to make it visible, we can go to View and then check Hidden items. So now you can see that the folder is visible. Alright, let's go back to VS code. If we drop down the git folder, we will find here some other folders and files. This is the source of Git, but we are not going to go through them because we are not interested in them. We just need to learn about how to use Git and how to work with it. Alright. We said that after git initialization files have changed their color, but we didn't say why this happened. So now our files are untracked. Basically, untracked means that git sees the files that we didn't have in the previous snapshot. For now, we don't have any snapshots because we didn't make commits yet. The only step that we went through was git initialization. Also you can see on the right side of file names the letter U, which actually means that those files are untracked. Besides that, in order to prove that those files are untracked, we can use another Git command, which actually will be used frequently throughout the course, and this command is git status. It gives us some useful information. It says that we are on the master branch, we don't have any commits yet, and also it says that we have some untracked files, index.html and style.css. You can notice that at this point those files have a red color, but once we go further, they will change their color in some cases. Alright. That's it about creating the first git repository. Before we finish this lecture, I want to show you one more thing. We have already used some git commands, we will be using other git commands as well throughout the course. And if you need to get some more information about Git commands, you can use GitHub. For that you need to specify a command that you want to get information about, let's say git init, and then we need --help. In case of Windows, a new tab will be opened in the browser, you can read this article. And if you're using Mac or Linux, then this information will be opened inside the terminal. Alright. That's it for now. We have seen how to create git repository, we used some git commands, and now it's time to move on to the next lecture. [No audio]

[No audio] Alright. So, in the previous lecture we created our very first repository, and now in this video I'm going to show you the whole cycle, I mean, beginning from untracked files to making commits. So, there are several phases which File or Project goes through. In general, remember that each file in your working directory can be in one of two states, tracked and untracked. Tracked files are files, which were in the last snapshot. As for untracked files, they are any files that were not in your last snapshot. If we consider our example, I mean, our little project, which we created in the previous lecture, files index.html and style.css are untracked files, because we didn't make any snapshots yet. In order to begin tracking a new file, we use the command git add and then we should specify the name of the file. Once we run this command, then this file will be staged. So in general, when file ends up in the staging area, it means that it's ready to be committed. In order to make commit, we should run the command git commit -m, which allows us to create a message with a little description. So, when you commit the version of the file, at the time you ran git add is what will be stored in the historical snapshot. Alright. You were able to change a file that was already tracked, and when you do that, then it will be modified. So it means that file that was tracked has been modified in the working directory, but it's not yet staged. In other words, in History, there is stored the version of the file, which we added it to the staging area firstly. So now if we want to make a snapshot with a modified version of the file, then we have to stage it again using git add command, and after that the file should be ready to commit. Alright, so this is a whole cycle, and those are the main phases which file goes through. Now it's time to go to our terminal and see all of this stuff in practice. So now we are in our folder, in which we have two untracked files. As you already know, the next step, which we have to do is to add those two files to the staging area. Before that, I'm going to insert some basic stuff into those two files. Let's just create a basic structure of HTML document. For that we need to place ! and press Tab. Also let's open style.css file and just change background color for the body element. Let's make it red. Alright. Now it's time to add those files to the staging area. For that, as I already have shown, we need to use the command git add, and we have to specify names of the files, index.html and style.css. So now those files are staged, and if we check the status using git status, then you will find that files are no longer red, they have changed their color, now they are green. And also we see here the text, No commits yet and Changes to be committed. So it means that those two files are ready to be committed. Also we see here text, which is saying that we can use command git rm --cached in order to unstage the files. So if we run this command for one of the files, [No audio] let's say for index.html, and then check the status again, you will see that index.html is again untracked. So remember that when you want to get rid of the file from the staging area, you need to use the command git rm --cached with the file name. Okay, let's add again index.html to the staging area. Run the command git add index.html. Alright. So now it's time to make our first commit, for that we need git commit, and then we have to create a message, which will have a little description of our snapshot. So we need -m, and then inside quotes we can write some text. In general this text should be short and descriptive, and I'm going to write here, "Initial commit". [No audio] So once we press Enter, then we will get the message saying, Please tell me who you are. And then down below you see that we have to run some commands. Actually it happened because we need some configuration. We have to specify the name of the user and the email. Git requires this information because it saves in History to whom belongs particular comment. In other words, it records which developer made the snapshots. So we have to run git config --global and then user.name. If you want to specify your name with two or more words, then you need to enclose them within quotes, but if you're going to write just name with a single word, then you don't need quotes. Alright. I'm going to write here some random name, let's say John. After that we have to configure email, so we need a similar command. By the way, if you want to use previous commands again, then you don't need to type them over and over again, you just need to press the up arrow key, and you will get the previous commands. Let's change the name and write an email instead of it, and then insert here some dummy email. Let's say john@gmail.com. [No audio] Of course you can use your own email, it's up to you. Alright, so now we can make a commit. But before that, let's make sure that all this configuration stuff is done correctly, I'm going to check the user name and the email. For that we have to run git config --global user.name. [No audio] As you see we got here name John, and the same we can do for email. [No audio] So, as you can see, everything is set up, and, finally, we are able to make the commit. For that we have to write git commit -m, and then the message "Initial commit". [No audio] So now, eventually we made our first commit, and in History there is recorded the current state of the project. When we proceed and continue working on the project, then in the future, every time we'll be able to go back in time and use the state of the project. Actually, you will be able to see all of this stuff throughout the course, and before we move on, I want to talk a little about commits. You may think that once we make a snapshot, then we store the copy of the files or the project, but that's not right. Every time you commit, Git basically takes a picture of what your files look like, at this specific moment, and it stores a reference to that snapshot. That's the huge advantage of using Git. Alright. For now I think it's enough. We have just learned about the phases of the cycle, I mean, what is staging area, modifying the files, also we made our first commit, and now it's time to move on to the next lecture. [No audio]

[No audio] Alright. So in the previous lecture, we made our very first commit, we have learned about how to do that. In this video, I'm going to make some other commits, and I will show you how to check the history of the commits. So first of all, I'm going to check the status of our repository. Let's run git status. As you see, for now we have nothing to commit and our working tree is clean. Let's go ahead and modify slightly our files. I'm going to create a new element inside an HTML document. Again, you don't need to know about how HTML and CSS work. The main thing here is to change and modify the files. So let's create a h1 heading element with some text, let's say, Git is awesome. Then I'm going to add some styles into style.css file, let's select h1 heading element, and change its font-size. Make it 40px. Alright. So we have made some changes in both of the files. Let's check the status. So, as you see, both files are modified. They have red color, and the next thing that we have to do is to add them to the staging area. You already know how to do that. We have to use the command git add. [No audio] Actually, we can either indicate both file names in one command, like so, [No audio] but imagine that we are working with a large project and we have to add 10s of files. Then, typing file names will be really hard and painful. So in order to avoid this, we can use just . instead of the file names, and all the files will be added to the staging area. Let's check again the status. [No audio] So now both files are added to the staging area, and the next thing that we have to do is to make the commit. For that we have to run the command git commit, [No audio] then -m. And now we have to insert the message, let's say "Added heading". [No audio] Alright. So now we have two commits in the history. I'm going to create one more commit, let's create a new file and call it script.js. So, what do you think, what will be the state of the newly-created file, I mean, script.js? Let's go ahead and see it. In order to check that let's run git status. As you see, it's an untracked file because it doesn't exist in the last snapshot. In order to start tracking this file, we had to add it to the staging area. So we have to write git add, and the name of the file, script.js. Now it is staged, and next we have to make a new commit. For that we have to run git commit -m, and as the message, let's write, "Add JS file". [No audio] Alright. So, at this point we already have 3 commits, and now I'm going to show you how we can check the history of our commits. For that we have to run the command git log. [No audio] As you can see, we got here 3 commits, let's go ahead and describe what do we have here. The first thing that we see here is some vague characters, which actually is the unique identifier of the commit. Then we see HEAD master. The HEAD is something like pointer, and actually, we'll discuss it a little bit later. Next we have an Author, which shows us the name and the email of an author of this commit. As you remember, we added those name and email while we were trying to make our first commit, and git has required some configuration. Next comes the date of the commit, and finally we have the message of the commit. As you noticed, the first commit that we see here is actually the last one that we have just made. Okay. Then we have second commit. It shows us the state of the project when we Added heading, and the last one represents our very first Initial commit. Alright, so that's the way how we can check the history of commits. For that we have to use git log command. Actually, there is another way to do the same, git log gives us some bunch of information. And if we want to get this information in kind of concise way, then we can run this command with flag --oneline. Now the information is much shorter. Each commit is shown in one line. The identifier is represented with just the first seven characters, but the number of the characters is enough to use the identifier in different commands. Also, we no longer have here Author and Date, and finally we have here just the message of the commit. So you're able to use both ways. Sometimes when you have lots of commits, then using that shorter way is more convenient. Alright. So that's all about how we can check the history of commits. For that we have to use command git log or it's shorter way, git log --oneline. Okay, see you in the next video. [No audio]

[No audio] Alright. So we have already made a couple of commits, and in the last lecture we have learned about how to check the history of commits using git log commands. In this video, I'm going to show you a little bit different way of making commits. Let's go ahead and create a new element inside an HTML document. Let's create a paragraph with some dummy Lorem ipsum text. Let's save it and check the current status, run git status. So we have modified the index.html file. Let's go ahead and add it to the staging area, write git add and the file name index.html. Let's check again status. So the file is ready to be committed, and now I'm going to show you how we can do that in kind of different way. We can run just git commit, [No audio] and once we press Enter, then the new file will be opened in our text editor. So this file is open in VS code, because we have chosen it as default text editor while we have been installing Git. This file is called COMMIT\_EDITMSG. Inside that file, there are a couple of comments. And now here we can insert the message of the commit. So basically, we are doing the same thing as we do in case of dash and flag, but in this case we use a text editor instead of the terminal. Besides that, you can notice in terminal a little hint saying, Waiting for your editor to close the file. So, once we insert here the message, let's say, Added paragraph, then save and close the file, then the commit will be automatically made, what you see actually here in the terminal. If we check the history using git log --oneline, then we will find that the new commit is added. Alright. Now I'm going to show you what happens if VS code is not set as a default text editor. In order to unset it, we have two different ways. First we can open a file called .git config in our text editor. This is a file which stores some configuration stuff, I'm going to open it and show you. Actually, you don't have to do that, you can just watch this part. So, in order to open the .git config file, we have to run the following command. First we have to indicate the name of the editor. In our case it's code, and then it should be followed by the name of the file. Before that, we have to type here till the sign, it allows us to navigate to the folder of the user. Then we need / and the name of the file, .gitconfig. Alright. So, as you see, the file is open in VS code. At first, I'm going to mention that the location of this file is not our project folder, it is placed inside the folder of the user, because it is a global file, and it has access on every git repository. That's why we use -- global flag with the git config command. Okay, so we have here some stuff, and down below you see that the editor is set as code. And also we have a name and the email of the user, which actually we added during the configuration. We are able to unset default editor from here, and for that we just need to get rid of those two lines, but instead of deleting them manually, I'm going to use an easier and more professional way. We can run the following command, git config --global --unset, and now we have to specify that we want to unset core.editor. So once we press Enter, then you will see that the two lines are automatically deleted from gitconfig file. Now let's make a new commit. For that I'm going to modify one of the files, let's say script.js. Let's create a variable inside it. Write const a = 50. Actually, I'm making little changes just for demonstration. I'm not going to waste time on coding, but in general, in the real world, making comments on such little changes is not recommended. Okay, let's check the status. So you see script.js file is modified, and now we have to add it to the staging area. So, let's run git add script.js. Let's check again status. [No audio] So the file is added and it is ready to be committed. So let's go ahead and run git commit. [No audio] Once we press Enter, then we will get a new screen in our terminal. Actually, this is a default git editor, which is called Vim. Here we have some comments, and we are able to insert here message for the commit. For that at first we have to click i, which stands for insert, and then we will be able to type the text. Let's say Created variable. [No audio] So once we insert the message, then in order to close this editor and go back to our terminal screen, we have to press Escape, then Shift column, and, in order to save the text, we have to type W, and finally in order to quit, we have to press Q. If we don't want to save, then we can just type Q and the editor will be closed. So as you see, we are back to our working screen and also commit is made. In order to check that, let's run git log --oneline. So, you can see that the commit is made with a relevant message. Alright. So that's it what I wanted to show regarding making commits. Before we finish this lecture, I want to set back Visual Studio code as the default text editor. For that we have to run the command git config --global then core.editor. [No audio] Then inside quotes we have to write "code and we have to add here the flag --wait", because it will allow git to wait until the message is created and file is closed. Alright, so after that, VS code will work as the default text editor. Let's take a little break here and move on to the next lecture. [No audio]

[No audio] Alright. So we have already learned some basic stuff regarding Git, and now in this section we're going to be talking about how to undo or discard the things in Git. Throughout this section we will be referring to some important commands such as checkout, revert and reset. Those commands allow us to go back in time and check all the versions of the project, I mean different snapshots, which we have already made, and also they allow us to manipulate on commits. Okay. Before we move on and start talking about those commands, first of all I want to refer to HEAD pointer, which you have already seen a couple of videos ago. Alright. Again I'm inside our working GITPROJECT folder. Let's display all of our commits, which we have made so far using git log --oneline. [No audio] So as you see, overall we have five different commits, and with the first one, which in fact is the last commit we see in (HEAD, which is a pointer. And also we see master, which is a default branch. The branch itself is a timeline of the commits. So, by default HEAD points to the last commit, but if we go back to any of the previous states of the project, then HEAD will point to the relevant commit, and you will see it soon. In order to prove that HEAD points to the last commit by default, we can run the following command, git show and then HEAD. So, as you can see, we got here the last commit with a bunch of information. I mean with author, date, and the message. Besides that, down below, we see some other stuff. To be more precise, the difference, but for now I'm not going to refer to it because it's a subject of different section. Now I just want to show you how we can use the HEAD. In this case, HEAD performed the same job as id of a commit can do. I mean, we can run the same git show command, [No audio] and if we pass here the id of commit instead of HEAD, by the way, when you work in git bash and you want to copy something, you just need to select the text and it will be automatically copied. So now you see that we got here the same result. So we're able to use HEAD instead of id, but it's interesting what can we do if we want to show up other commits using the HEAD. For that, let's at first clear the window, then log again commits using git log --oneline. And now let's see how we can access other commits using the HEAD. Let's say on this second one. For that we should run git show, then HEAD, which should be followed by tilde sign, and then we have to type 1. So we wrote here 1, but it points to this second commit. And the same way works HEAD tilde 2, 3, and so on. So if we press enter, then the second commit will show up. Okay. So that's all about the HEAD pointer. We will use it in the future throughout the course. For now, let's stop here and move on to the next lecture. [No audio]

[No audio] Alright. It's time to start talking about undoing things in Git. As we said in the previous lecture, there are three different commands which allow us to undo things. And the first one that I'm going to talking about in this lecture is git checkout. So with the help of this command we are able to unmodify files. Also we can go back in time and check different states of our project. It also allows us to move between different branches. Ok. Let's go ahead and at first see how we can unmodify files. Before that, let's modify files and make some changes in all of our three files. Let's create a span element inside paragraph with some text, let's say Hi there. Then go to style.css file, select paragraph, and change its color. Make it green. [No audio] And finally, let's open this script.js file and create another variable. const b [No audio] =100. [No audio] Alright. So we have modified all three files. Let's check the status. [No audio] As you can see, all of our files are modified. Now suppose that we accidentally made some error and we need to unmodify the file. Let's say script.js. For that we can use the checkout commands, which actually is shown here as well. So if we run git checkout, and the filename script.js, then the second variable that we have just created will be gone, and also if we check again the status, [No audio] script.js file will be no longer shown here because now it is unmodified. So you agree that it's really useful. If you want to unmodify all the files at the same time, then you can run git checkout with dot. By the way, instead of a dot you can use an asterisk as well. Ok. So if we check again the status, then you will find that our working tree is clean and all the changes that we made are discarded. Ok. Another job that git checkout can do is to go back in time and check the previous states of the project. Unlike other commands such as revert, and especially reset, check out is the safest one, because we check out, we cannot change or delete previous commits. Alright. Let's log commits using git log --oneline. Suppose that you want to go back in time and check out the state of your project where we Added heading in HTML document. I mean, let's say we want to check out our project only second commit. In order to do that, we have to use git checkout, [No audio] and then we have to specify the proper id of the commit. So let's copy id and Paste it here. So once we press enter, we will get the previous state of our project. As you see, script.js file no longer exists and here just the tab is opened. You can see here in parentheses that file is deleted from disk. So we just need to close this tab. Also in html file we have just heading element. And in style.css file, style for a paragraph is removed. Besides that you can see here in the terminal the following text, You are in 'detached HEAD' state. It means that head pointer no longer refers to the last commit. Also, you can see down below that instead of the master, here is displayed the id of the commit. Alright. Let's log commits, write git log --oneline. [No audio] As you see, just the first two commits are displayed. It doesn't mean that the rest of the commits are deleted. We have just moved back in time and we are in the previous state of the project. Of course, now head points to the commit on which we currently are. [No audio] Ok. So that's how we can check the previous state of the project. In order to go back, we need to run git checkout master. So you see that script.js file is back, and if we run again git log --oneline. Then all our commits will be back and head will point again to the last commit. So checkout is very safe because it is a read only. You cannot change or edit previous commits, and now I'm going to prove that. Let's go back again to the previous commit, let's say to the second one, write git checkout, and paste your id. [No audio] And now let's try to make a commit. Let's add in the style.css file color:blue, [No audio] then check the status. [No audio] So the file is modified, and in order to add it to the staging area, let's run git add, and a file name style.css. Then again check the status. [No audio] It is green, so it is staged, and finally, let's go ahead and make the commit. Let's run git commit -m, and write here the message "Test commit". [No audio] Alright. So let's log the commits. Run git log --oneline You can see that the commit is added. But if we go back using git checkout master [No audio] and then log commits using git log --oneline, you will find that here we don't see that newly created commit. And also if we go back again to the previous state using git checkout with id, and then if we run git log --oneline, [No audio] you will find that commit no longer exists here as well. So it means that checkout command is just a read only and we are not able to change, edit the commit, or add the new one in the previous state. Actually for that we can use other commands like revert or reset, which we will discuss in the coming lectures. But anyway, if you want to create a new commit in case of checkout, you can do that by creating other branches, which you will see later in the course. Alright. That's it about git checkout. Next we have to discuss how to undo things using git revert, and for that let's move on to the next lecture. [No audio]

[No audio] In the last lecture we talked about the git checkout. This command allows it to unmodify the files and also it helps us to go back in time and check the state of the project on any commit we want. As you remember, git checkout is just a read-only command, you are not able to undo things or create a new commits in previous states of the project. Now we're going to introduce you to other commands, which allow us to undo things in Git. The first command that I'm going to talk about is git revert. It is used to undo the changes to your projects, commit history. So the revert command takes the specified commit, inverse the changes from the commit and creates a new revert commit. Let's go ahead and see in practice how it actually works. At first, let's log the commit history using git log --oneline. So, as you see, overall we have five commits. Suppose that we want to undo the changes from the commit in which we added the paragraph in the HTML document. So, how can we do that? We should run the command, git revert and then we have to specify the ID of the commit. So let's Copy and Paste it here. Once we run this command, then the new temporary file will be opened inside of our text editor. You can see here revert with the message of the commit. Actually we are able to change this message, but for now I'm going to leave it as it is. Then down below you will find that this reverts commit and we have an id of the commit, which we're going to revert. So, now we have to close this file and once we do that then the paragraph will be deleted from the HTML file. And also, as we already said, the new commit will be made. So if we run git log --oneline, then you will find that the new commit is made, Revert "Added paragraph". So, we have gone back in time and undid the specified commit. Actually between those two commits, we had one commit, created variable. If we check the script.js file, you will find that the variable still exists. It means that when you use git revert command, you don't touch any of the changes which was made in another commit. You always undo this specified commit. So this picture is very useful because sometimes you may have some mistakes in your previous specified commits and git revert allows you to fix those mistakes without changing any other commit. Alright. Let's move on and talk about git reset, which actually is known as the dangerous and unsafe command, because it can permanently delete your work, so you should be careful with git reset. Let's go ahead and see actually how this command works. Before that I'm going to add a couple of commits to our project. Let's create a simple navigation in HTML document, use nav element, and then inside of it insert li with couple of list items. [No audio] Again, if you're not familiar with those different elements, and in general with HTML or CSS or JavaScript, it's not really a problem. You can feel free and use just the text files, or even you can work with Office documents like Word, Excel and so on. We just need to make some changes in our files. Alright. Let's go ahead and add the index.html file to the staging area. For that we need git add and the file name, then check the status. [No audio] So the file is ready to be committed. Let's make a new commit with the message "Added navigation". [No audio] Then let's go ahead and log the commit history. [No audio] So here we have our last commit. Now, I'm going to make another one. Let's go to the script.js file and create some simple function. [No audio] Then add this file to the staging area, we need git add script.js. And finally make the commit with the message "Created function". [No audio] Then log the history. [No audio] So, now suppose that we want to delete those two commits, which we have just made. So how can we do that? That's the moment when git reset comes in, we can run git reset, and now we can use three different flags, which actually act in different ways. The first one is --mixed, which is a default flag. It means that if we run just git reset, then the mixed flag will work anyway. Besides that, we have soft and hard. [No audio] I'm going to discuss all of those three flags, let's start with mixed. So, now we have to indicate the id of the commit. In this case, when we want to delete those two commits, we had to specify the id of the previous commit, I mean the id of Revert "Added paragraph". So, if we run this command and then check the history, [No audio] you will say that the last two commits are removed, but it doesn't mean that we removed physically the navigation and the function from our files, as you see both of them still exists. So it means that we have just removed the commits from the history. If we check status, [No audio] you will find that the index.html and the script.js files are unstaged, so they no longer exist at the staging area. So, in order to discard the changes from the files, I mean to remove them permanently, we can run git checkout, [No audio] then dot. Now, as you see, we no longer have here the navigation and also the function and the JavaScript file. If we check the status, [No audio] then our working tree will be clean. So that's the way how git reset mixed works. It just removes the commits from the history and also it unstaged the files from the staging area. They end up in the working directory. Let's go ahead and describe other flags. Let's say soft. First of all, log the history. [No audio] Now suppose that we want to remove the next two commits, I mean Revert "Added paragraph" and Created variable. As we said, now we're going to use the flag soft, so let's run git reset�--soft, and then we have to specify the id. [No audio] Let's log the history again. As you see, commits are deleted, but like the previous case, files are not changed, and the JavaScript file will still have the variable. The difference between the soft and mixed flags is that in case of soft, the files are maintained on the staging area, they are not unstaged if we check the status. You will see that both files are green, so it means that they are staged. Now, if we want to make the changes in the files, we have to unstage both of them. For that we have to use git reset. So, in order to unstage both files, we have to place here ., and now if we check the status, [No audio] files will be unstaged, and finally, in order to make the changes, we have to use again git checkout with dot. As you see, the variable is deleted from the JavaScript file. Also, as you can see, the paragraph is displayed back, which maybe, is a slightly unexpected. It happened because we in fact deleted Revert "Added paragraph". Therefore the Commit "Added paragraph" is no longer undid. Okay. So that's the way how git reset --soft works. So both flags, I mean mixed and soft, remove the commits, but the first one unstage the file from the staging area. As for the second one, it maintains them and doesn't unstage. As we saw, in both cases we have gone through a couple of steps in order to make the changes in files and not only remove the commits, but if you want to remove the commit and also make the changes in file simultaneously, you can simply use git reset with the hard flag. Let's log the history again. [No audio] So, now suppose that you want to delete all of the commits except the initial one, and also at the same time you want to make the changes in files as well. For that we have to run git reset --hard and then we have to specify the id. So, as you see, the changes are made in files, the heading is removed, also the script.js file is deleted, and we have the state of the project from the initial commit. If we check the status, you will see that the working tree is clean, and also if we log the history, we will get just a single initial commit. Alright. So that's the way how git revert and git reset work. I know that it might be a little bit hard to grasp. So if you feel yourself confused, if anything in this video is not clear, then I recommend to re-watch this lecture and test things on your own. Okay. That's it. See you in the next video. [No audio]

[No audio] Okay. So we have already learned about a couple of things in Git, I mean how to track files, how to add them to the staging area, how to make commits and also how to undo things so far. I think now is the perfect time to talk about how to ignore files or folders from tracking, I mean the files and folders which shouldn't be recognized by Git. When you work with projects, you might have some auto generated files or different folders which you don't need to track all the time, because it might slow down your work in process. So we need some solution which will allow us to ignore those files. For that in Git we use a .gitignore file. Actually it doesn't have any name. When you want to create it, you just need to indicate the extension .gitignore. Let's go ahead and create this file and see actually how it works. [No audio] Now let's check the status. So for now we have just one untracked file, .gitignore. Suppose that we should have some text files, which we need to ignore, let's go ahead and create them. Let's say text-1.txt and text-2.txt. As you see, those newly created files are green, and also we see here letter U, which actually stands for untracked. So if we check the status, then you will find that we have three untracked files. So now we need to specify those text files inside of the gitignore file. For that, we just need to indicate the path of the file, in this case text files and gitignore are placed in the same directory, so we just need to specify the file names. Actually, before the names, we need to place / followed by the name of the file. [No audio] If we save, then we will no longer have a green color with the letter U. It means that git already ignores those two files. If we check the status, we will get just one untracked file, which is the gitignore file itself. So again, once you specify the files in gitignore, then Git will no longer recognize those files. In order to prove that again, let's modify those text files. [No audio] And also create a new element in the index.html file. Let's place here Hello there. Now if we check the status, [No audio] we will get just one modified file which is index.html and also one untracked file, .gitignore. Alright. So that's the way how gitignore works. Now I'm going to show you one important thing. Suppose that we want to ignore the index.html and style.css files as well. For that, we have to put them in gitignore file. [No audio] Then let's make some changes in both of the files. I'm going to create paragraph with some text. [No audio] And then in style.css file, let's create style for h1 heading out. [No audio] Now, if we check the status, [No audio] then may be unexpectedly, those two files will be still modified. So in this case, Git did not ignore HTML and CSS files, and the reason of that behavior is that those two files have been already modified before we created .gitignore file. So in order to fix that, we have to delete the cache from the history. For that we have to run the following command. We need git rm, which stands for remove, then -r, --cached, and in order to include all the files we need dot. So now we cleaned the cache, and I want you to know that we just cleaned the cache and the files themselves are still here. So if we check the status, [No audio] we will get here just one untracked file. So now the HTML and CSS files are no longer recognized by Git. Now as we see, changes are ready to be committed. Let's unstage files from the staging area, we need to run git reset ., let's check the status. [No audio] So now files are modified and finally we need git checkout ., so if we check the status again, then we'll get only one untracked file, .gitignore. Actually, in the real world, it's recommended to create the .gitignore file once you create the repository in order to avoid such unexpected things. Alright. Before we finish this lecture, I'm going to show you a couple of things. Let's create a new folder. I'm going to call it auto-gen-files, I mean auto generated files. Then inside this folder, let's create several text files. [No audio] Now suppose that we want to ignore all those text files. For that, we have to indicate them in gitignore file. Before we do that, I'm going to show you how we can make the comments and the .gitignore file. We just need to place the pound sign followed by the text of the comment Auto generated files. [No audio] Alright. So let's go ahead and specify the file names. First of all, we need to indicate here the folder name, auto-gen-files, [No audio] then /, and then the file name, a.txt. Let's do the same for the rest of the files. I'm going to duplicate this line and just change name of the txt files. [No audio] Let's check the status. So, as you can see, all three files are ignored. In this case, we have just three of them, but sometimes we might have hundreds of such files, and then indicating the file names individually will be really crazy, right? So what can we do here. In order to solve this problem with just one line of code, we can use an \* instead of the file names. Now, if we check the status we will get the same result. So you can simply use this technique. Okay. Now suppose that we have another file inside of this folder, but in this case with a different extension, let's say about.html, [No audio] and we need to ignore again just the text files. So in that case we still need to use the asterisk, but it should be followed by the file extension .txt. So if we check again the status, [No audio] then we will get here .gitignore as the untracked file, also folder, auto-gen-files. But if we take a look at the sidebar, you will see that only about.html file is untracked. All other txt files still are not recognized. [No audio] Okay. That's it about gitignore. It's the really useful and helpful feature, and you will be using it quite frequently when you will work with your own projects in the future. Once we prepared the gitignore file, let's make the commit. At first, add it to the staging area, [No audio] then make the commit [No audio] with the message "Prepared, .gitignore" file, [No audio] and finally check the history, run git log --oneline Okay. That's it. See you next time. [No audio]

[No audio] So we already have learned about the basics of Git. We have gone through some important topics. We know how to create the repository, how to make commits, and also how to go back in time, check the previous state of the project, and undo things. So I think we have quite enough knowledge to move on and introduce you to GitHub. First of all, let's find out what GitHub�is. As a definition, we can say that GitHub is an online service where you can share your code or files in order to collaborate with different people. In other words, it is another computer on which every person who is working on a specific project will have access. For example, you can imagine a web developer team who wants to build a website and everyone has to update their code simultaneously while working on the project. In this case, GitHub allows them to build a centralized repository where everyone can interact with each other and manage the code. So eventually the final product will be built on it, online repository. Alright. Let's consider a difference between Git and GitHub. As usual, people are a little bit confused regarding the difference between Git and GitHub. Actually, as you already know, Git is a version control tool, and it allows us to manage the project locally. Additionally, we can say that it allows us to fetch the data from the central server or push data to it. As for the GitHub, it is a hosting platform, which allows you to host a repository on a remote server. Besides that, we can also say that GitHub is a user friendly public platform with a nice visual interface, and millions of users are sharing their projects to the entire world. Alright. We can talk a lot more about GitHub and its features, but I think for now it's not necessary to over complicate things. Would be better if we go through the features of GitHub step by step in the coming lectures. So, if you're an absolute beginner with GitHub, then I will show you how to use it. You need to create an account on GitHub. So let's get started. First of all you have to go to the website called github.com, where you can find a very simple registration form. You just need to type here your unique Username, Email, and Password. Once you fill up those input fields then you have to click this green button. Again, the registration process is going to be very simple, so I'm sure that you won't have any trouble. Once you sign in, first you should click Verify, and you will get some funny creature here which you have to rotate. [No audio] Once you do that, then click Done, and after that we will move on to a GitHub Welcome page. Now we have to go through some steps. Here we have two different options to choose a subscription. The first one is Free. As for the second one, is a Pro option, which will charge you with $7 per month. Actually the main difference between those two options is that in case of the Free option you are allowed to have only three collaborators for a private repository. As for the second option, you don't have any limit. For our course, the first option is absolutely enough. So I'm going to choose it. We don't need to check any of those boxes. Let's just click Continue. [No audio] So here we have to answer 2 different questions. The first one is about the level of programming experience, suppose that we are new to programming. As for the second question, let's check the second option, Learning Git and GitHub. Let's leave this field empty and click Submit. [No audio] Now we need to verify our email account. For that you have to visit your email account where you should get the email from GitHub. Once you click verify email address, you will get to another Welcome page. Alright. I think for now it's enough. Let's continue in the next lecture. [No audio]

[No audio] So in the last lecture we created an account on GitHub and now we're going to explore the website, I mean some of its basic parts, and then we will start creating our very first repository. Currently we are on the Welcome page of GitHub. Here we see two different buttons, Read the guide and Start a project, and actually before we create our first repository, I'm going to show you how the existing repositories look like. You can either check out some repositories on the right hand side of the page. GitHub shows us some random repos, or you can just search for them from here. I'm going to search for, let's say, react. So, here we have some results regarding react, let's go to the first one which is the actual repository of React library. So now we're on the page of react's repository and actually all other repos look in a similar way. Here we have different files and folders which React library consists of. If we click one of them, we will be able to see some code. Here we can see the green button, which allows us to Clone or download the entire source. At the top of the source files you see a couple of different tabs. The first one is commits. Overall we have more than 11,000 commits. If we click it, then we will be able to explore all the commits, which were made so far. As you can see, each commit has its own id, similar, which we saw in our previous lectures. Next to commits we have branches, which for now you're not familiar with. Also we have here releases of different versions of react. Then it is followed by the number of contributors. Those are the people who actually work on developing this library. Alright. Let's move on and describe some other things here. At the top of this page you see that this repo is used by more than 2 million people. Next to it we have Watch, and if we dropdown it, we should see a couple of options. If you click Watching, for example, then you will be able to get updates regarding this repository. Also you can give a Star to it and so on. For now I think it's enough. Let's go ahead, check out our profile. You're able to customize your profile. You can Add your bio. Then we have a couple of input fields for your Company name, for Location and Website. Also you're able to change avatar and set your profile image. Alright. Actually you can go ahead and explore all those things on your own. You can play around and check out different pages in order to get familiar with GitHub website. Let's go to the main page and start creating a new repository. For that we have to click either the Start a project button or you can click here the link on the left hand side of the page, Create a repository. As you see, we moved to a different page where you can create a repo. We see here our username followed by the input field in which you should enter the name of the repo. I'm going to call it the exact same name, which we used for our local repository, I mean gitProject. [No audio] Then down below we have another input field for Description, which actually is optional. Let's insert here a little description. Let's say, This is my first GitHub Repository. [No audio] Then, down below, we have two different options, Public and Private. If you choose Public, then anyone will be able to see your Public repository, but you will choose who can commit. As for the Private repo, you are able to choose who can see and commit to this repository. Alright. Let's select Public. Next we can initialize this repo with a README file, but for now I'm not going to do that. Down below we are able to Add gitignore and some license files. For now we don't need any of them. Let's just click Create a repository. Now we ended up on another page which is already similar to the page which we described recently, I am at the top part of this page. Also here we have some instructions for creating a new repo or adding an existing one. And regarding this instructions, let's talk about in the next lecture. [No audio]

Okay. So, we created our very first repository in the previous lecture, but for now, as you can see, we don't have here any files or folders because we should add them from the local repository. Here we see some instructions. Also on top of them we have a link of our repository. So the first part of this instructions shows us all the steps starting from the Git initializing through the pushing local repository to GitHub. Most of those commands, you're already familiar with, I mean those three commands, git init, git add, and git commit. We have been using them in previous lectures. The first command here shows us how to create the README file with some comments. For now, let's don't pay much attention to it. As for the last two commands, they're used to connect the local repository to GitHub, and then to push the repo to GitHub. So this instructions are supposed to be used when you don't have your local repository yet, but in our case we have already created it, so we need to use the second part of this instructions, I mean those two commands. So, let's go back to our working environment where we have our local repository. Let's check the history of our commits, run git log --oneline. So overall we have two commits. Inside of auto-gen-files folder, there is just one untracked file, about.html. I'm going to add it to the staging area and then make a new commit. Let's run git add. Then we need to specify the folder name, auto-gen-files/, and filename about.html. Then check the status, [No audio] and now make new commit [No audio] with the message "Added about.html file". And finally check the history, run git log --oneline. Alright. So now we have three commits and it's time to push our local repository to GitHub. The first step that you have to go through is to connect our local repo to GitHub, and for that we have to run the following command. We need git remote add, next we need to set the name of the remote. It is actually optional, but commonly it is called origin, so I'm not going to change it. And after that we have to specify the link of our repository, which actually was generated by GitHub. In general, GitHub gives us two types of links. The first one starts with HTTPS. As for the second one, it starts with SSH. In this case, we're going to use HTTPS, so let's copy the link from here and paste it in our terminal. So, once we run this command, then our local repository will be connected to GitHub, and the next step is to push it to GitHub. For that we need to run the following command, git push, then the name of the remote, origin, and finally we have to indicate here the name of the branch. For now we don't know much about branches, the only thing that we know is that currently we are working on the default master branch, so let's indicate it here. As you see, we got here a little popup, which requires us to enter our GitHub credentials. So let's quickly enter our Username and Password, [No audio] and after that we just need to reload the page. [No audio] Alright. So here we have our repository with all the files and folders, which we had locally. Okay. So we have successfully pushed our local repository to GitHub. Next I'm going to show you how to add commits from GitHub and how to pull data from GitHub to the local repository. So, let's move on. [No audio]

Alright. So in the previous lecture we learned about how to connect the local repository to GitHub and also how to push it to GitHub. We pushed our existing local repository. So, here we have our files and folders. At the top of those files we have a tab commits, and you can see that overall we have 3 commits. If we click it, then we will see all 3 commits, which we have made so far, we see here the message of the commit, then on the right hand side there is a proper id of commit, and also we see the author of the commit and the time when the commit was made. In our case, the author of commit is tryinggit It is a different from the name of the user which we used in the configuration. If you remember, we used some dummy name in the email and that's why GitHub didn't recognize them. By default, it set this name as the author of commits. So it's changing configuration, the name and the email and use the valid ones. At first, let's check our current name in the email. For that we have to run git config --global then user.name. [No audio] So the name is set as John. Let's check the email. As the email, we have john@gmail.com. Alright. Let's go ahead and change both of them. I'm going to set name as codeandcreateweb. [No audio] As for the email, lets set it as codeandcreateweb@gmail.com. So, now once we change the name and the email, it doesn't mean that the author of previous commits will be changed accordingly. If we reload the page, then still we will have the previous name. Let's see what happens if we add a new commit. I'm going to insert here h1 heading element with the text Hi there. Then let's add this file to the staging area, [No audio] and now I'm going to make a new commit with the commit message, "Commit from new author". [No audio] After that we should push our local repository to GitHub, and for that, as you remember, we have to run the following command, git push origin, which is the name of the remote, and then master, which is a name of the branch. [No audio] Now if we reload the page, then we'll get a new commit with a new author. Alright. Now it's time to make a new commit, but now from GitHub I'm going to show you how to edit files and how to create a new commit. Let's open index.html file. So in order to edit this file, we should click here this icon and after that we will be able to make some changes in index.html file. I'm going to add here a paragraph. [No audio] Unfortunately, here we don't have any help from the editor. We should type everything manually. So once we added this file, then down below we should make a commit. The first input field is for the message of the commit. Let's insert here Added paragraph. [No audio] As for the second input field, it is for extended description, and actually it is optional. Anyway, let's institute something. Let's say, We updated index.html file [No audio] from GitHub. [No audio] And now we should click Commit changes. [No audio] So, now we have created a new commit from GitHub successfully. As you see, there is a new commit with commit message, Added paragraph. Also if we click those three dots here, then the description will be displayed. Okay. So we created new commit from GitHub, but let's see what happens on the local repository. If we check the history using git log --oneline, then we will get only the old commits. It means that our local repository is outdated and it doesn't recognize the newly created commit from GitHub. So, now it's time to pull the data from GitHub to the local repository, and for that let's move on to the next lecture. [No audio]

[No audio] Okay. So we made our very first commit on GitHub, but as you remember from the previous lecture, when we check the history of our local repository, newly created commit didn't end up there. So our local repository is outdated and we have to pull the data from GitHub to it. In order to do that, we have to run the following command, git pull origin and then master. [No audio] Now if we check again the history [No audio] then we'll get you the last commit from GitHub. So once you make the commit on GitHub, then in order to update your local repository, you should use the git pull command. Alright. When you work with Git and GitHub, you should be frequently using git pull command with specifying the name of the remote, I mean origin, and the name of the branch master. In order to avoid indicating those names all the time, Git allows us to set kind of shorter way to pull data from the remote repository. So for that we have to run the command git branch --set-upstream -to = origin/master and then again master. [No audio] So the first master is a branch on our repository on GitHub, I mean this branch right here, which is created by default once we create the repository on GitHub. As for the second master, it is a default branch on the local repository. So once we run this command and then make a new commit on GitHub, by the way, I'm going to show you how to create a new folder or file on GitHub. You have to click Create new file. Then, if you need a new folder, you should set its name followed by the /. And now we have to indicate the name of the file, let's say script.js. After that, let's make commit, Added JS file. [No audio] Now, in order to pull that newly created commit to local repository, we have to use again git pull command, but now without indicating any of the names like origin or master, we just need git pull, and if we check the history, then we will find here newly created commit, Added JS file. Alright. So that's it. In this lecture we have learned about how to pull the data from GitHub to the local repository, and also I showed you how to do that in kind of shorter way. [No audio]

Hello, and welcome to our new section in which we're going to be talking about one of the important features of Git which is branching. In Git, branches are a part of our everyday developing process. For example, when you want to add a new feature to your project, or maybe you might need to fix a bug, then you can use branching. First of all let's recall what branch itself is. As you can see on the diagram, there is a project timeline. We have here different commits, and this timeline is actually a branch. By default, when you create a repository and then make different commits, you're working on the master branch, which actually you have already seen lots of times in the previous sections. Sometimes you may need to add a new feature to your project, and at the same time, you don't want to touch or break the current state of the project. In that case, the best solution is to create a new branch and work on that newly-created branch, I mean, to add files, make commits, and so on. Then after creating and testing the new feature, if you're happy with that, then you can merge this branch to the master branch. You're able to create as many branches as you wish. Also, it's really helpful when, for example, several developers are working on the same project, each of them can work on a different branch without breaking other developer's work. Alright. Let's go ahead and see in practice how to work with branches. Again, for now we are on the master branch. So, in order to create a new branch, we need to run the following command, git branch, and now we have to specify the name of the branch. In this case, let's say feature-1. Now, in order to check which branch we are working on, and also what branches do we have, we can run git branch. Now, as you see, overall we have two branches, master and feature-1, and currently we are on the master branch. It has different color and also in front of it, there is placed an asterisk. Let's see how we can switch to a different branch. For that we have to use git checkout, and then we have to specify the name of the branch, in this case feature-1. As you see, now we switch to branch feature-1, and if we run again, git branch, you will see that now we are working on a feature-1 branch, also the name of the branch inside of the parentheses is different. Now if we check the commit history using git log --oneline, we will get all the commits, which we have made so far. So, we are able to use those commits and also make new ones, which won't be available on the master branch. In order to prove that, I'm going to make some changes to the project and make a new commit. Let's open style.css file and change the color of heading element. Let's make it green. Then add this file to the staging area using git add, [No audio] and finally make a new commit with the message "Styled heading". [No audio] So, now if we check the history, then we will get here the new commit, Styled heading. But actually if we switch to the master branch using git checkout master, [No audio] then you will see that the Style for the heading element is gone from the CSS file, and also if we check the history, then the last commit won't be available here. Alright, let's move on and see how we can create and switch to the newly-created branch with just one command, suppose that we want to create another branch, feature-2, and switch to it immediately. We need to run git checkout, then we have to use flag -b, and we have to specify the name of the branch, feature-2. [No audio] So, now if we run git branch, you will find that overall we have three different branches, and currently we are on the newly-created feature-2 branch. Alright. Before we finish this lecture, I'm going to show you how to delete branches. For that we have to run git branch and the flag -D, but we need to write D in uppercase, and then we have to specify the name of the branch. Remember that you cannot delete the branch which you're currently working on, in this case we are on feature-2. So if we try to delete it, then we will get an error saying that we Cannot delete branch 'feature-2', but if we change feature-2 into, for example, feature-1, then it should be deleted. If we run git branch [No audio] then feature-1 branch will be no longer available. Alright. So that's it for now. In this lecture, we have learned about how to create a new branch, how to switch between different branches, and also how to delete them. See you in the next lecture. [No audio]

Alright. So in the last lecture, we learned about branches in Git, we created new branches, and also we saw how to delete them. Let's check what branches do we have here. For that we have to run git branch. [No audio] So, now overall, we have two branches, master and feature-2, and we are working on the feature-2 branch. Alright, let's check the history of commits on the feature-2 branch. Run git log --oneline. So here we have all the commits, which we have made so far on the master branch. I'm going to add new commits on feature-2, let's add some style to the heading element, select it and change its color. [No audio] Then add style.css file to the staging area, then make new commit, [No audio] with message "Styled heading", [No audio] and finally check the history, run again git log --oneline. [No audio] Alright. So here we have our last commit, and also here we see that it belongs to feature-2 branch, which is one commit ahead of the master branch. Okay, now I'm going to show you how we can add a feature-2 branch to our GitHub repository. It is very simple, we just need to run git push then origin, which should be followed by the name of the branch, feature-2. [No audio] Now, if we check our GitHub repository, the first thing that we may notice here is our recently pushed branches, in this case, feature-2, and also we see green button on the right hand side, Compare & pull request. Actually, regarding pull requests, we will talk a bit later. So now if we reload the page and drop down this button here, we'll find branch feature-2. Let's go ahead and select it, then go to commits tab, and, you see here we have last commit, which we made on the feature-2 branch. Alright. Now I'm going to show you how we can create branches from GitHub, and then we will add these branches to our local repository. So, in order to create a new branch, we need to drop down this button. As you see here, we have an input field, and we have to enter the name of the new branch inside of that input field. Let's say feature-1. Then we have to click Create branch. So, as you see, we have created a new branch from GitHub successfully. If we check tab branches, we'll find all three branches, master, feature-1, and feature-2. As you see, master is a default branch. And also on the right hand side of the feature-1 and feature-2 branches, we have some numbers. If we hover over it, it will show us that feature-2 branch is one commit ahead of master. That's because we added a new commit on feature-2 branch, and that commit doesn't exist on master branch. Okay, now let's see how we can add branch from GitHub to the local repository. For that, first of all, we have to run command git pull origin feature-1, and then we need git checkout --track origin, / and the name of the branch, feature-1. So now we added feature-1 to the local repository, and also we switched to it. In order to check that let's run git branch. So, overall we have three branches, master, feature-1, and feature-2, and currently we're working on feature-1. I'm going to add new commit on GitHub to feature-1 branch. Let's open script.js file and create a new variable, [No audio] then make commit with the text Created variable. [No audio] So, now in order to add this commit to the local repository, we should run git pull origin, and the name of the branch, feature-1. [No audio] Now, if we check the script.js file, you will find the newly-created variable, and also if we check the history of commits using git log --oneline, then we will find here our last commit. So, each time when you make a new commit on GitHub on feature-1 branch, then to add it to the local repository, you just need to run git pull origin feature-1 command. Alright, let's see how we can delete the branch from GitHub, and also I'm going to show you how we can delete the remote branch from the terminal. So, in order to delete the branch on GitHub, we have to go to the tab branches, then we just need to click that delete icon. [No audio] So, as you see, we no longer have here branch feature-1. We can do the same locally. Let's delete branch feature-2. For that we need to run git push origin --delete, and then we have to specify the name of the branch, feature-2. So once we run this command, then branch on GitHub won't be available. [No audio] Alright. So, we deleted branches on GitHub, but they still exist in the local repository. So I'm going to delete them from here as well. First of all, let's switch to master branch, run git checkout master, then delete branch feature-1. For that we need git branch -D, and the name of the branch, feature-1. Let's do the same for feature-2 as well, and finally run git branch. As you see, we had here just the master branch. Alright. That's it for now about branching on GitHub. We have to consider some other topics regarding branches, and for that, let's move on to the next lecture. [No audio]

[No audio] In the last section we learned about how to create different branches in the local repository and also I showed you how to do the same on GitHub. In a real world, when you create different branches and work on them, eventually you need to combine them with the master branch, and this process is called merging branches. So, in this section we're going to look at how to merge branches, what type of merges do we have, and also how to resolve some conflicts that might occur during merging the branches. Alright. Let's go to our working environment and start working on merging branches. First of all, let's check what branches do we have for now, run git branch. So, currently we have just one branch, which is master. Also let's log the history. [No audio] So we have here a couple of commits from previous lectures. I'm going to create a new branch Let's call it dev, which stands for develop, and also I'm going to add a couple of commits to the new branch. For that, first of all we need to switch to that branch. Let's run git checkout�dev, [No audio] then open style.css file, and change color for heading. [No audio] Let's make it blue. [No audio] Now, I'm going to add this file to the staging area and then make commit as we have already done lots of times. Previously we used two different commands, git add and git commit, but we can use shorter way with just one command. We can run git commit -am and then we have to specify the message, let's say "Styled heading". [No audio] So, now we added the file to the staging area and made a commit simultaneously. So this is a shorter way, but you have to remember that this command works only for modified and deleted files, but if you have new files which are not tracked, this command won't work and we have to use previous commands, I mean git add and git commit separately. Alright. Let's add one more commit. I'm going to add style to the paragraph. Let's change its color as well, [No audio] and then again add the file to the staging area and make commit, run git commit -am with the message "Styled paragraph". [No audio] Let's log history. [No audio] So, as you see, we have last 2 commits here and dev branch is 2 commits ahead of the master branch. So, now suppose that we are happy with those changes and we want to merge those two branches. To do that, first of all, we have to switch to master branch using git checkout. [No audio] So, at a glance you can notice that styles for heading and paragraph are no longer available, and also if you run history, we will get all commits which were made only master branch so far. So, now we have to merge those two branches and for that we have to run command git merge, and we have to specify the name of the branch, dev. So, once we run this command, then the styles will be displayed, and both branches will be merged. If you run the history, you will see that now head points to both branches, master and dev. So we have combined two different branches successfully, and this type of merge is kind of basic one and it is called Fast-forward merge. So in the case of Fast Forward merge, there is a linear path from the master branch to the second branch, which in our case is dev branch. So this branch is created based on the last commit of the master branch. It means that while we were working on dev branch, there was not added any new commits on master branch. And once we merge those two branches, then simply commits from dev branch were added after this base commit to the master branch. So, any changes which were made on dev branch should be applied and they will overwrite old state of the master branch, and there won't be any conflicts. To prove that, let's switch back to dev branch, [No audio] then change colors of heading in paragraph [No audio] and make new commit, run git commit -am, with the message "Updated styles". [No audio] Then go back to the master branch [No audio] and merge both branches, run git merge dev. So, as you're seeing now, new colors are applied to heading and paragraph, and the old styles are overwritten without any conflict. Alright. So that's the way how we can merge two branches in a Fast-forward way. There is another type of merge, and regarding that, let's talk in the next lecture. [No audio]

[No audio] Alright. So, in the last lecture we learned about how to merge two branches and we considered Fast-forward type of merging. Now I'm going to show you another type of merge, which is called 3-way merge. This kind of merge occurs when we create a new branch, work on that branch, and at the same time someone else makes commits on the master branch before we merge the branches. Let's consider this type of merge with an example. Let's recall what branches do we have. I'm going to run git branch. [No audio] So overall we have two branches, master and dev, and currently we're on the master branch. Next I'm going to log the history. [No audio] So, as we see, HEAD points to both branches because the last thing that we did was merging of those branches. Now I'm going to switch to dev branch, let's run git checkout dev, and then let's make a couple of commits. Open script.js file and create a variable, [No audio] then make a new commit. I'm going to use again git commit -am, which allows us to add file to the staging area and make the commit simultaneously as the message. Let's insert here "Created variable", [No audio] then log the history. [No audio] Next, I'm going to make another commit. Let's create a function in JavaScript file [No audio] and then make a commit. Use again git commit -am with a message "Created function". [No audio] And finally, let's run the history. [No audio] Alright. So now we made 2 commits and dev branch is 2 commit ahead of the master Branch. Now suppose that someone else is working on the master branch as well, and is making some commits. To imitate this process, I'm going to switch back to the master branch. [No audio] Then I'm going to make some changes in index.html file. Let's create span element with the content, This is span out. [No audio] Then make a commit, run again git commit -am, with message "Added span element". [No audio] And lastly, let's log the history. [No audio] Now suppose that we want to merge those two branches, in this case we will no longer have a Fast-forward merge because we've been working on both branches simultaneously. Let's go ahead and merge those two branches, run git merge dev. [No audio] As you see, the new file is opened in our text editor, and also we got the hint in the terminal saying that git is waiting to close this file in editor. So, we don't have to do anything in this file. Let's leave it as it is and just close it. So, merge is made successfully, but in this case we have a different type of merge, which is called 3-way merge. Also, as you see here, it was made by the 'recursive' strategy, which actually is the default strategy of 3-way merge. Let's log the history. [No audio] So, as you see, we have here new commit with the text, Merge branch 'dev'. So now all the changes which we made on dev branch are available on the master branches, but, unlike the Fast-forward approach, HEAD points to master and not to both branches. The reason is that after making commits on the dev branch, we made a commit on the master branch as well and that's why we see here this master next to a HEAD pointer and not both branches. If we switch to dev branch, [No audio] then this panel element won't be available, and also if we check the history, we will find only the commits, which we made on that branch. Alright. Let's take a look at the diagram and see once again when 3-way merge occurs. So, when we create a new branch, make some commits on that branch and then someone else makes commit on the master branch, it means that we no longer have Fast-forward merge and the merge type is 3-way merge. So, in this case Git will check out if there are any conflicts in files because people may work on the same files. If everything is okay, then dev branch will merge to master, and the commits will be placed in a sequence in history. So that's the case when 3-way merge occurs. Alright. In some cases there might be merging conflicts. You have heard this term several times, and to learn what it means, let's move on to the next lecture. [No audio]

[No audio] So, in the previous lectures we learned about how to merge two different branches in Fast-forward and 3-way merge ways. Sometimes multiple developers may try to edit the same content. For example, if one developer tries to edit code that the second developer is editing, then the conflict may occur. So in this lecture we're going to talk about merge conflicts. Actually merging and conflicts are a common part of Git. In other version control systems it is slightly hard to deal with merge conflicts, but Git makes it relatively easy. Alright. Let's see in practice when merge conflicts occur and how to deal with them. First of all, let's see what branches do we have run git branch. So, for now we have two branches, dev and master, and currently we're working on the dev branch. Suppose that we want to make some change in script.js file from dev branch, let's change the value of the variable a, make it 50, and then make a new commit, run git commit with the flag -am and with the message, "Changed the value of the variable a". [No audio] Let's log the history. [No audio] So, here we have a new commit, which is made on the dev branch. Now suppose that someone else is editing script.js file and is working on the same line, but from the master branch. So, to imitate this process, I'm going to switch to the master branch, [No audio] and I'm going to change the name of the variable. Let's make it c. [No audio] Then, let's make new commit, run git commit -am with a message, "Changed the name of the variable". [No audio] And finally, let's run the history. [No audio] So, as you see, we have here the last commit made on the master branch. Now suppose that we need to merge those two branches. For that, as you already know, we have to run git merge and we have to specify the name of the branch, in this case, dev. So, once we run this command, then Git will say that we have a Merge conflict in script.js file. And we will get here some different texts inside of the file. So, because of that we have been working on the same line from different branches. Then Git was not able to decide which version of the code to keep. Therefore, the conflict has arisen. So now it displays both versions of the code and we just need to choose which one is appropriate for us. In order to solve this conflict, we just need to edit this code manually. First of all, I'm going to delete those lines which are added by Git [No audio] and then I had to decide which version of the code I want to keep. Suppose that we need to change the name and the value as well. After that we have to save the file and add it to the staging area. [No audio] So, once we add it to the staging area, then we have to make a new commit, but in this case we just have to run git commit without any message. So, once we run this command, then the new file will be opened in a text editor as it was in case of a 3-way merge. Here we can change the message of the commit, but let's leave it as it is. So once we close this file, then new commit will be made, branches will be merged, and conflict will be solved. If we run the history, [No audio] then you will see the last commit, Merge branch 'dev'. So it means that we have merged branches successfully. Alright. So that's the case when merge conflicts occur and also that's the way how you can fix them. Now I'm going to show you how to stop merging conflict. For that, again let's edit the same line of code from different branches. Currently we are on the master branch, and let's go ahead and change the name of the function. Let's make it myFunc, [No audio] then make a new commit, run git commit -am with the message "Changed the name of the function". [No audio] Next I'm going to switch to dev branch [No audio] and now let's insert here inside of the function, the block of code console.log('Hi there'). [No audio] Then again, make a new commit, git commit -am with the message "Added a block of code". [No audio] Alright. So now it's time to merge branches. For that let's go back to the master branch [No audio] and then run git merge dev. [No audio] So, again we've got here Merge conflict because we worked on the same line of code, and now if we want to stop merging, we can run git merge --abort. [No audio] So this command will stop merging, also we will no longer have here conflict, but it keeps all the changes which we have made on both branches. As you see the name of the function is changed, and also if we switch back to dev branch, [No audio] then the function will have a block of code. So remember that git merge --abort just stops merging, and it keeps all the changes that are made on the same line of code from different branches. Alright. Before we finish this lecture, let's merge again branches and also solve the conflict. Let's go back to the master branch, [No audio] then run git merge dev. [No audio] So I'm going to leave here the second version of the code. Let's delete all the lines of code which we don't need. [No audio] Then add the file to the staging area. [No audio] Next we need to run git commit. [No audio] And finally, let's check the history. [No audio] So, as you see, everything works fine. Alright. That's it about merge conflicts. In real world such conflicts may occur frequently, but thanks to Git, is quite simple to deal with them. But anyway, when the developer team is working on a project, then each member of the team should know what is his or her working area and which files to work on, in order to avoid such merge conflicts. Okay. Let's move on to the next lecture. [No audio]

Alright. Once we are familiar with merging branches and also solving merge conflicts, now it's time to discuss another feature of Git, which is called Git rebase. Git rebase�is one of the tools which is used to integrate changes from one branch into another. In other words, rebasing in the process of moving or combining a sequence of commits to a new base commit. In order to make these words more understandable, let's take a look at this diagram. Suppose that we have a couple of commits on the master branch, then we create a new branch and add to it some other commits, and then after that we create a new commit on the master branch. So in this case, by default, base commit is the commit from where we created a new branch. But using Git rebase, we are able to change the base commit and rebase our branch on the newly created commit on the master branch. Also the changes which we made on the master branch will be available on the second branch as well. Alright. Let's go ahead and see in practice what we're talking about. First of all, I'm going to make some changes in our local repository. We have already made lots of commits, and I think it might be slightly confusing, so I'm going to delete most of them. Let's just leave here first 5 commits. I'm going to copy this id. Then, in order to delete commits, we have to run git reset --hard and paste your id. So if you run git log --oneline, we will get here just the first 5 commits. And, besides that I'm going to delete dev branch at all and create the brand new one. So let's run git branch -d and the name of the branch, dev. So now the branch is deleted, and we have to create a new one. For that we need to run git branch. Let's call this branch feature. [No audio] Then, I'm going to switch to it, [No audio] and finally check the history. [No audio] So everything works fine. HEAD points to both branches, I mean master and feature, and it means that they have the identical histories. So now I'm going to switch back to the master branch and make a new commit. [No audio] Let's change the font size for the heading element. [No audio] Let's make it 30px, and then make a new commit [No audio] with the message "changed font size for h1". Then run the history. [No audio] So the new commit is created, and now the master branch is one commit ahead of the feature branch. So it means that this change is not available on the feature branch if we switch to it, [No audio] and check the history. [No audio] Then you will find that here we don't have the last commit made on the master branch. So now if we want to make the last commit from the master branch available on the feature branch as well. We can use git rebase, which will allow us to�rebase�the feature branch on the newly created commit, which was made on the master branch. So, let's run git rebase master, then log the history. [No audio] So now as you see, HEAD points to both branches, feature and master, and it means that now the last commit from the master branch is available on the feature branch as well. Alright. Let's consider again with different example how Git rebase works. I'm going to add a couple of commits on the feature branch. Let's change the font-size for paragraph, [No audio] then make a commit. [No audio] with a message "Changed font size for p". [No audio] Next I'm going to add some margin to the body. [No audio] Let's set it as 15px, and again make the commit [No audio] "Added margin to the body". [No audio] And then run history. So, we have here 2 new commits and the feature branch is two commits ahead of the master branch. Now let's switch to master and create one commit. [No audio] I'm going to add a span element in index.html file with some content, This is a span element, [No audio] and then make the commit. [No audio] with the message "Added span". [No audio] And finally, let's log the history. So, as you see, we have all the commits made on the master branch, but we don't have here the commits from the feature branch. So the next thing that I'm going to do is to switch back to the feature branch and rebase it on the last commit, which we have just made on the master branch. So let's go to the feature branch and then run git rebase master. So now if we log the history, you will find that feature branch is rebased on the last commit made from the master branch. As you see we had here the commit, Added span, and it is followed by 2 commits, which we have made before this commit on the feature branch. So, now all the commits are available on the feature branch, but if we go to the master branch [No audio] and check the history. [No audio] You will find that still we have here the commits which were made only on the master branch. Now, in order to make commits from the feature branch available on the master branch as well, we can merge those branches, so we can run git merge [No audio] feature. So, as you see, now the branches are merged, and instead of 3-way merge, we have here the Fast-forward merge. The reason is that we changed the base commit, I mean, we rebased the feature branch on the last commit of the master branch, and that's why Git used here the Fast-forward merge. Alright. See you in the next lecture. [No audio]

[No audio] Alright. So in the previous lectures we talked about how to merge branches, how to solve some merge conflicts using different ways, I mean Fast-forward and 3-way merge, and also how to change the base commit using Git rebase. Until now we have been working locally and now it's time to go ahead and work on GitHub. In this lecture, I'm going to show you how to merge branches on GitHub. First of all, let's recall what do we have on our remote repository. So as you can see, we have a couple of commits made previously, and we have just one branch which is a master. I'm going to go ahead and create a new branch. As you remember, we just need to drop down this button here, then type name of the branch inside of the input field, let's call it feature-1, [No audio] and then we have to click Create branch. [No audio] So, as you see, we have now 2 branches. If we drop down again this button will find two different branches, master and feature-1. Alright. So now the next thing that I'm going to do is to make some commits on that newly-created branch. Suppose that we want to add some styles to heading and paragraph. I'm going to open the style.css file. Then, to edit this file we have to click this little icon here, and after that we will be able to edit the file. Lets select h1 heading elements and change its color. [No audio] Make it, let's say, green. [No audio] Once we edit this file, we have to make a new commit. Let's insert here the commit message, Changed the color of h1. And finally click Commit changes. [No audio] Alright. So we have made a new commit to the feature-1 branch successfully. As you see, on the master branch, overall we have 6 commits, but if we switch to the feature-1 branch, you will find 7 commits. It means that the feature-1 branch is 1 commit ahead of the master branch. I'm going to add one more commit. Let's add it once again, the style.css file, [No audio] and now change the color of the paragraph. [No audio] Let's make it blue. [No audio] Then down below, make a new commit with a message, Changed the color of p. [No audio] Alright. So currently the feature-1 branch is 2 commits ahead of the master branch. Suppose that we are happy with the changes and now we want to merge the feature-1 branch to the master branch. As you can see, here is displayed a green button called Compare & pull request. I'm going to click this button. Now we ended up on a new page. Let's go ahead and describe what do we have here. So, to merge branches on GitHub, we need to create a pull request. That's why we see here heading saying, Open a pull request. And underneath it we can read, Create a new pull request by comparing changes across two branches. So before we merge the branches, we need to compare the changes between two branches and create a pull request. Here we see two branches, master and feature-1, the branches which should be compared. Next to them there is a green check mark followed by the text, Able to merge. This branches can be automatically merged. It means that we don't have any merging conflicts, and the branches are free to merge. Next we have some input fields in which you can leave some comments. Let's insert here Updated style.css file. [No audio] Let's leave the second input field empty. Down below there are displayed the commits which we made on the feature-1 branch. And at the end of the page, you can find the changes that we made to the style.css file. Those + signs indicate the code that we added recently inside of the CSS file. Alright. So everything is ready to create a pull request. To do that, we just need to click this green button here. Okay. So we have created the pull request. We see here the username who wants to merge 2 commits into master from feature-1 branch. Also down below we see, This branch has no conflicts with the base branch. The base branch is the same as the master branch, so we can merge the branches. Let's click the button, Merge pull request, and then click to Confirm the merge. [No audio] Alright. So, as you can see, the pull request is successfully merged and closed, and now we can Delete the branch. You're all set--the feature-1 branch can be safely deleted. Let's click this button here. [No audio] So we have successfully merged the changes from the feature-1 branch to the master branch. If we check the commits history, we will find 3 new commits. The first 2 commits describe the changes made in the style.css file. As for the last commit, it's saying that we merge the pull request from the feature-1 branch. Okay. So that's the way how we can merge branches on GitHub. In this case we have been working on the remote repository from just one account. In the real world, when you want to contribute to someone else's repository, then you might not have the permission to merge the branches. You need to send a pull request to the owner of the repo who will check the changes and then merge the branches. We will discuss all those things a bit later. But before that, let's see how to deal with merge conflicts on GitHub in the next lecture. [No audio]

In the last lecture, we learned about how to create a pull request and merge two different branches on GitHub. As it was in the case of the local repository, sometimes you might have merge conflicts on the remote repository as well. So in this lecture I'm going to set up some scenario in which you might get the merge conflict, and I will show you how to deal with it. So currently we have just 1 branch on our GitHub repository with a couple of commits. Let's go ahead and create a new branch. I'm going to call it test. [No audio] So the new branch is created and now let's go ahead and make some commits on that newly-created branch. Suppose that we want to add some code inside of the script.js file, let's open it, then click the edit icon and create the variable, [No audio] assign to it some number value, let's say 100. Then down below, make a new commit, Created variable. [No audio] After that I'm going to make another commit. Let's add some more code to the script.js file, create another variable and assign to it some plain function, [No audio] and then again make the commit with the message, Created function. [No audio] Alright. So we made 2 new commits on the test branch, and now it is 2 commits ahead of the master branch. Now suppose that we or someone else is going to make a new commit on the master branch, and is trying to add it the same line of code which we have been working on, on the test branch. Let's see what will happen. I'm going to switch to the master branch. Then let's open script.js file, [No audio] and create a new variable with some string value. Let's say, 'Hi there', and then make a new commit, Created variable on master. [No audio] Alright. So all the changes are made, and now suppose that we need to merge those two branches. First of all we need to create a pull request. For that, let's go ahead and click the button, Compare & pull request. Once we do that, then we will go to the page which you're already familiar with. At a glance, you might notice that instead of the green check mark and the text, Able to merge, we have here red X sign and the text, Can't automatically merge. So, we are facing the merge conflict. The reason is that we tried to add it the same line of code from the different branches. Right after this warning text we can read, Don't worry, you can still create the pull request. So, despite the merge conflict, we still can create a pull request. Let's insert here some comments, let's say, Updated script.js file. [No audio] Then click the button. [No audio] So now we got here another warning saying that, This branch has conflicts that must be resolved. Right next to this text there is a button called Resolve conflicts. Let's go ahead and click this button. Now, as you can see, we've got here some text editor in which we should resolve the conflict. We have here both versions of the code and we need to choose which one to keep. GitHub itself cannot decide, it needs our help, and this is the exact same situation which we saw while we've been working on a local repository. Now we need to resolve the conflict manually. Suppose that we want to keep variable b instead of variable a, so let's delete all this stuff and just keep the variable b and the function. After that let's click Mark as resolved. So the conflict is resolved and we can proceed. Let's click Commit merge, [No audio] then click Merge pull request, and finally Confirm the merge. Alright. So again the conflict is resolved, pull request is merged and closed. Now we can feel free and Delete the test branch. [No audio] If we check the commits history, we will find here a couple of new commits, which were made automatically in the process of merging. Okay. So that's the way how we can resolve the merge conflicts on GitHub. Now we have to move on and learn about how to copy the remote repository on our GitHub account, how to clone it to our local computer, and how to contribute to someone else's repo. So, see you in the next video. [No audio]

[No audio] Alright. So, in the last lectures we learned about how to create pull requests, how to merge branches on GitHub, and also how to deal with the merge conflicts. Now, as I promised, I'm going to show you how to copy someone else's repository on our GitHub account, and how to clone it to our local computer. So in this lecture we will be looking at forking and cloning on GitHub. Those two options allow us to get access to any of the open source repositories, which are available on GitHub, work on them, make some changes and updates, and contribute those updates to the original project. To proceed with the lecture, we will need to use the second account, I have already created a brand new one. As you see, we don't have here any repositories. I'm going to search for the repository, which we have been working on so far, and then we will copy it to that newly-created account. So, first of all, to work with me along the way, let's go ahead and create a brand new account, or you can feel free and use any other accounts if you have one, of course. Once you have access to a different account, then go ahead and search for our working repository. In general, we can search for and use any of the open source repositories, which are available on GitHub, but for now I'm going to use our repository in order to make things a bit more easier. I'm going to search for the username of our previous account, because if we search for the name of the repo, which is gitProject, [No audio] we will get tons of results with the same name. So let's search for the username. In my case it's codeandcreateweb, but of course, in your case it should be your username. Then click here down below, the Users. So, here is our username and the repository as well. Alright. Now we have to copy the repository to our current account. To do that we have to click the button called fork, which is placed at the top right side of the page. Once we click it, then after a few seconds, [No audio] the entire repository will be copied to our new account. So, here it is. Now we have the exact same copy of the original repo, all the source files are here. Also if we go to the commits tab, we'll find all commits which have been made so far. So using the fork option, GitHub allows us to copy the original repository to our account. If we check out the original repository, then, right after the fork button, we'll find the number of forks. In this case it's 1. If we click it, then we'll find the owner of the original repo and the username who forked the repo. Alright. So that's it about forking. Now I'm going to show you how to clone the repository to our local machine. First of all, let's go to the terminal and create a new folder on Desktop. Currently we are in our working folder, so let's go up one directory. For that we need to run cd .., then I'm going to create a new folder. For that, as you remember, we had to use the command mkdir, and we have to specify the name of the folder. Let's say git-project-clone. [No audio] So once the folder is created, let's enter it. [No audio] Now let's go back to our GitHub account. So to clone the repository, we have to click this green button here, it's called Clone or download. Once we click it, then the link should be displayed. Let's go ahead and copy this link, then go back to the terminal and run the following command, git clone, and now we have to paste here the link, which we have just copied. [No audio] Alright. So, the repository is cloned. Let's run ls, which shows us the files and folders, which the current directory consists of. As you see, there is added a new folder called GITPROJECT. If we enter this folder, [No audio] then we'll find a copy of the original repository. We have here all the source files, and also if we run the history using�git log --oneline, then we will get all the commits that were made on the original repository. Alright. So that's the way how we can make a copy of the remote repository to the GitHub account using a fork and also how to clone the repo to our local computer using git-clone. Next I'm going to show you how to contribute to the open source repository, and for that let's move on to the next lecture. [No audio]

No audio] Alright. So in the last lecture we talked about how to copy the remote repository to our account using a fork, and also we saw how to clone it to our local computer. Now it's time to move on and consider how we can update the repository and then contribute to the original one. Nowadays there are tons of open source projects on GitHub, which different developers are working on. They copy this repositories, update them, and then send pull request to the owners to apply those changes and updates to the original project. On the other hand, the owners of the project received pull requests from developers, considered the changes, and if they're happy, then they merge updates to the original repository. This process is called contributing, and that's the way how different libraries and frameworks are updating today. Alright. So in this lecture, we're going to go through this entire process currently and working on the repository, which we cloned in the last lecture, is the exact copy of the original project, which we have been working so far. So, now I'm going to make some changes to this project. Here I have opened the clone project in VS code. Let's go to the index.html file and create some new elements. Let's say form element. [No audio] And inside of it I'm going to insert input and button [No audio] with a text, Click here. Then let's go ahead and add these changes to the staging area and make a new commit. For that I'm going to use again the shorter way. We have to run command git commit -am, with the message, "Created form element". [No audio] Let's run the history. [No audio] So we have here a new commit. Let's go ahead and add one more commit. I'm going to add some styles to the newly-created form element. Let's open style.css file and insert here some styles. Let's define margin and padding for form element. [No audio] And then again make a new commit [No audio] with a message "Styled form element". Alright. So that's it. We have updated the project, and suppose that we want to contribute these changes to the original repository. So, what should we do? First of all, we need to push these changes to our second GitHub account, I mean, the account where we forked the repository. For that we have to run git push origin master. [No audio] As you see, we've got here some error. Actually, that's the issue of Git and GitHub credentials, which we have already set up previously. We need a permission to push the changes to the remote repository. To do that, we have to change the credentials from the control panel. So let's go ahead and open it. Then we had to go to User Accounts, [No audio] select Manage Windows Credentials. Click here git and github, and then click Edit. [No audio] So, now I'm going to change the Username and the Password. You have to use your User name and Password of the second account. [No audio] So, once we do that, then we'll be able to push the changes to the remote repository. So, let's run once again, git push origin master. [No audio] Now, if we check our GitHub account and reload the page, [No audio] then we will find here the last two commits, which we have just made. Alright. Fine. So now we can send those changes to the owner of the original repository. For that we have to create a new pull request. We need to click this button here. [No audio] Now, as you see, GitHub is comparing the changes between two branches. In this case, those two branches are the master branch of the original repository, and the master branch of our second repo. In our case, there are no conflicts, and branches can be automatically merged. So we can feel free and Create a pull request. Let's click this green button here. Now GitHub is asking for to make some comments and descriptions. In the real world, when you try to contribute to some projects, you have to send some descriptive comments to make an easier and clear the merging process for the owners of the project. So let's put here some comments, write Updated project, [No audio] and then down below, I'm going to insert some descriptive text, let's say, Added a new form element in index.html file, [No audio] and styled it from the style.css file. [No audio] So now we're ready to Create a pull request, let's click this green button here. Okay. So the pull request is created. As you might notice, we're not allowed to merge the branches from here, because it's the job of the owner of the project. Now let's go ahead and switch to the owners account, I mean the account which owns the original project. If we reload the page, then we'll get here one Pull requests. Let's click it. So you can find here the pull request which we have just made from the second account, with all its comments and changes. We can review the Commits made by the another user. Also we can review all the changes, which were made to the project. As you see, there are two files changed, index.html and style.css. The code in green color with + signs is the updated version of the file in both cases. So, after checking the changes, if we are happy with them, we can merge the branches. Actually you can write back to the user some text, let's say Great job, [No audio] and then you can merge the branches. [No audio] So the original project is Updated. If we check the commits tab, we will find all the commits made by another user, followed by the commit about merging their branches. Alright. So that's the way how we can contribute to some open source repository on GitHub. Now I'm going to show you what happens if there is some merging conflict. Let's make some change from the second user. I'm going to do that straight from GitHub. Let's increase the size of the margin and make it 30px. [No audio] Also make a new commit, Increased margin. [No audio] Then suppose that before we make the pull request, in the mean time, the owner of the project changed the size of the margin as well. To imitate this process, let's switch back to the owner's account, and change the size of the margin. Let's make it 25px, and then make a new commit, [No audio] Changed margin. [No audio] After that, let's create a pull request from the second account. [No audio] Click the button here. So, as you see, we have a conflict, because both users were trying to edit the same line of the code. So what can we do here? Firstly, we can Create the pull request anyway, despite the conflict. [No audio] Next we can either Resolve the conflict from here or we can leave it as it is and let the owner to do that. The owner of the project will get the pull request with conflict, anyway. So, if we switch back to the owner's account and check the Pull requests tab, we will find the request with the conflict. Now we can Resolve the conflict from here, as it was in the previous case. Suppose that we prefer to increase the margin to 30px, so let's make here the appropriate changes. [No audio] then, Mark as resolved. [No audio] Commit merge. [No audio] Now GitHub warns us about the merging. Click this button and, finally merge pull request [No audio] If we checked the Commits history, then we'll find all the commits made on both accounts, including the Merge commits. [No audio] Alright. So that's it about contributing to the original open source repository. I hope it makes sense to you. Let's move on to the next lecture. [No audio]

Hello and welcome to our new section, in which we're going to be talking about one of the most important topics regarding GitHub, and it is collaboration. In the previous section we discussed how we can copy the remote repository to our GitHub account, then how to clone it to our local computer, make some changes and updates, and eventually contribute to the original project. This process is something like collaboration, but in fact it's not. In the case of collaboration, the owner of the project can add different developers as collaborators, which then will have almost identical rights to manage the project. I'm saying almost because there might be some rules which should be protected by the collaborators. Alright. Let's see in practice how to collaborate on GitHub. First of all, I'm going to create a brand new local repository. I have created a brand new folder on the Desktop, called git-collaboration, which then I opened in VS code and also with terminal here. Let's go ahead and quickly initialize a git repository and then make a couple of commits. Let's run git init. [No audio] So the .git folder is created. It means that we have initialized git repo. Next I'm going to create two different files, index.html and style.css [No audio] Let's add them to the staging area, run git add ., then check the status, and make a commit, run git commit -m with the message "Created index.html and style.css files". Let's run the history. [No audio] So here is our first commit. Alright. Next let's insert the basic structure of the HTML document inside of the index.html file, then make a commit. In this case, I'm going to use a shorter way. Let's run git commit -am [No audio] with the message "Added basic html structure". [No audio] And then run again git log--oneline. [No audio] So we have here 2 commits. Let's make some more commits. I'm going to create an h1 heading element with the content, This is heading element. [No audio] Then make a commit, run git commit -am. [No audio] with the message "Added h1". And finally, make one more commit. Let's open style.css file and create some reset styles. I'm going to set margin and padding for all the elements as 0. [No audio] Then make again a commit [No audio] with the message "Added reset styles". And lastly, let's check the history. Run git log--oneline. So, as you can see, overall we had 4 commits and we're good to go. Now the next thing that I'm going to do is to create the remote repository and push the local repository into it. Let's go ahead and go to our GitHub account. Then I'm going to click the + sign here and select a New repository. I'm going to call this repo the same name, I mean git-collaboration, then click this green button here. [No audio] We've got here some instructions, and now I'm going to copy the link from here. Alright. Let's go to the terminal and push the local repo to the remote repository. For that we have to run git remote add origin, and we have to Paste here the link of the repo, and finally we have to run git push origin master. [No audio] Actually you might get here the error about the permissions. If so, then you have to change again the credentials from the control panel as we did in the previous lecture. Okay. So if we go to the GitHub Page and reload the page, we will find here our repository. Alright. So now we're ready to set up the team of collaborators. For that I'm going to use again a different account. So, in this case, I'm the owner of the project and we have to add a collaborator to the project. To do that we have to go to Settings. Then on the left hand side we have to select Collaborators tab where you can find the input fields in which we have to enter the username of a collaborator. Let's type here codeandcreateweb123. As you see, GitHub has found the user automatically, let's select it, and then click the button, Add collaborator. So, we added the user as a collaborator, but on the other hand we need to get the response from that user, I mean the collaborator should accept the invitation. If we check the GitHub account of that user, then there won't be any notifications or invitations. So, what should we do? We have to check the email. Let's go to the Gmail account of that user. As you can see, we've got here an email from the GitHub saying that the owner of the project is inviting us as the collaborator for the project. We should click here View invitation, and then we have to Accept the invitation. Once we do that, then we will redirect it to our GitHub account, and as you can see, now the repository is available for us. Also, if we go back to the owner's account and reload the page, [No audio] the user will be added as a collaborator, we are no longer waiting for the response. Alright. So in this lecture we created a brand new repository and added a collaborator to the project successfully. See you in the next one. [No audio]

[No audio] In the last lecture we created the brand new repository, pushed it to the GitHub account, and also we added successfully a collaborator to the project. Now the collaborator has access to the repository and he or she can manipulate on the project. Let's go ahead and see how we can work on the repository as the collaborator. By the way, to imitate the collaboration process, and actually to work properly, I'm using a different computer. Otherwise we won't be able to work as a collaborator in the right way. If you have a different computer, then you can use it for the second account, but if not, then you can just watch the videos. If you use just one computer, then you will get different results than us, so it might be slightly confusing. Alright. So the first thing that you have to remember is that we don't need to fork the repository anymore. We just need to clone it to our local computer. As you already know, to clone the repository, we have to click this green button here called Clone or download. It will generate the link of the repository. Let's copy it, then go to the terminal, and create a new folder. Currently we are on the Desktop. Let's create a new folder, [No audio] call it git-project-collab, [No audio] then go into this folder, and finally clone the repository. We have to run git clone, and we have to Paste here link of the repository. [No audio] Alright. So we have successfully formed the project, let's check the directory. As you see, we have here a folder called git-collaboration. Let's go ahead and enter this folder. [No audio] So if we run the history of the commits using git log --oneline, then we will get all 4 commits made by the owner of the repository. Also, if we open the folder in VS code, [No audio] we will find here index.html and style.css files with the proper content. Alright. So now let's go ahead and make some changes to the project. Let's add some styles to the h1 heading element. For that I'm going to edit the style.css file. Let's increase the font-size of h1 heading element, [No audio] then make a new commit with the message� "Increased font size of h1". [No audio] Let's run the history. [No audio] So the commit is added successfully, let's make one more commit. I'm going to change the color of h1 element. Let's make it blue, [No audio] and then make again a new commit [No audio] with the message "Changed color of h1". [No audio] Finally lets run the history. [No audio] So we have here 2 new commits made by the collaborator. Alright. Now it's time to push these changes from the local repository to GitHub. For that we have to run git push origin master. [No audio] So the command is run successfully. Let's go ahead and check the GitHub accounts. [No audio] As you can see, the changes are pushed fine, and the repository is updated accordingly. We didn't need a pull request or some permission from the owner of the project. We have pushed the changes to the GitHub without any problem. If we check the commits tab, we will find 2 new commits made by the collaborator. Also, if we check the account of the collaborator, we will get the same situation. So in general, this kind of collaboration is not quite effective because the owner doesn't have any controlling mechanism and that might cause some unexpected results. Alright. Let's stop here and move on to the next lecture in which I will show you how to protect the branches and how to control the collaboration process. [No audio]

Alright. So in the last lecture we pushed the changes from the local repository to GitHub as the collaborator. We did it without any problem, I mean, without any pull requests or permission from the owner of the repository. As I said in the previous lecture, in the real world, that situation is not quite convenient. If you are the owner of the project and you have some collaborators in your team, then you might need some control mechanism to check others' work and avoid any unexpected results. To be more precise, GitHub allows the owners of the repositories to protect the branches by requiring the reviews before merging the branches. Alright. Let's go ahead and figure out what I'm talking about. I'm going to protect the master branch. In most cases, in the real world, the master branch is deployable and nobody touches it for testing purposes, so it should be protected. To protect the branch, we have to go to Settings, and then we have to click Branches. Here we see a little section called Branch protection rules. On the right hand side of the heading, we can find the button called Add rule. Let's go ahead and click the button. So we've got here an input field in which we should indicate the name of the branch, and down below we can find some rule settings. Let's enter here the name of the branch master. [No audio] After that I'm going to take care of the Rule settings. So, in this case, I want to use the first rule, which is Require pull request reviews before merging. So once we check this checkbox here, then all commits must be made to a non-protected branch. In this case, the branch that is not master and have to be submitted via a pull request, with the required number of approving reviews. In other words, the pull request can be accepted by the reviewer to merge the branches. As for the number of approving reviews, it should be selected from this dropdown menu. In our case we have just one collaborator, so as the number of approving reviews I'm going to select 1. Then finally click this green button here and create the rule. [No audio] Alright. So as you can see, now the rule applies to the master branch. Let's go back to the terminal and continue working. I want to remind you that currently I'm working from a different computer to imitate the process of collaboration properly. I'm going to edit the index.html file. Let's create the paragraph with some dummy text, and then make a new commit [No audio] with a message. "Added p". [No audio] So now we need to push the changes to GitHub, but we can't do it over the master branch because it's protected. If I run git push origin master. [No audio] then we will get an error, because, as we said, the master branch is protected. So what should we do? We have to create a new branch, push it to the GitHub and then merge it to the master branch. So let's go ahead and create a new branch. For that, let's run git checkout -b followed by the name of the branch, let's say test. As you remember, this command allows us to create a new branch and switch to it at the same time. Let's run the history. [No audio] So here we have all the commits which have been made so far. I'm going to add a new commit on the test branch. Let's change the font-size of the p. [No audio] Make it 30px [No audio] and then make a new commit [No audio] with the message "Increased font size of p". [No audio] Finally, let's run the history. [No audio] As you see, the test branch is 1 commit ahead of the master branch. Alright. So now we can push the changes over that newly-created branch on GitHub, run again git push origin, [No audio] and then indicate here the name of the branch test. [No audio] So, once this command is run, if we go to the collaborators account, we will find here the pushed test branch in a slightly different color. So now we have to compare changes and create a pull request. Let's go ahead and click this button here. [No audio] Then I'm going to change here the comments, let's enter here, Added and styled p, [No audio] and then click the button to Create a pull request. [No audio] So, now we've got here some warnings. The first one says that the Review is required. As for the second one, it says that the merge is blocked. And also you can see that we are not able to Merge the pull request, because the button is disabled. Let's go ahead and check the owner's account, reload the page. As you see, we have here 1 Pull requests. Let's go ahead and open it. [No audio] You can find here the same warnings, Review is required and Merging is blocked. But despite this I can still Merge the pull request as the owner of the repository. Alright. So what should we do? First of all, we have to set the reviewer, I mean the user which will review the changes and approve the pull request. Here on the right hand side of the page we see the Reviewers section. Also, here is a hint saying that At least 1 approving review is required to merge this pull request. That comes from the Rule settings, which we have just set up. Let's go ahead and set the reviewer. For that we have to click this icon here. As you can see, GitHub suggests automatically to set the owner of the repo as a reviewer. Let's select it. After that, we have to go to the Files Changed tab, where we can find all the changes made by the collaborator. So after reviewing the changes, if the owner is happy with them, then he or she should approve them. At the top right corner of this section, we can find a green button called Review changes. If I click it, then the input field will be displayed with some options. I mean the options like Comment, Approve, and Request changes. In this case we have to select Approve. Also let's add here some comments, Good job, and click Submit a review. [No audio] So, now the owner of the project has approved the changes made by the collaborator, and we can simply merge the pull request. [No audio] If we check the Commits history, we will find all the commits made by the collaborator, followed by the Merge commit. The same we can find if we check the collaborators account. [No audio] Alright. So that's the way how we can protect the branches and control collaborator's work on GitHub. Okay. Congratulations. We're done, and our course is over. I hope it was interesting, you enjoyed it and learned lots of different things regarding Git and GitHub. Nowadays, every developer should know how to deal with the most popular and widely used version control systems. After finishing this course, you will have all the needed skills and knowledge about Git and GitHub. I want to remind you that you have lifetime access on the course. So, you can come back anytime, recall some important topics about GitHub. Alright. Thanks for being with us. Wish you all the best. Bye bye. [No audio]