

Practice 4: Git

Diseño Software (614G01015)

Eduardo Mosqueira Rey (Coordinador)

Departamento de Ciencias de la Computación y Tecnologías de la Información
Facultad de Informática



UNIVERSIDADE DA CORUÑA

Table Of Contents

1 Introduction to Git

- Version Control Systems
- Git
- Centralized Workflow

2 Creating a Git Repository

3 Git and IntelliJ IDEA



Version Control

Version Control System (VCS)

A system that records the changes made over time in a file or set of files, so you can go back to specific versions later.

- Functioning is more effective with plain-text files that can be simultaneously modified by different users.
- Gives you a backup of the developed code.
- Gives you a log of the changes made in the code.
- Lets you share code easily, facilitating collaborative development.



Basic Concepts

■ Repository:

- The place where the source code is stored in all its different versions.



REPOSITORY

■ Working copy:

- The current version of the code that is under development and subject to changes.
- Visible only for the local user that is editing it.



■ Commit:

- Moves the working copy to the repository.
- Includes an explanatory text and generates an identifier to indicate the version.

WORKING COPY

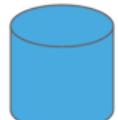
■ Update:

- Brings the working copy up to date with the repository, merging the changes in both.
- Can result in conflicts if two users have modified the same line of code concurrently.



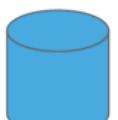
WORKING COPY

COMMIT



REPOSITORY

UPDATE

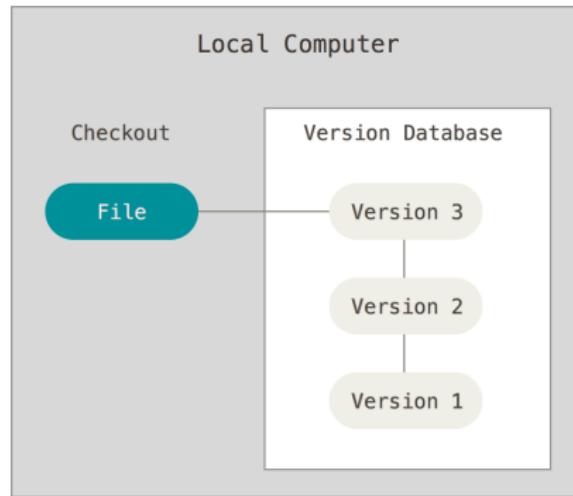


REPOSITORY

Local Version Control

Local Version Control

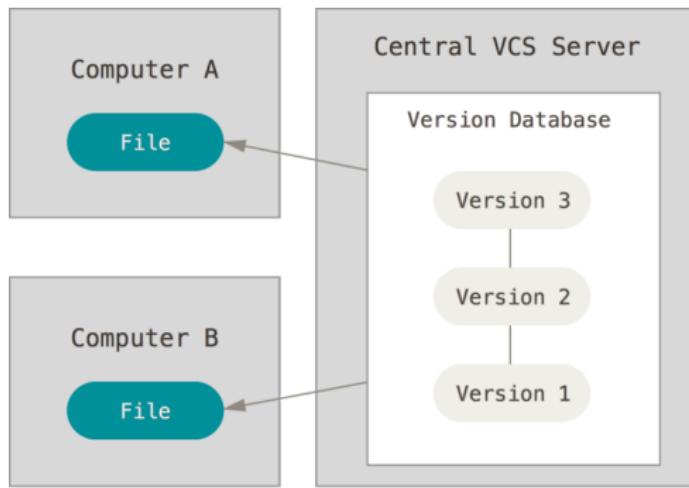
Both the working copy and the repository are stored in the local computer.



Centralized Version Control

Centralized Version Control (*Centralized VCS or CVCS*)

The repository is in a server in the cloud. Multiple clients download their own local copies from that place (Subversion).



Centralized Version Control

■ Advantages

- Facilitates collaborative development.
- Everybody can know what the other collaborators are working on.
- Administrators have extensive control over what each user can do.

■ Disadvantages

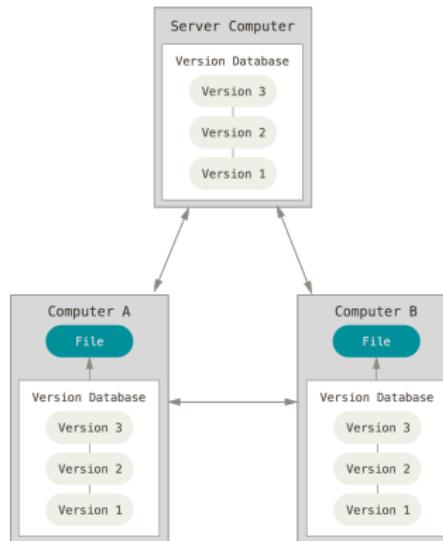
- As development is centralized in the server, the workflow is not robust to server problems.
- If the server goes down, the collaborative workflow is interrupted.
- If there are storage failures in the server, and no *backups* have been made, the project's history may be lost (only local snapshots would survive).



Distributed Version Control

Distributed Version Control (*Distributed VCS or DVCS*)

It also consists of a repository in a remote server and local working copies for the clients but, additionally, clients also store a copy (a clone) of the remote repository (Git or Mercurial).



Distributed Version Control

■ Advantages

- If a server stops working, any of the repositories available on the client side could restore it (each clone is complete copy of the repository).
- Allows establishing workflows that are not possible for centralized systems.
- Most of the operations are done locally, so you don't need an internet connection to use it.

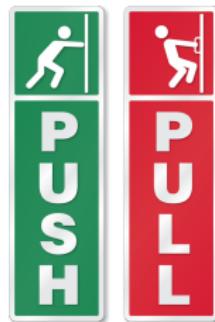
■ Disadvantages

- Adds an extra layer of complexity compared to centralized versions.
- You must follow an orderly workflow to avoid problems when dealing with the different options that are available.

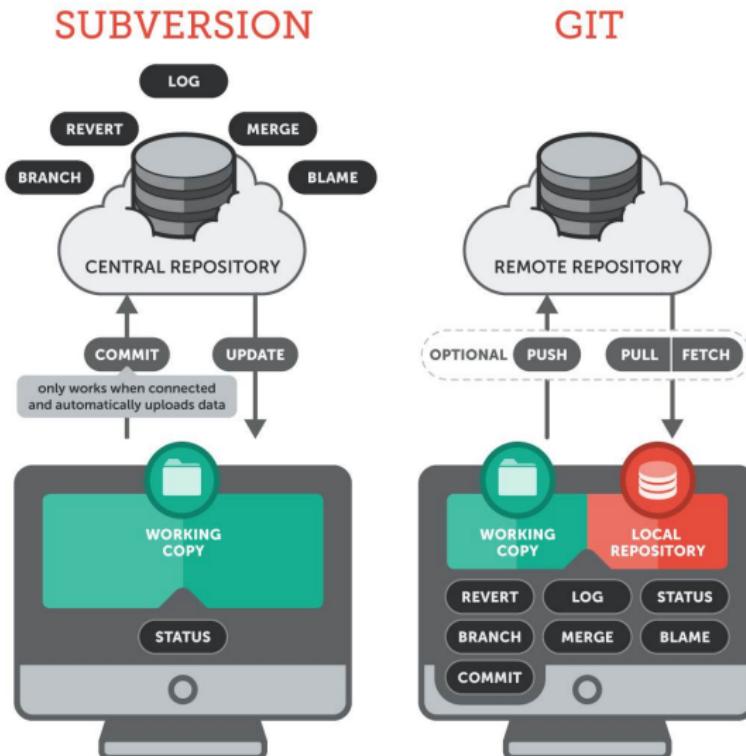


Distributed Version Control

- Distributed VCSs add new operations to the usual *commit* and *update* operations. Namely, they add two operations called *push* and *pull* for communicating with the remote repository.
 - **Push:** Updates the remote repository with the contents of the local repository.
 - **Pull:** Updates the working copy and the local repository with the new changes at the remote repository.



Centralized vs. Distributed Workflow



The Git version control system



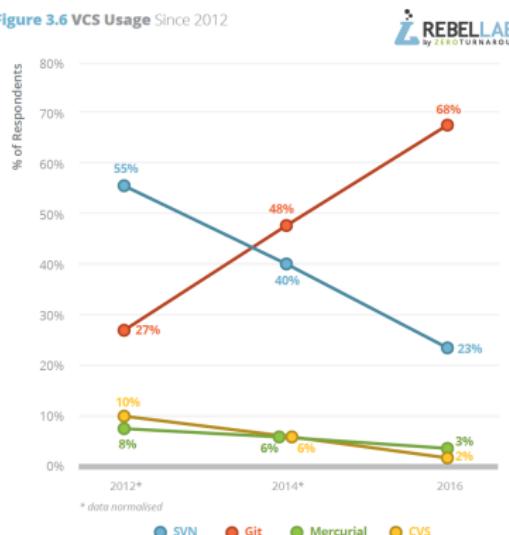
- Created by Linus Torvalds in 2005 for *Linux* development.
- It is a distributed version control system.
- Its goals are influenced by the needs of a large-scale project like Linux, i.e., speed, simple design, big support for non-linear development (thousands of parallel branches), suitable for managing big projects, etc.



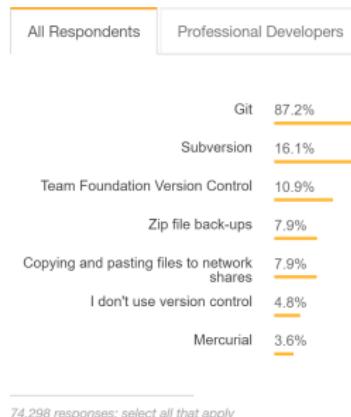
VCS Technologies

- Git currently dominates VCSs, to the detriment of the formerly more popular SVN¹.

Figure 3.6 VCS Usage Since 2012



Version Control



¹ <https://www.jrebel.com/blog/java-tools-and-technologies-2016>

Git solutions in the cloud

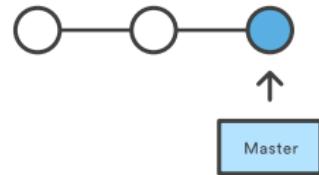
- Git became popular mainly thanks to cloud services, which add a social layer to code development.
- The most popular one is **GitHub** but widely-used alternatives exist, such as **BitBucket** or **GitLab**.
- GitLab has an *open source* version that you can download and install at your own server for free.



Git Concepts

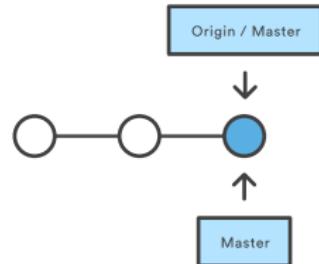
■ Master branch:

- Git structures source code in branches. There is always a default branch, usually called `master`.
- `Master` basically points to the latest revision of the code committed to the local repository.



■ Origin/Master branch:

- Git has a remote repository called `origin` by default.
- The remote repository has its own `master` branch, called `origin/master`.
- When we download a repository, both `master` branches point to the same place.

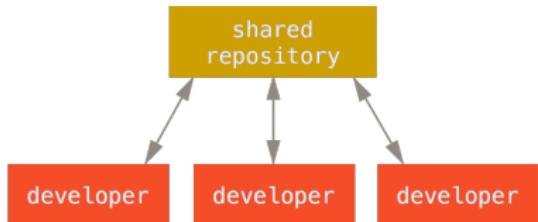


Centralized workflow

Centralized workflow

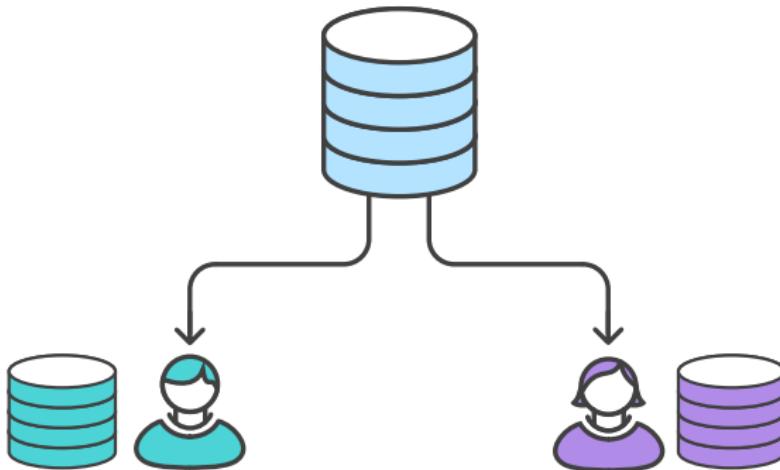
A central repository maintains the work of all developers and all of them synchronize with it.

- It is a similar workflow to that of centralized systems like Subversion.
- Useful for simple projects with few developers.
- Does not really exploit Git's capabilities, but it is suitable for this course.
- Other, more complex workflows can be consulted at: <https://www.atlassian.com/git/tutorials/comparing-workflows>



Centralized workflow

- You create a central repository that is locally cloned by all users.



Centralized workflow

- Imagine John is working on a particular feature.



Centralized workflow

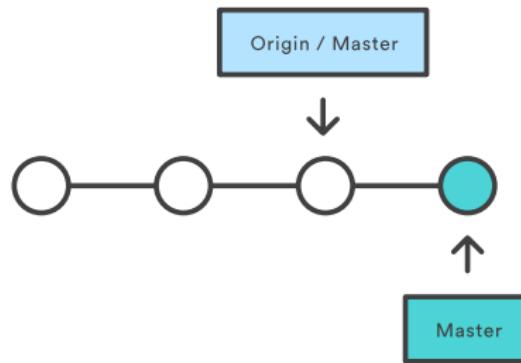
- Mary is working on another feature in parallel.



Centralized workflow

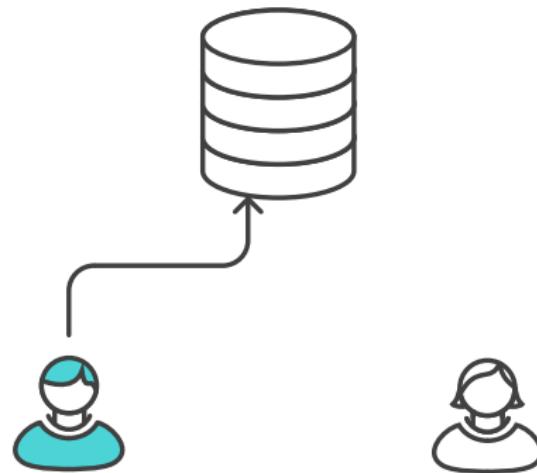
- John *commits* his changes to his local repository.
- Is master branch is ahead of the origin/master branch.

Before Pushing



Centralized workflow

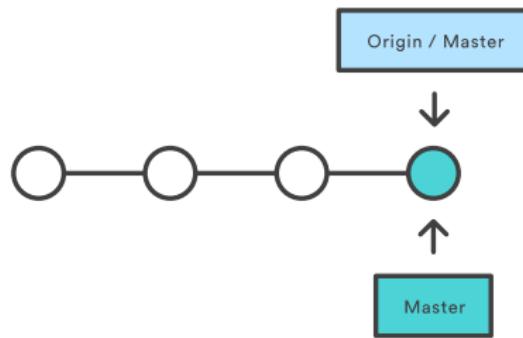
- John *pushes* his changes to the central repository.



Centralized workflow

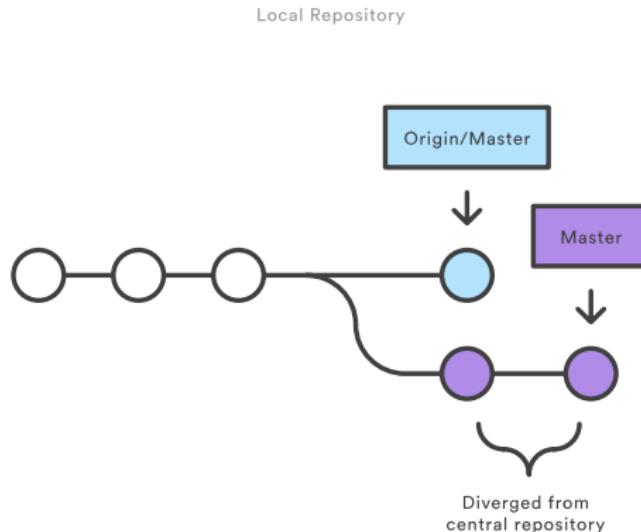
- To John, his local master branch and the origin/master branch point to the same place.

After Pushing



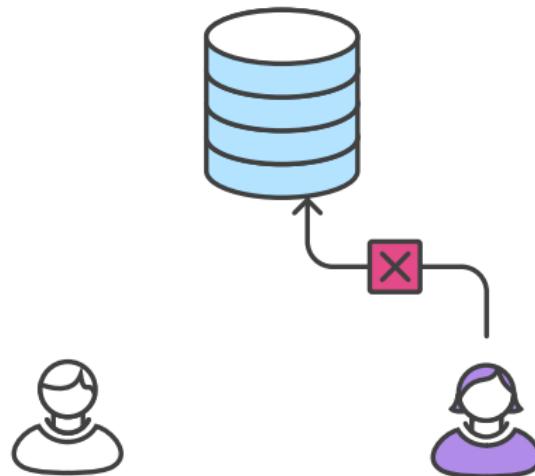
Centralized workflow

- Mary *commits* her changes to her local repository, unaware that the remote repository has changed.
- This means that Mary's changes diverge from the central repository.



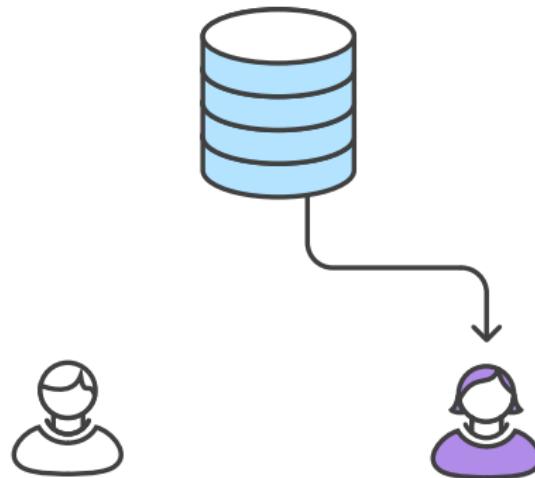
Centralized workflow

- When Mary tries to *push* her changes to the remote repository, she gets an error, because it has changes that are not in her local repository.



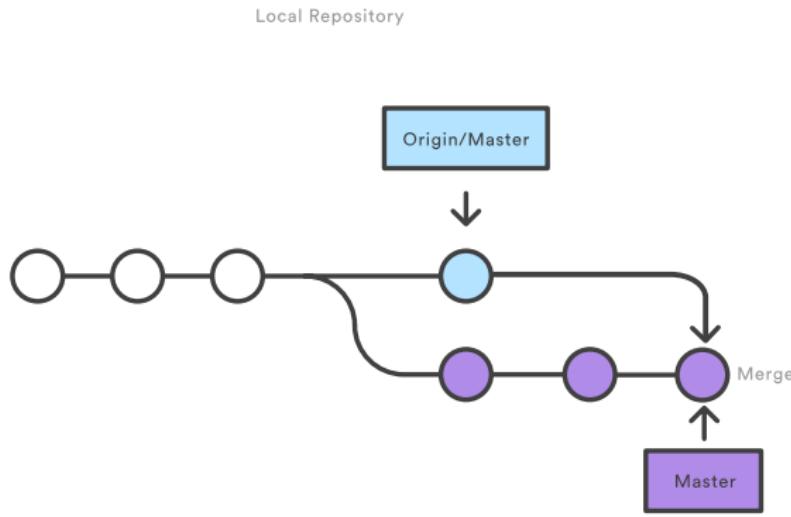
Centralized workflow

- Mary has to *pull* the remote changes and *merge* them with hers.



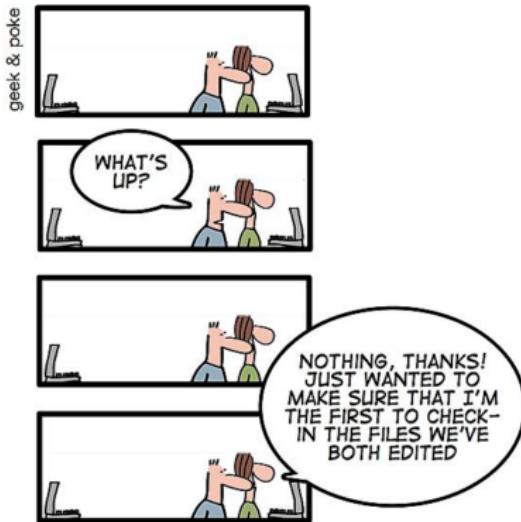
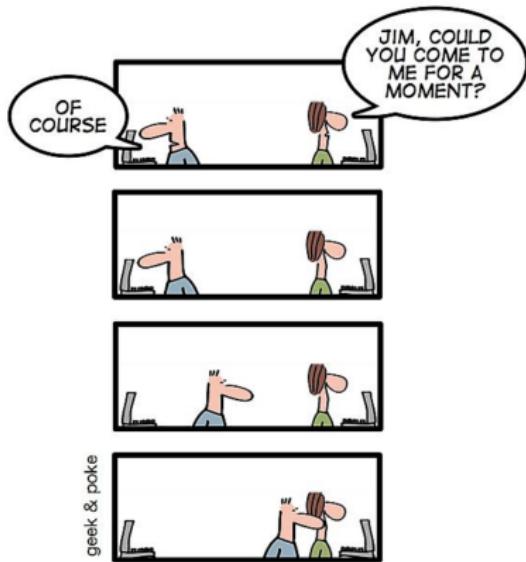
Centralized workflow

- The *merge* may work correctly, or it may cause conflicts that Mary has to resolve.
- Once they've been resolved, a new local *commit* is created containing both John and Mary's changes.



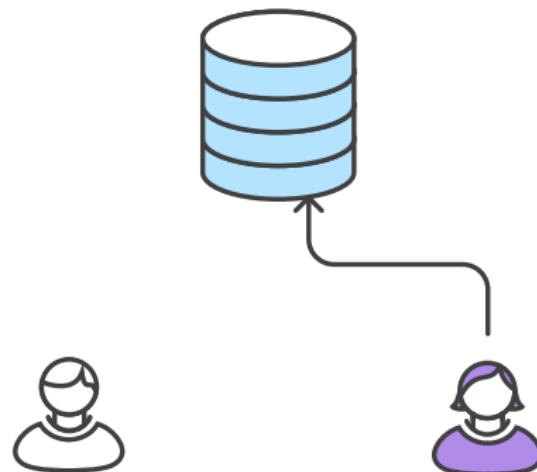
It is important to be the first to *commit* changes...

BEING A CODER MADE EASY



Centralized workflow

- When the *merge* has been made successfully, Mary must *push* it to upload the resulting changes to the central repository for the other developers.



Centralized workflow

- Now, John's repository is the one that is behind, compared to the remote repository, so John must update it doing a *pull*.

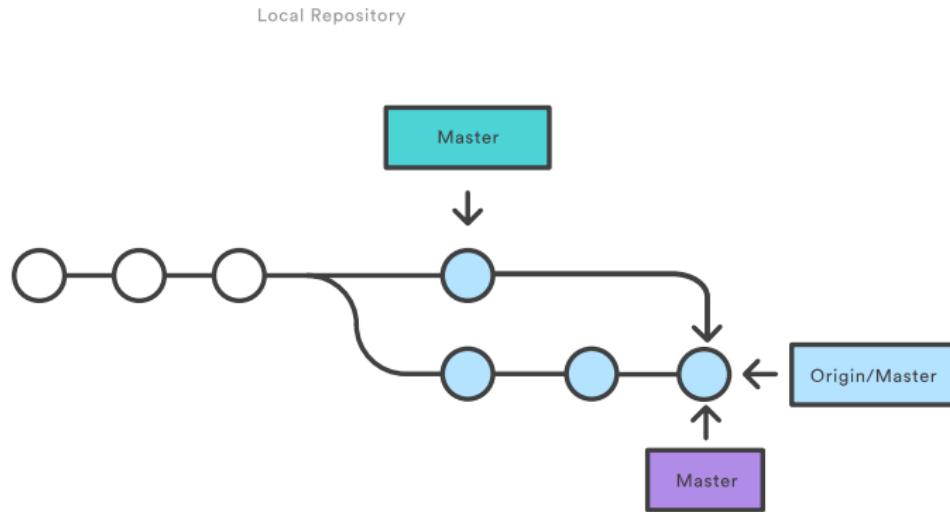


Table Of Contents

1 Introduction to Git

2 Creating a Git Repository

- Git Repository on GitHub
- Installing Git

3 Git and IntelliJ IDEA



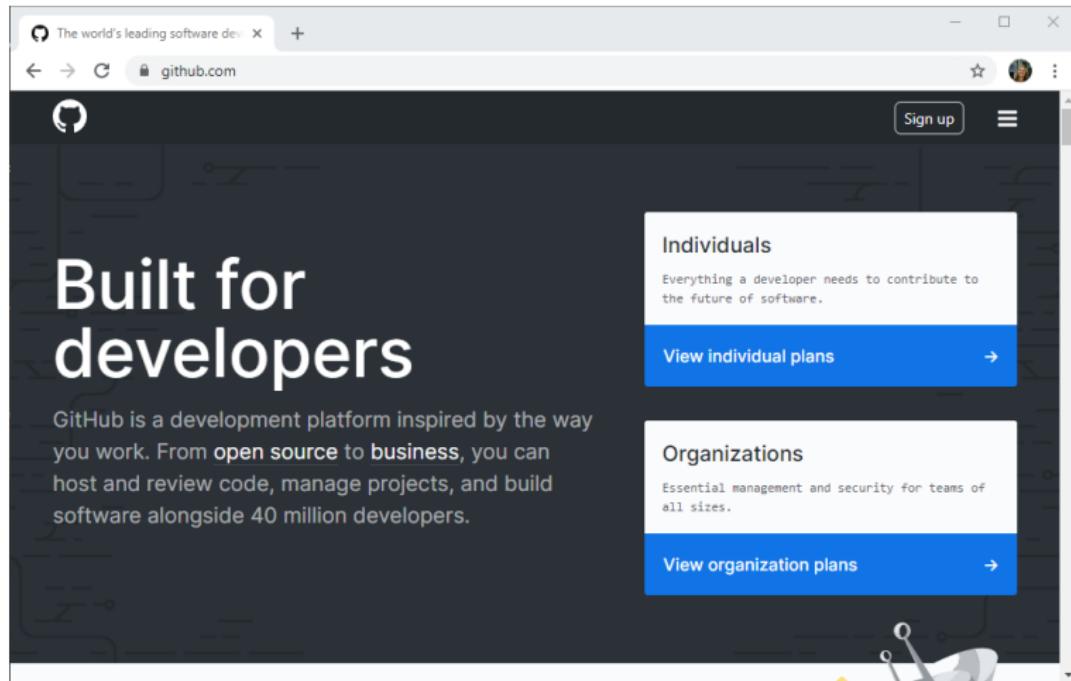
Git Repository on GitHub

- We will create a repository in Git by following the steps below:
 - 1 We will create a GitHub account (if not already created).
 - 2 One of the group members will create the repository with the name of the group.
 - 3 He or she will then include the other member of the group as a collaborator.
 - 4 Finally, the teachers of the subject as collaborators must be included as collaborators too.



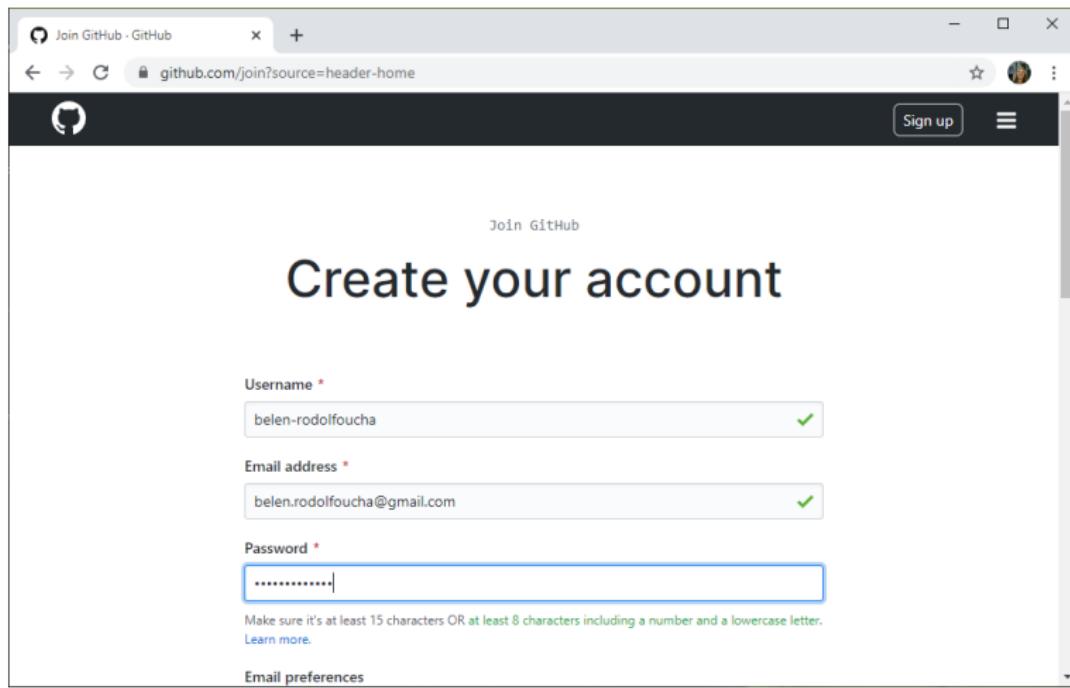
Creating an account on GitHub

- We will go to <https://github.com/> to create an account (if we don't already have one).



Creating an account on GitHub

- We will need to provide a username, email address and password.



The screenshot shows a web browser window for GitHub's account creation page. The URL in the address bar is `github.com/join?source=header-home`. The page title is "Join GitHub". The main heading is "Create your account". There are three input fields: "Username" with value "belen-rodolfoucha", "Email address" with value "belen.rodolfoucha@gmail.com", and "Password" with value "*****". Below the password field is a note: "Make sure it's at least 15 characters OR at least 8 characters including a number and a lowercase letter." A "Sign up" button is visible in the top right corner.

Join GitHub

Create your account

Username *

 ✓

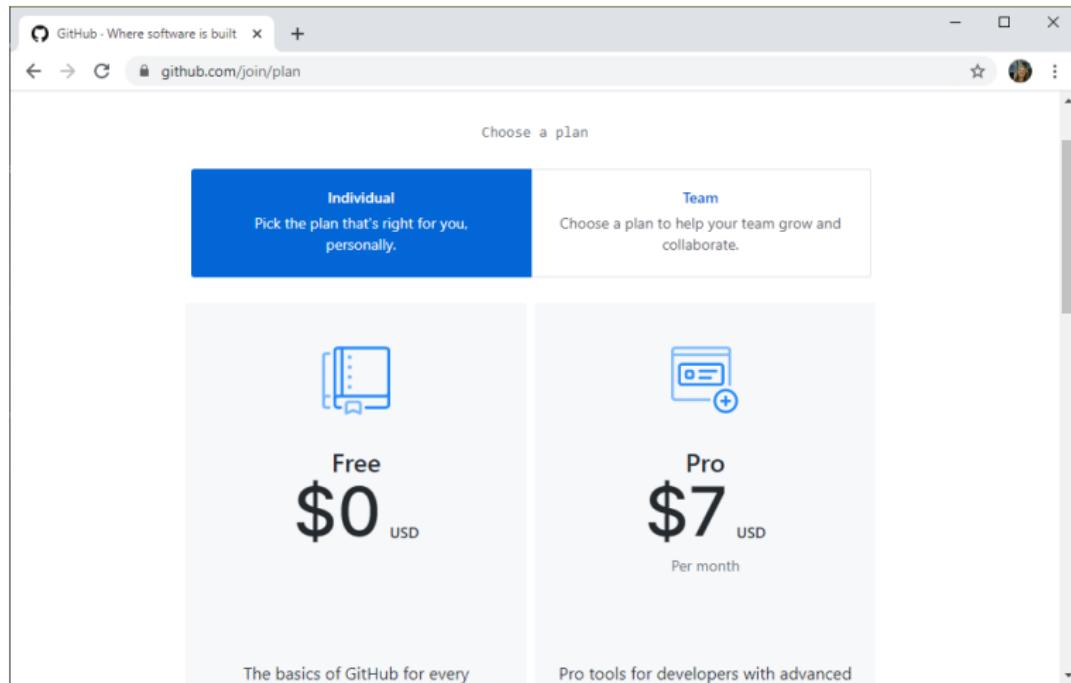
Email address *

 ✓

Password *

Creating an account on GitHub

- We will create a free account. It has certain restrictions but they do not affect us.



Creating an account on GitHub

- GitHub asks you questions to get to know you better, but you can skip this step if you're not interested in answering them.

The screenshot shows a web browser window for GitHub at github.com/join/customize. The title bar says "GitHub - Where software is built". The main content area has a dark header with a profile icon and a bell icon. Below it, a message says "Selected plan: Free". The central part of the page features a large "Welcome to GitHub" heading. Below it, a message reads: "Woohoo! You've joined millions of developers who are doing their best work on GitHub. Tell us what you're interested in. We'll help you get there." At the bottom, there's a section titled "What kind of work do you do, mainly?" with four options: "Software Engineer" (I write code), "Student" (I go to school), "Product Manager" (I make things), and "UX & Design" (I design things). A small circular logo with horizontal stripes is visible in the bottom right corner of the slide.

Creating an account on GitHub

- When you finish creating your account, you'll be asked to verify your e-mail address before you start using GitHub.

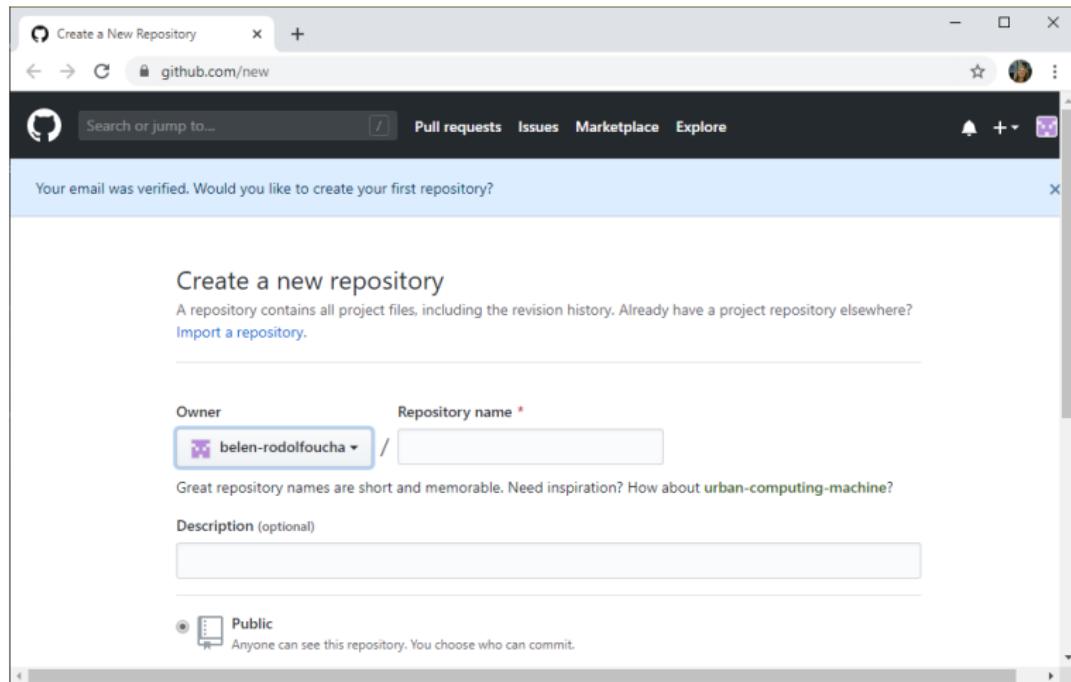
The screenshot shows a web browser window with the URL github.com/account/unverified-email. The page has a dark header with the GitHub logo and a navigation bar. The main content area features a GitHub octocat icon and the text "Please verify your email address". Below this, a message reads: "Before you can contribute on GitHub, we need you to verify your email address." It then states: "An email containing verification instructions was sent to belen.rodolfocha@gmail.com". At the bottom, there are two buttons: "Resend verification email" and "Change your email settings".

GitHub

Subscribe to our newsletter
Get product updates, company news, and more.

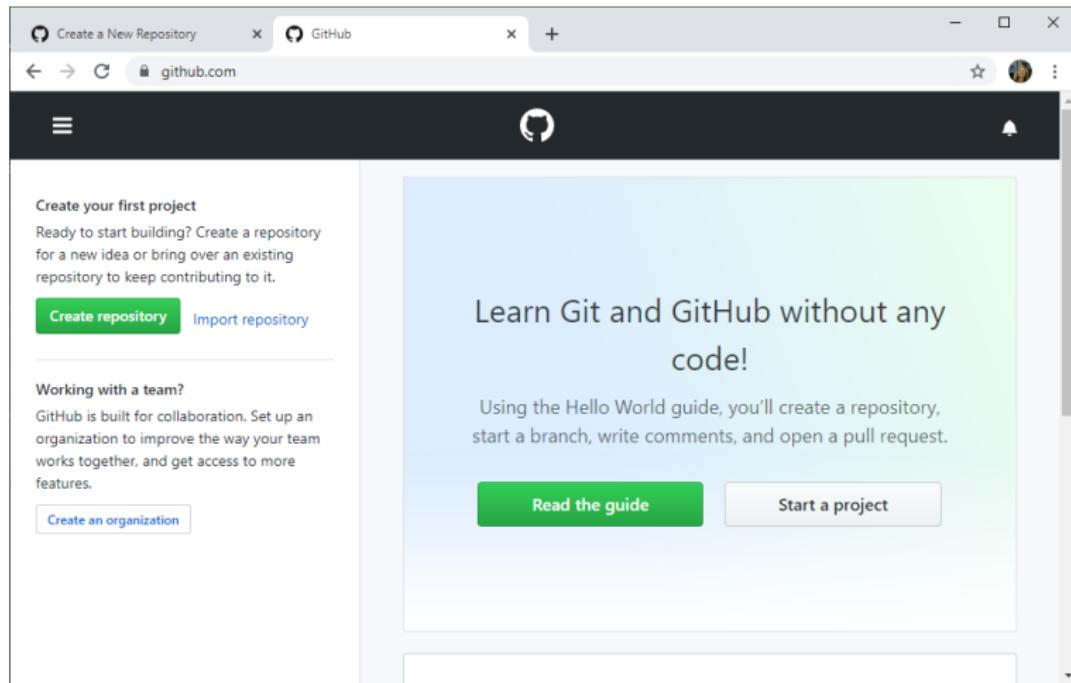
Creating an account on GitHub

- Once the GitHub address is verified, it automatically goes to the option of creating a new repository.



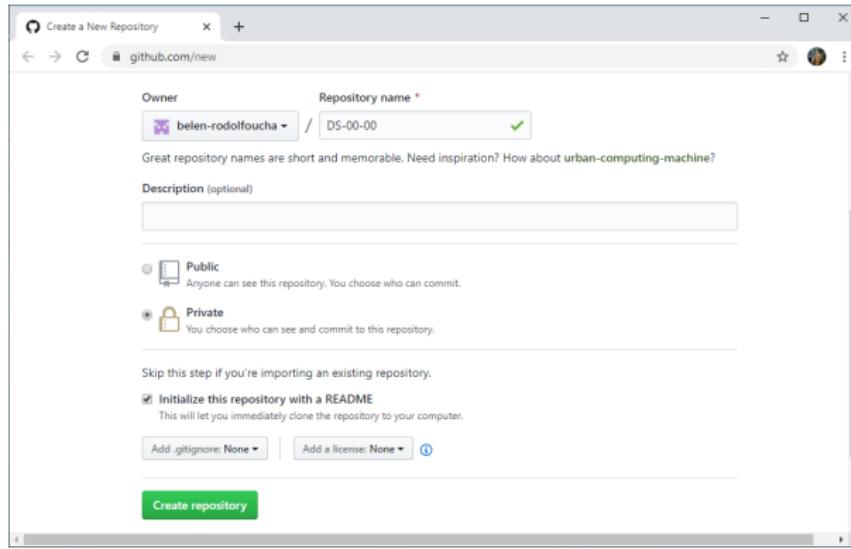
Creating an account on GitHub

- If you don't want to create it now you can always create it later by going to the option *Create repository*.



Creating an account on GitHub

- We are now going to create a repository with the name of your practice group (as you have registered it in the Moodle wiki), for example: DS-00-00, which should be **private**.
- We will indicate that it initializes the directory with a README.



Creating an account on GitHub

- We already have the repository in GitHub created, a commit has been made that has included the README .md file in it.

The screenshot shows a GitHub repository page for the user 'belen-rodolfoucha'. The repository name is 'DS-00-00'. The page includes the following details:

- Code**: 1 commit, 1 branch, 0 packages, 0 releases.
- Branch**: master
- Actions**: New pull request
- Files**: README.md (Initial commit, now)
- Commit**: belen-rodolfoucha Initial commit (17de042 now)
- Repository Name**: DS-00-00

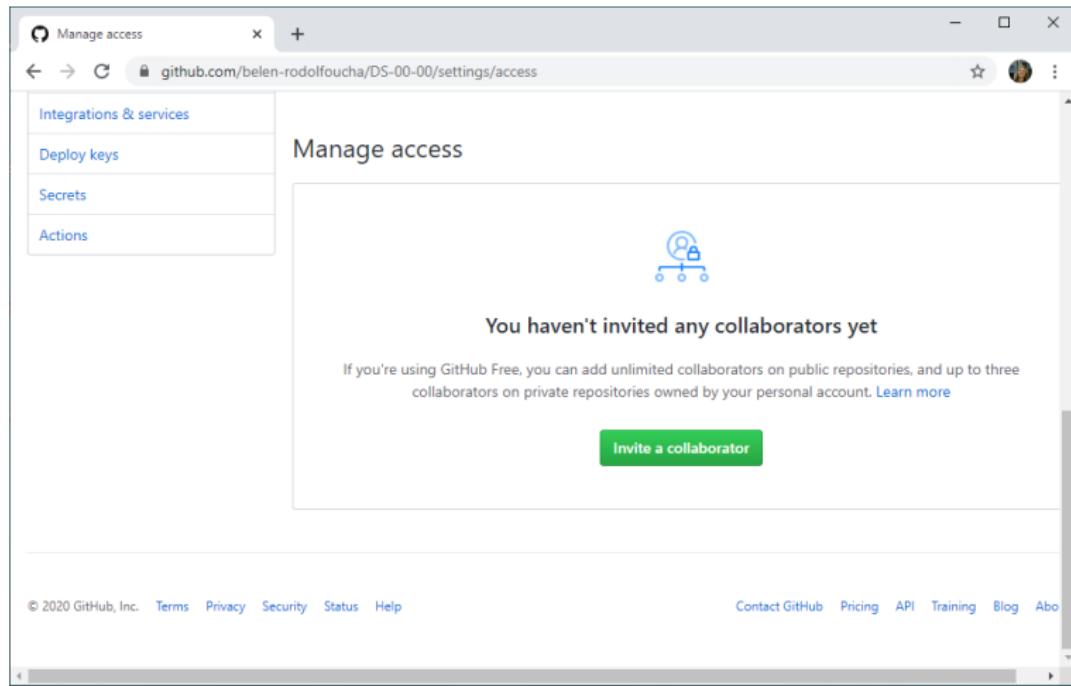
Adding your partner to your Git repository

- The next step is to give your partner access to the repository. You will need to go to *Settings* → *Manage access...*

The screenshot shows the 'Manage access' settings page for a GitHub repository named 'belen-rodolfoucha / DS-00-00'. The repository is set to 'Private'. The 'Settings' tab is selected. On the left, a sidebar lists options: Options, Manage access (which is selected), Branches, Webhooks, Notifications, Integrations & services, Deploy keys, Secrets, and Actions. The main content area is titled 'Who has access'. It shows two sections: 'PRIVATE REPOSITORY' (with a lock icon) and 'DIRECT ACCESS' (with a people icon). Under 'PRIVATE REPOSITORY', it says 'Only those with access to this repository can view it.' and has a 'Manage' button. Under 'DIRECT ACCESS', it says '0 collaborators have access to this repository. Only you can contribute to this repository.' A large blue button at the bottom says 'Manage access' with a gear icon. At the very bottom, there is a message: 'More about managing access to your repository'.

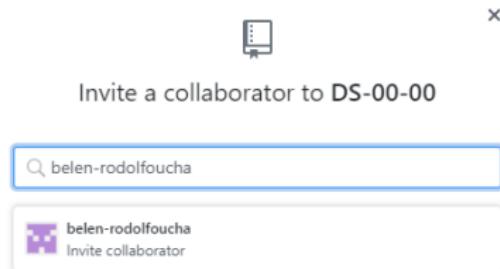
Adding your partner to your Git repository

- ...and make scroll and click on *Invite a collaborator*.



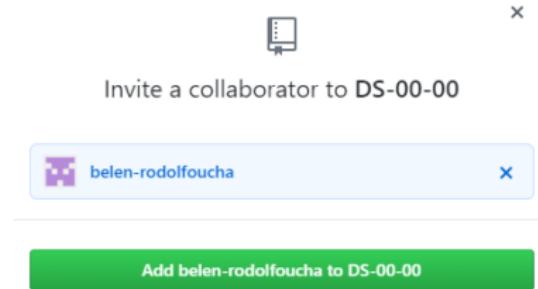
Adding your partner to your Git repository

- You look for your partner by its username...



Adding your partner to your Git repository

- ...and you add it as a repository collaborator.



Adding your partner to your Git repository

- At first, the repository waits (*Pending Invite*) for the response from your partner (who will receive an email asking if he or she accepts or not).

The screenshot shows the GitHub repository settings for managing access. On the left, a sidebar lists options: Options, Manage access (selected), Branches, Webhooks, Notifications, Integrations & services, Deploy keys, Autolink references, Secrets, and Actions. The main area is titled "Who has access". It shows two sections: "PRIVATE REPOSITORY" (with a lock icon) and "DIRECT ACCESS" (with a people icon). Under PRIVATE REPOSITORY, it says "Only those with access to this repository can view it." and has a "Manage" button. Under DIRECT ACCESS, it says "1 has access to this repository. 1 invitation." and shows a list with one item: "belen-rodolfoucha" (with a purple user icon), followed by "Awaiting belen-rodolfoucha's response" and "Pending invite" with a file icon. A green "Invite a collaborator" button is visible. At the bottom right, there is a small circular logo with a stylized design.

Adding your partner to your Git repository

- As soon as the invitation is accepted you will see that his or her status is updated to *Collaborator*.

The screenshot shows two parts of the GitHub repository settings interface.

Who has access:

- PRIVATE REPOSITORY**: Only those with access to this repository can view it.
- DIRECT ACCESS**: 1 has access to this repository. 1 collaborator.

Manage access:

- A green button labeled "Invite a collaborator".
- A search bar with placeholder text "Find a collaborator...".
- A list of users with checkboxes:
 - belen-rodolfoucha (Collaborator)



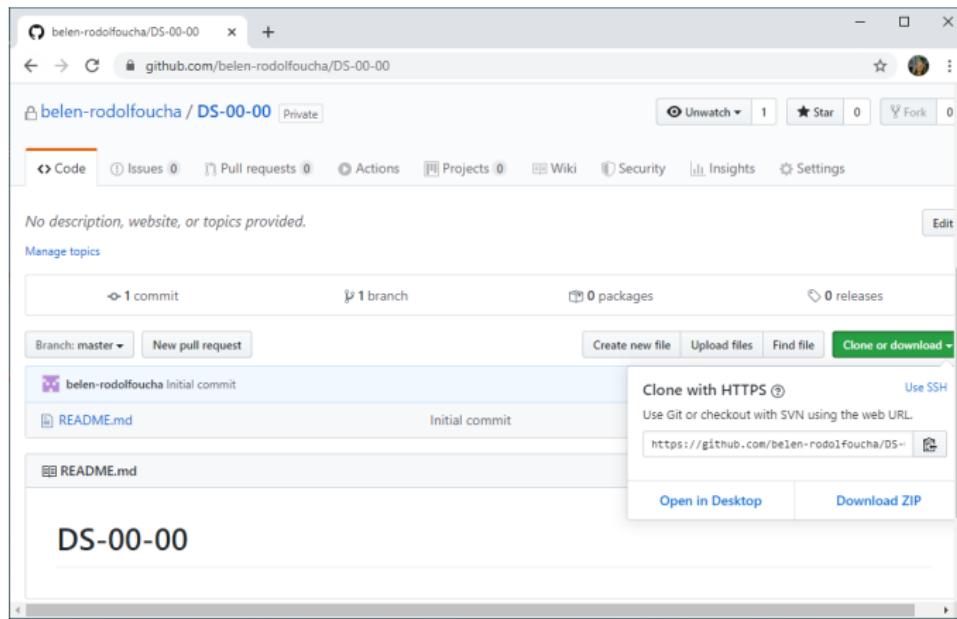
Adding teachers to your Git repository

- You will use the same procedure as to add a partner.
- **Importante:**
 - **It is your responsibility to create the repository and to share it with your practice teacher.**
 - In Moodle you have the list of teachers for each group and their corresponding users in GitHub.
 - After the deadline, teachers will clone, as soon as possible, the repositories that you have shared with us. If this has not been done, it will be understood that the practice has not been submitted.



Git Repository on GitHub

- The last step in GitHub will be to keep the address of the remote repository that we will need to clone it locally. We can get it by clicking on the clone button.



Installing Git

- The next step is to install Git in your computer (if it's not already) from <https://git-scm.com/downloads>.

git --local-branching-on-the-cheap

Search entire site...

About

Documentation

Downloads

GUI Clients

Logos

Community

The entire [Pro Git book](#) written by Scott Chacon and Ben Straub is available to [read online for free](#). Dead tree versions are available on [Amazon.com](#).

Downloads

[Mac OS X](#) [Windows](#)

[Linux/Unix](#)

Older releases are available and the [Git source repository](#) is on GitHub.

GUI Clients

Git comes with built-in GUI tools (`git-gui`, `gitk`), but there are several third-party tools for users looking for a platform-specific experience.

Logos

Various Git logos in PNG (bitmap) and EPS (vector) formats are available for use in online and print projects.

Latest source Release
2.25.0
Release Notes (2020-01-13)
[Download 2.25.0 for Windows](#)

Installing Git

- Install Git with the default options. Make sure that allowing *3rd-party software* is enabled (it should be by default).

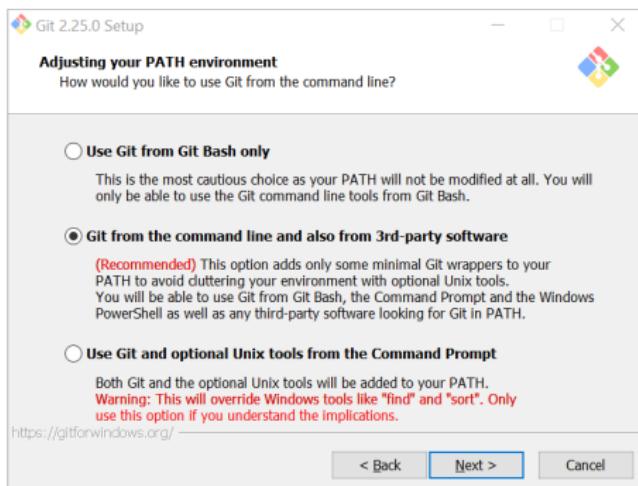


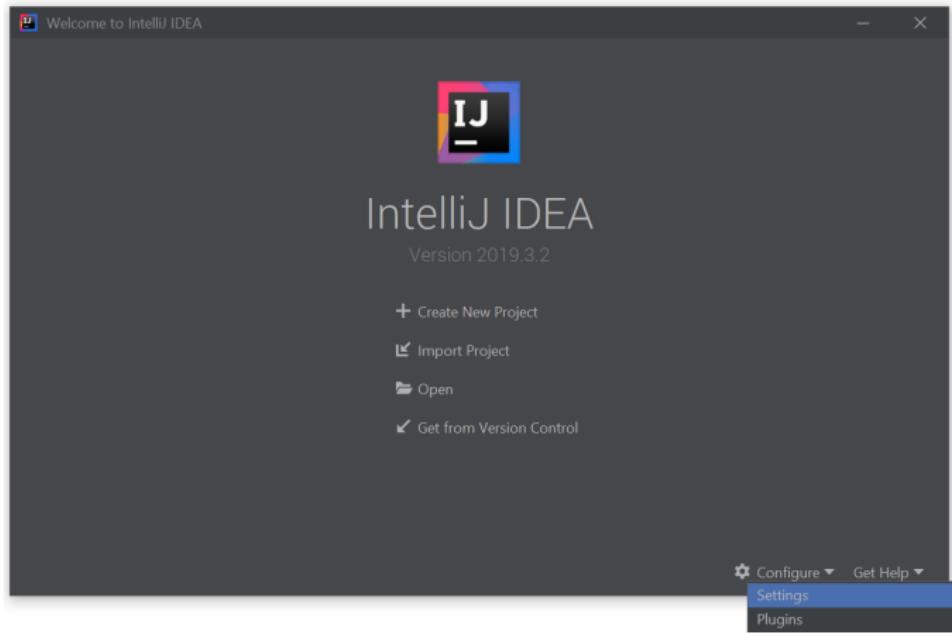
Table Of Contents

- 1 Introduction to Git**
- 2 Creating a Git Repository**
- 3 Git and IntelliJ IDEA**
 - Uploading an IDEA Project into the Repository
 - Centralized workflow with IntelliJ IDEA
 - Other operations with the Repository



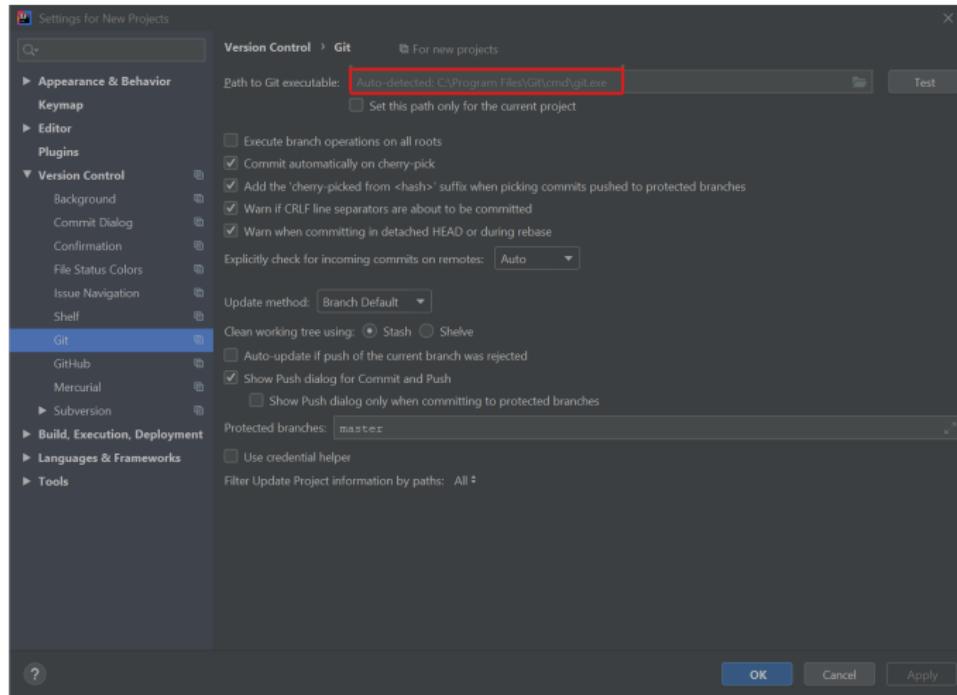
FIC's Git Repository with IntelliJ IDEA

- IntelliJ IDEA will auto-detect the Git installation, but we can check its correctness opening the configuration...



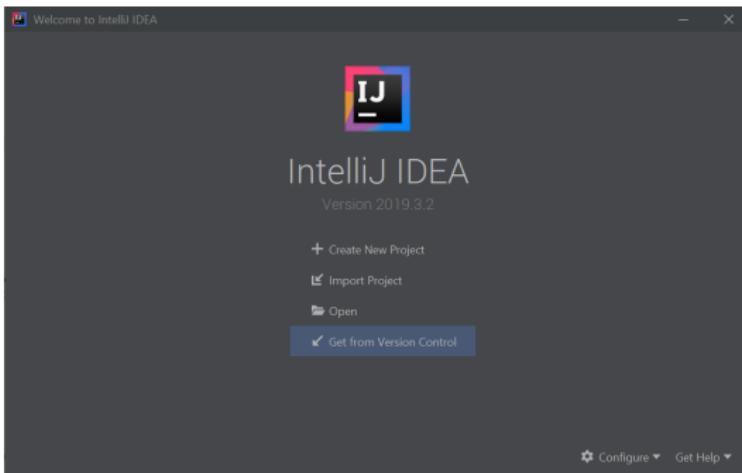
FIC's Git Repository with IntelliJ IDEA

- ...and looking at *Version Control - Git*.



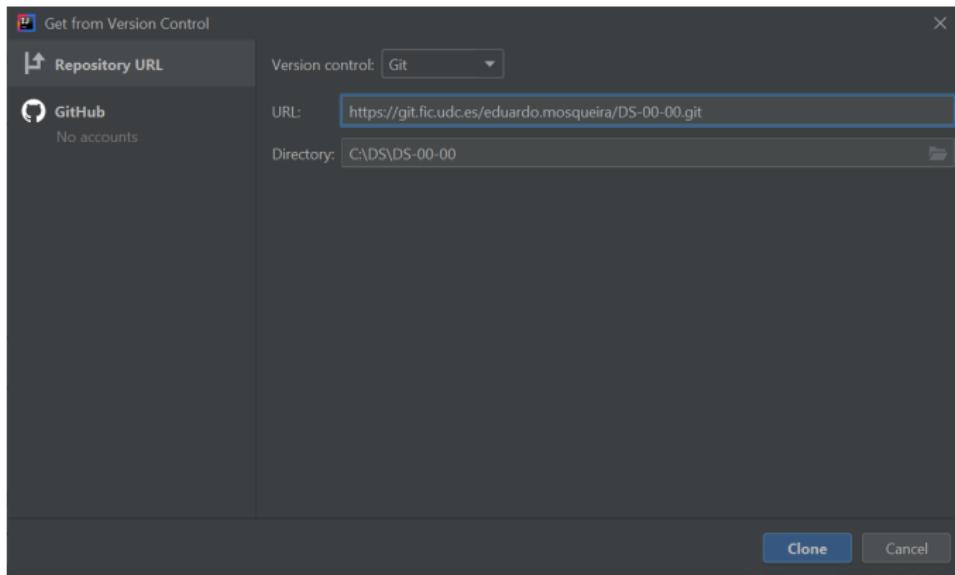
FIC's Git Repository with IntelliJ IDEA

- Once Git is installed, we must copy (clone) our GitLab repository (empty, so far) into a local computer.
- To do that, we select *Get from Version Control* on IntelliJ IDEA.



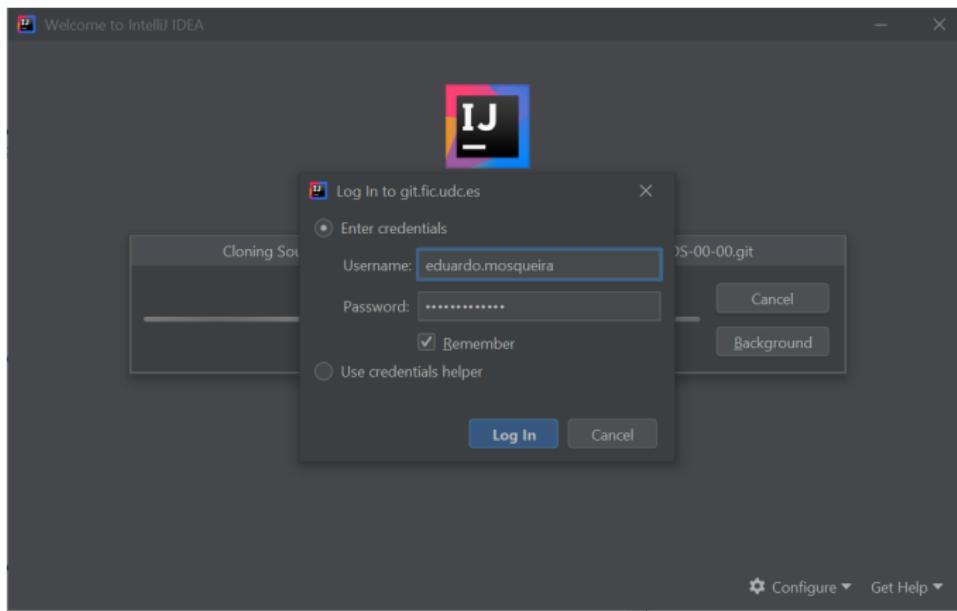
FIC's Git Repository with IntelliJ IDEA

- We paste that URL on IDEA, choose the local folder for the repository, and click *Clone*.



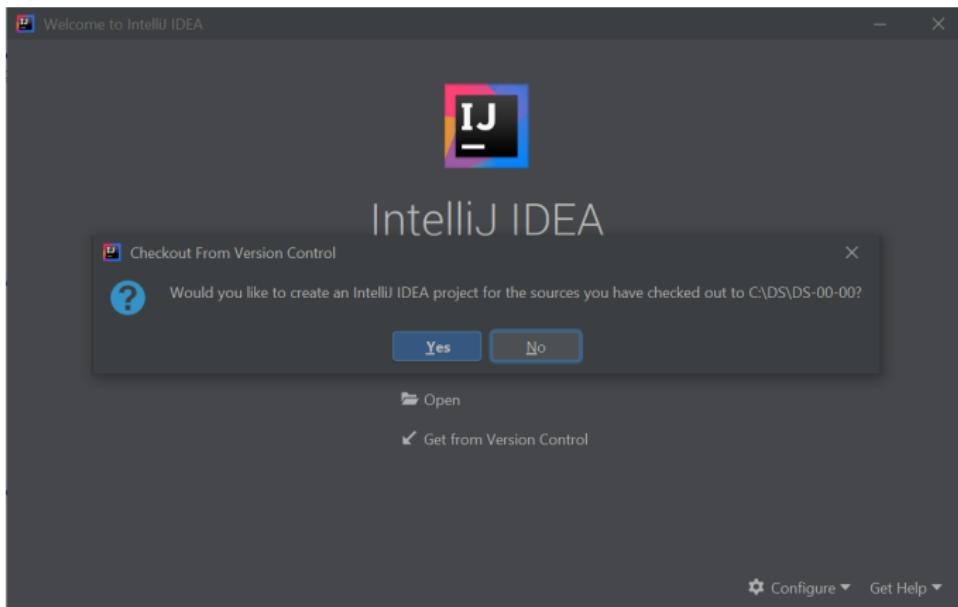
FIC's Git Repository with IntelliJ IDEA

- It will ask for our *Username* and *Password* (simply your UDC credentials) and we tell it to save them.



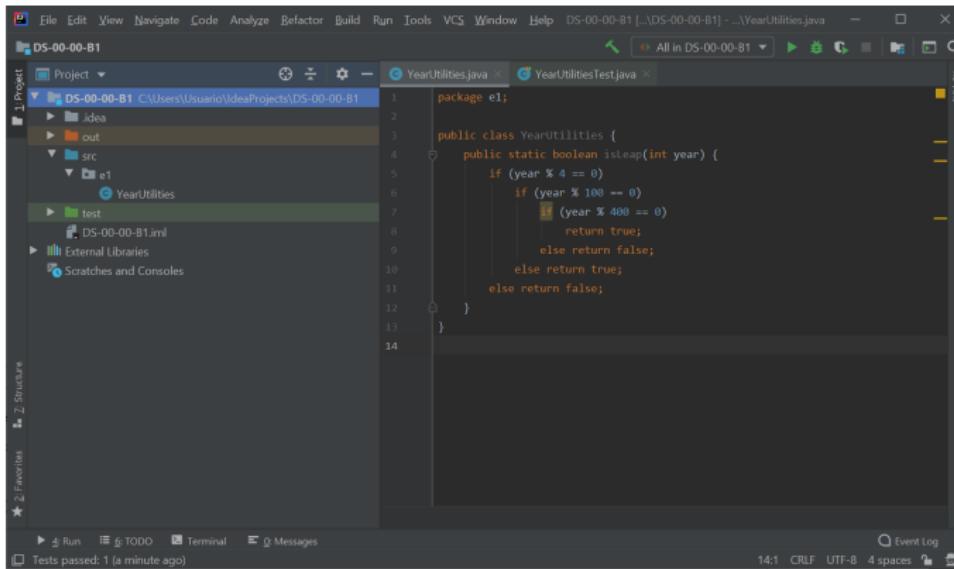
FIC's Git Repository with IntelliJ IDEA

- It will ask whether we want to create an IntelliJ IDEA project in the recently created local copy.
- If the project is already created in other location, it is more convenient to copy it from the browser. Therefore, we choose no.



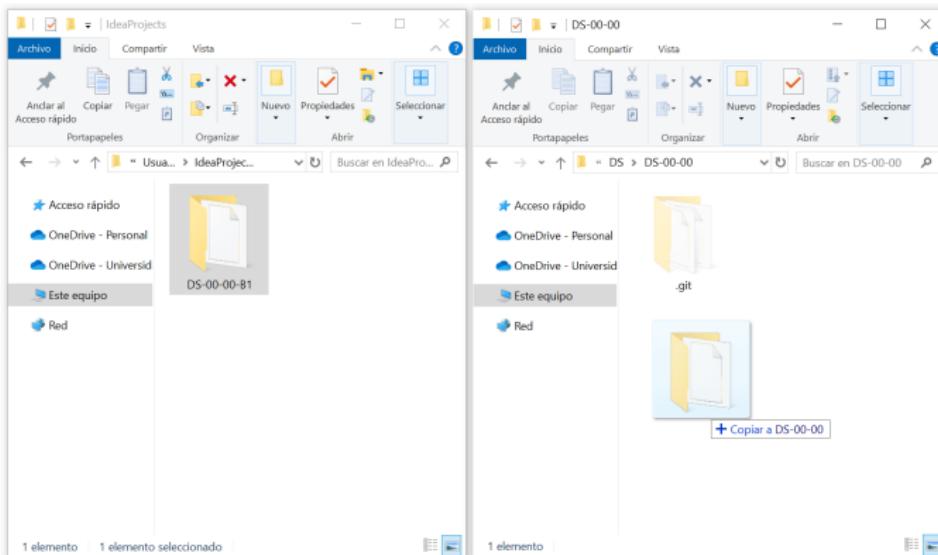
Uploading the project into the Repository with IntelliJ IDEA

- We now go to the created project (e.g., the one for the leap years renamed as DS-00-00-B1)...



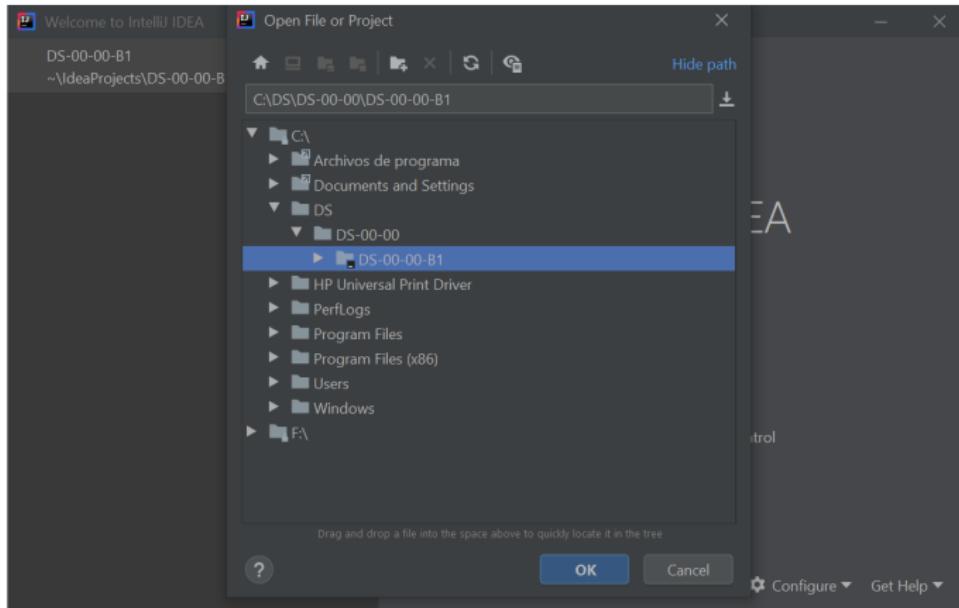
Uploading the project into the Repository with IntelliJ IDEA

- And simply, from the browser, we copy the whole folder into the previously cloned repository.
- We can also see that the target folder has a hidden .git folder that should not be touched.



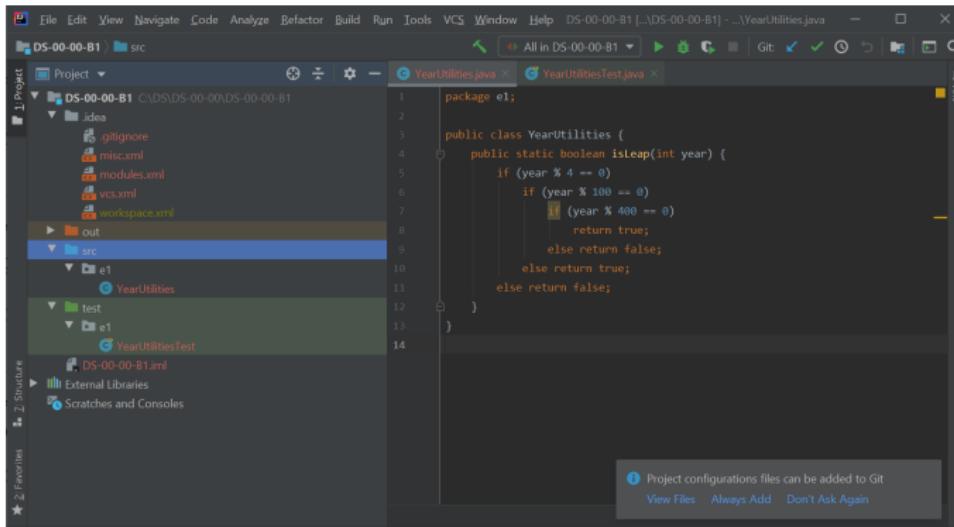
Uploading the project into the Repository with IntelliJ IDEA

- The next step is opening the new project, which is in the Git-controlled location.



Uploading the project into the Repository with IntelliJ IDEA

- The files marked in light red are in the local copy but are not marked for uploading to the repository.
- The file in yellow (`workspace.xml`) will be ignored by Git.
- It will also ask if we want to upload the project's configuration files into the repository.



Uploading the project into the Repository with IntelliJ IDEA

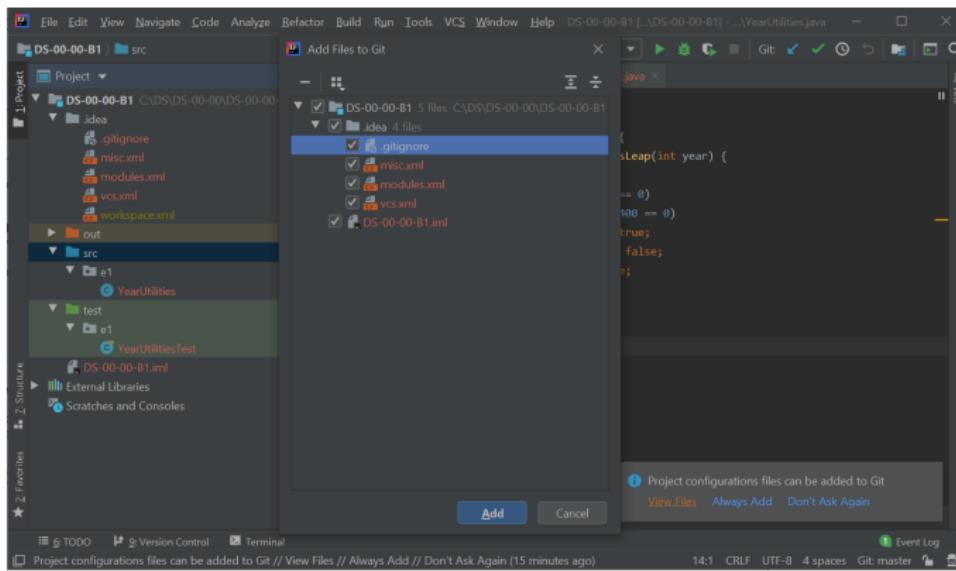
■ Which files will we upload?

- We must upload all the sources (`src` and `test` folders).
- We will also upload the project's configuration files (`.idea` folder) and the `.iml` file. This might be unnecessary if we used different IDEs or *build* systems like Maven or Gradle, but that is not the case in this course.
- The `.idea/workspace.xml` file must not be uploaded as it stores the local configuration of our own IDE (open files, etc.). This has been ignored by adding a `.gitignore` to the folder, which simply contains the name of that configuration file.
- The compiled files (`out` folder) are generated from the sources, so they must not be uploaded to the repository.



Uploading the project into the Repository with IntelliJ IDEA

- If we click on *View Files* on the configuration *pop-up*, we can state that we do want to upload the project's configuration. We click *Add*.



Uploading the project into the Repository with IntelliJ IDEA

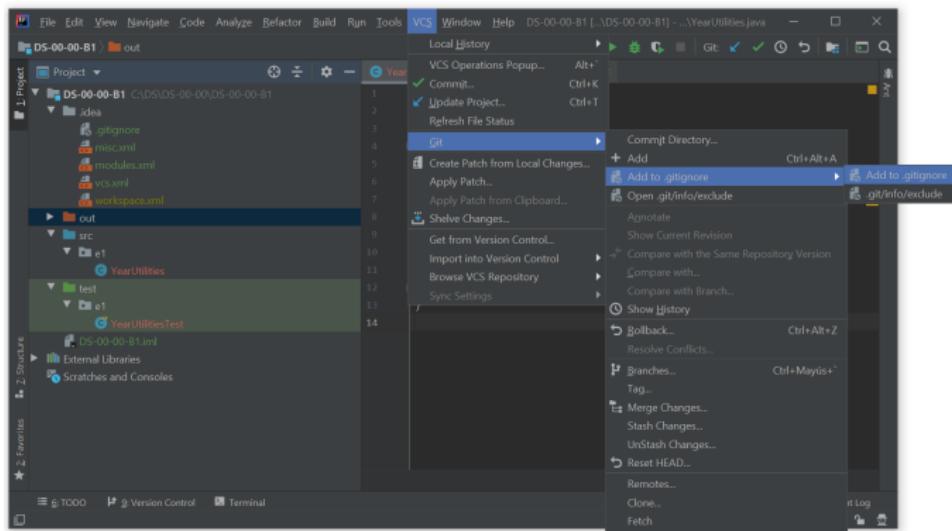
- We now see that all the configuration files are marked in green, which means that they are ready for uploading.

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project View:** Shows the project structure under "DS-00-00-B1". It includes an "idea" folder containing ".gitignore", "misc.xml", "modules.xml", "vcs.xml", and "workspace.xml". It also contains "out", "src", and "e1" folders. "e1" contains "YearUtilities.java" and "YearUtilitiesTest.java". "src" contains "test" and "a1" folders, which contain "YearUtilitiesTest.java" and "DS-00-00-B1.iml" respectively. "External Libraries" and "Scratches and Consoles" are also listed.
- Code Editor:** Displays the content of "YearUtilities.java". The code defines a class "YearUtilities" with a static method "isLeap" that checks if a year is a leap year based on the rules: divisible by 4, divisible by 100, and divisible by 400.
- Status Bar:** At the bottom, it shows "14:1 CRLF UTF-8 4 spaces Git: master" and an "Event Log" icon.

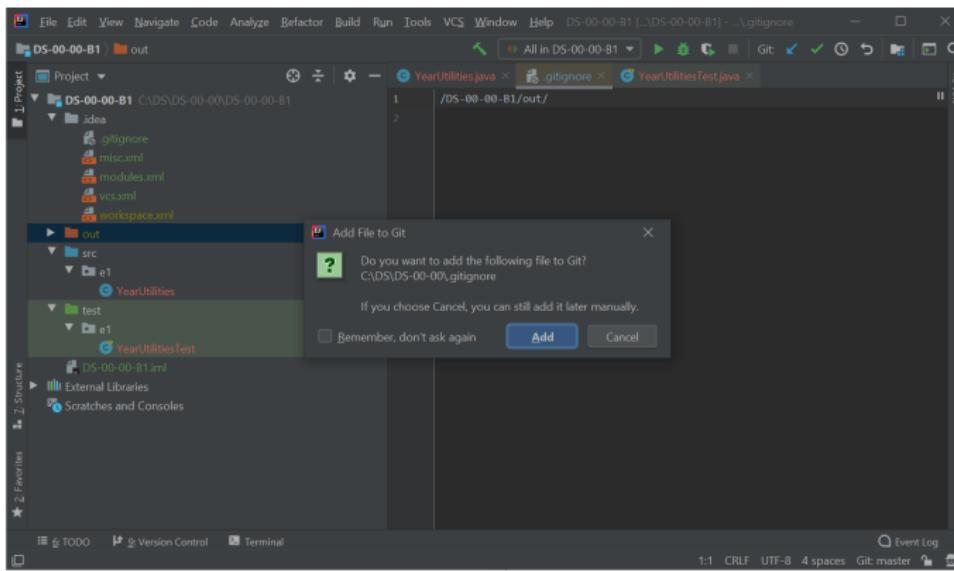
Uploading the project into the Repository with IntelliJ IDEA

- In order to avoid being asked again about the `out` folder, we click on it and from the `VCS - Git` menu we create a `.gitignore` file (it will ask for permission).



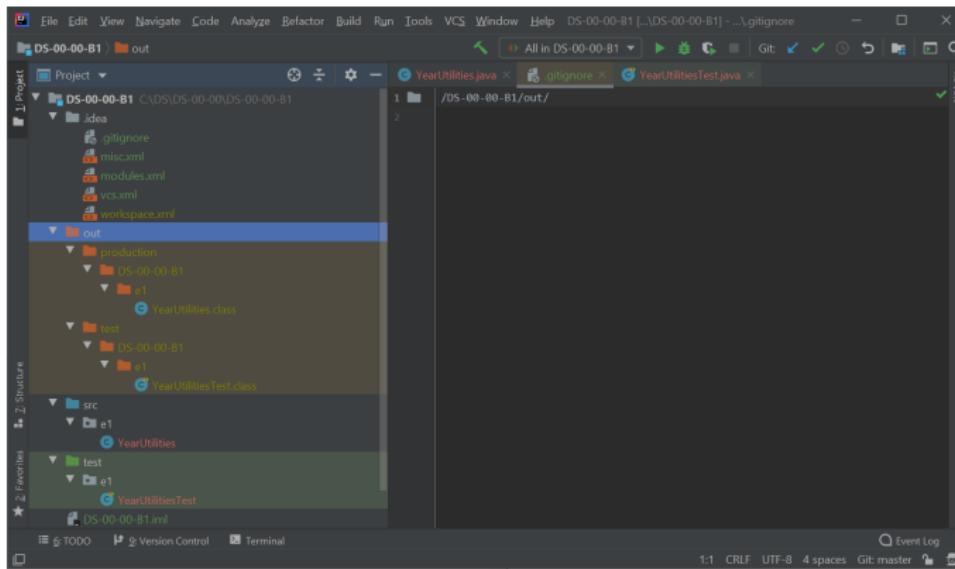
Uploading the project into the Repository with IntelliJ IDEA

- The created `.gitignore` file must be also uploaded to the repository.



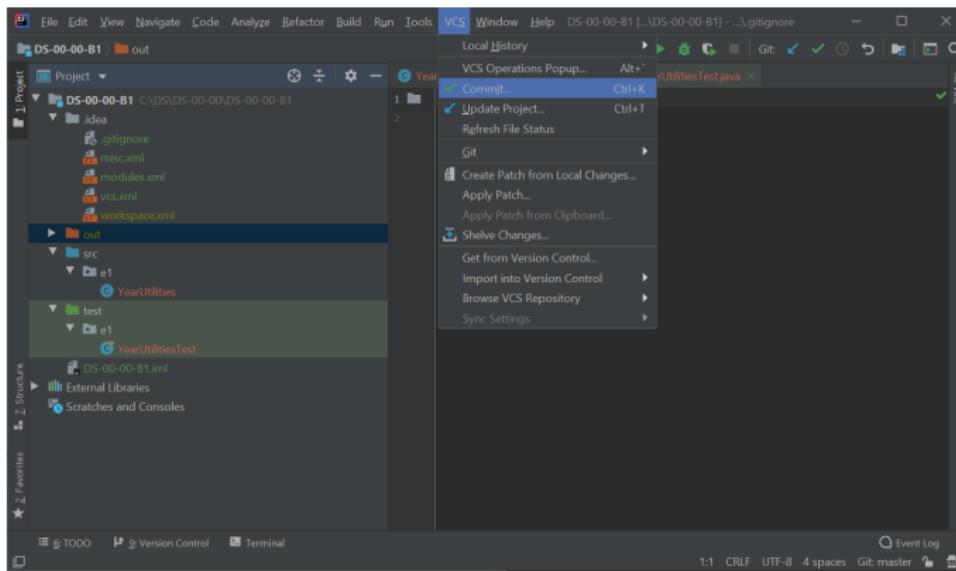
Uploading the project into the Repository with IntelliJ IDEA

- Now the `out` folder and all its contents are marked in yellow, so they will be ignored by Git.



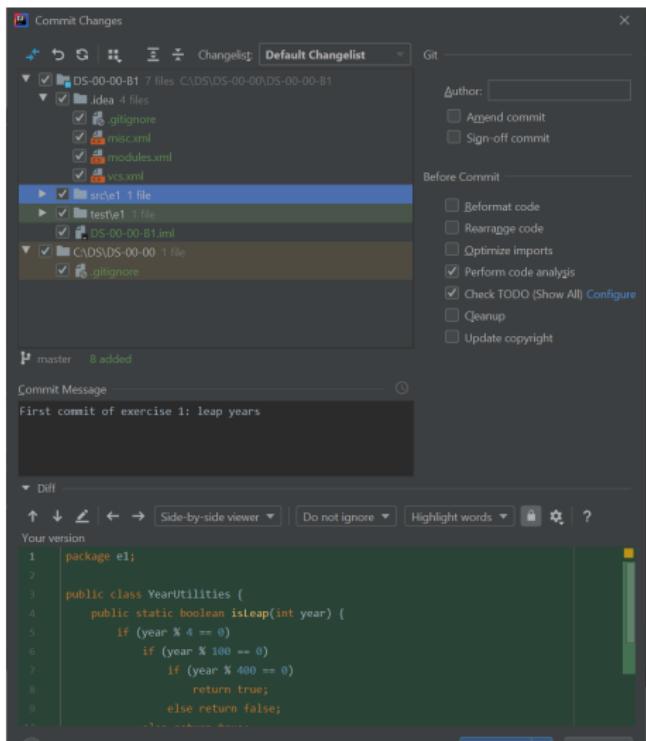
Uploading the project into the Repository with IntelliJ IDEA

- Now we *commit* the changes to upload the versioned files.



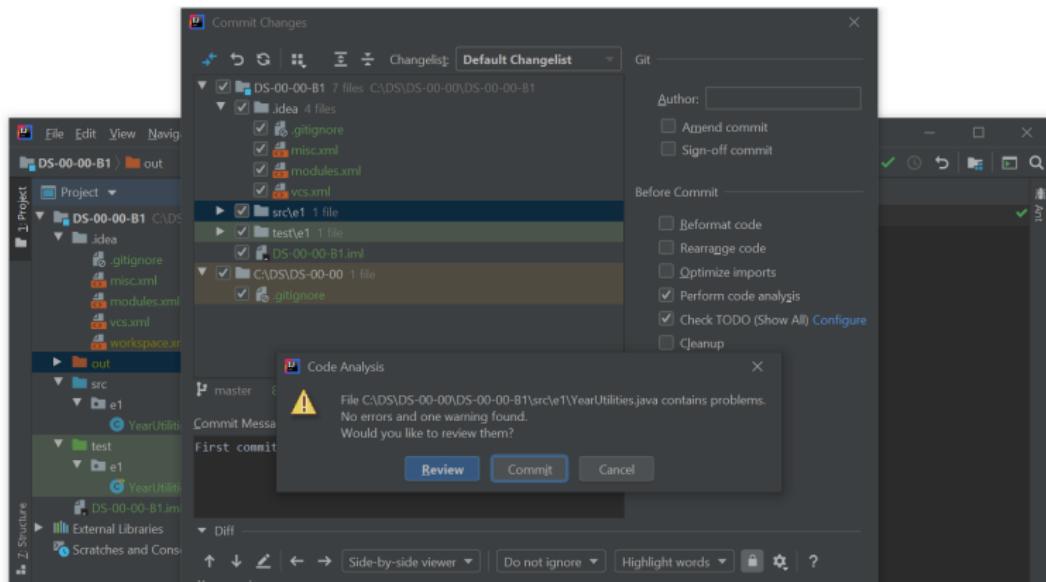
Uploading the project into the Repository with IntelliJ IDEA

- All that is marked in green will be uploaded.
- All that is ignored will not appear.
- The local files that are not versioned (from the `src` and `test` folders) will appear and we must mark them for uploading.
- A small *diff* section will appear, which lets us see the changes being made.
- Messages are mandatory and should be self-explanatory.



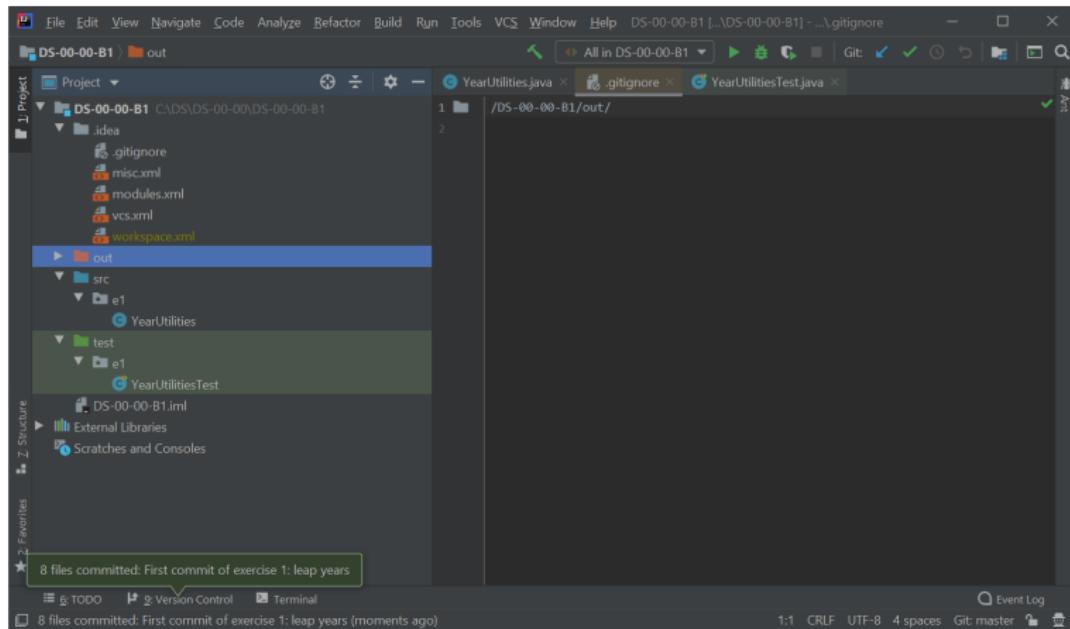
Uploading the project into the Repository with IntelliJ IDEA

- If *Perform code analysis* is enabled, it may tell us to revise the code before we *commit*.
- We can ignore it and *commit* directly, or follow the IDE's recommendations and *commit* later.



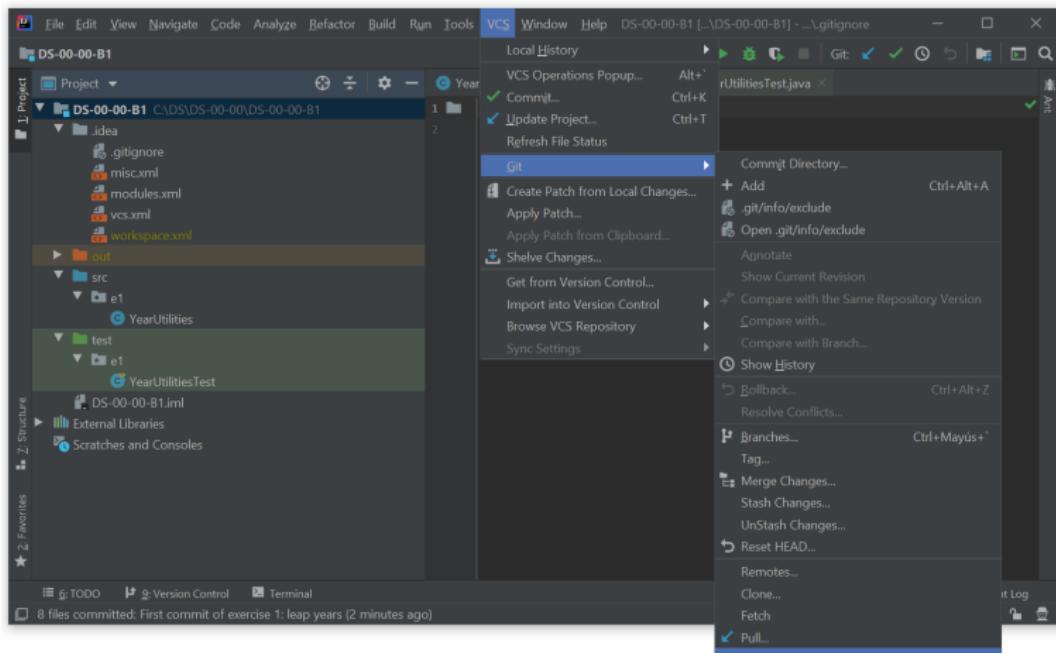
Uploading the project into the Repository with IntelliJ IDEA

- IDEA confirms that changes have been committed. However, they are so far only in our local repository.



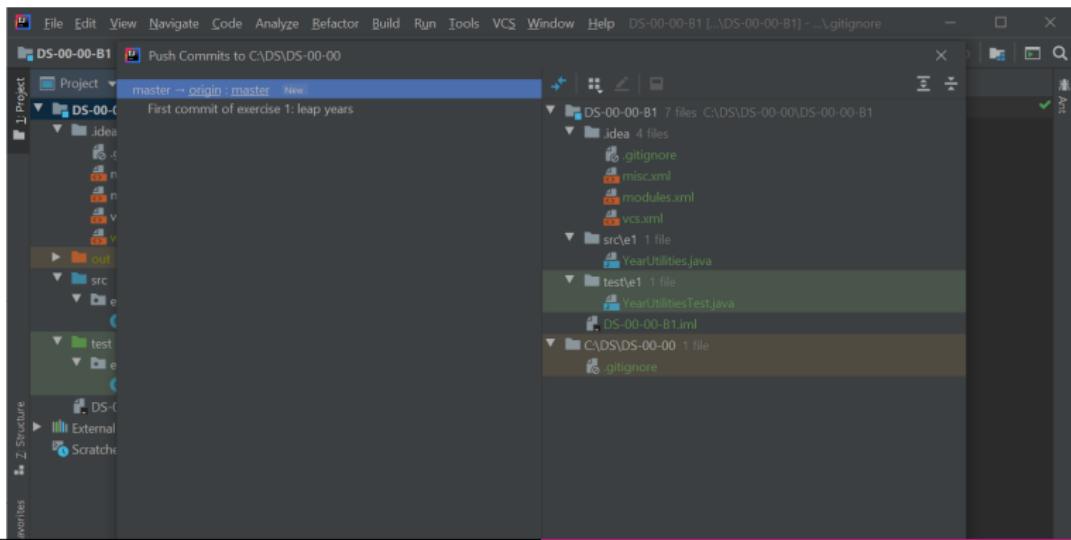
Uploading the project into the Repository with IntelliJ IDEA

- To upload the files into the remote repository, the action is to *push* the changes.



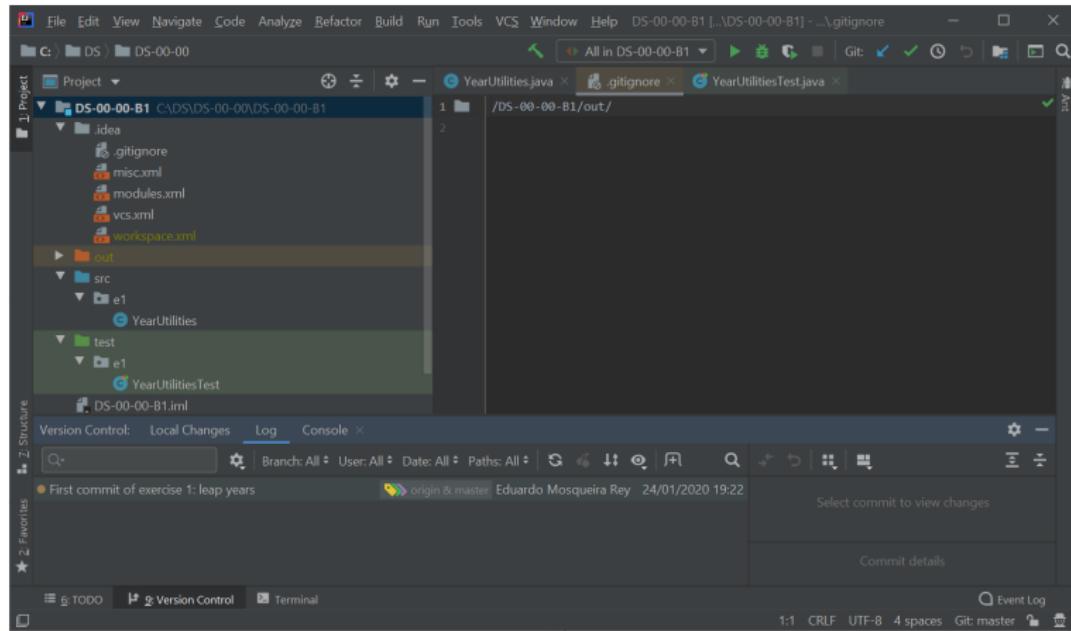
Uploading the project into the Repository with IntelliJ IDEA

- It will ask us which remote branch will be associated with our local branch.
- As we can see, it associates by default master to origin/master, so no further action is required. We simply click Push.



Uploading the project into the Repository with IntelliJ IDEA

- If pushing succeeds, a notification *pop-up* will appear.
- We can see the result in *Version Control - Log*.



Editing code

- We can now make changes to our code.
- The new lines are highlighted in green on the side, the modified lines are in blue and the deleted lines have a triangle.

The screenshot shows the IntelliJ IDEA interface with the following details:

- File Bar:** File, Edit, View, Navigate, Code, Analyze, Refactor, Build, Run, Tools, VCS, Window, Help.
- Project Bar:** DS-00-00-B1, src, e1, YearUtilities.
- Code Editor:** The file YearUtilities.java is open. The code is as follows:

```
1 package e1;
2 public class YearUtilities {
3     /**
4      * Determine if a year is leap
5      * @param year The year to check
6      * @return True if leap, false otherwise
7      */
8     public static boolean isleap(int year) {
9         if (year % 4 == 0)
10             if (year % 100 == 0)
11                 return year % 400 == 0;
12             else return true;
13         else return false;
14     }
15 }
16
```

Annotations on the left margin indicate changes made to the code:

- Line 1: New line (green highlight).
- Line 2: Modified line (blue highlight).
- Line 4: Deleted line (red triangle).
- Line 5: Deleted line (red triangle).
- Line 6: Deleted line (red triangle).
- Line 7: Deleted line (red triangle).
- Line 8: New line (green highlight).
- Line 9: Modified line (blue highlight).
- Line 10: Deleted line (red triangle).
- Line 11: Deleted line (red triangle).
- Line 12: Deleted line (red triangle).
- Line 13: Deleted line (red triangle).
- Line 14: Deleted line (red triangle).
- Line 15: Deleted line (red triangle).

Toolbars and Status Bar: TODO, Version Control, Terminal, Event Log. Status bar shows: 16:1 CRLF UTF-8 4 spaces Git: master.

Editing code

- Clicking on the side we can see the changes in more detail.

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project View:** On the left, the project structure for "DS-00-00-B1" is displayed. It includes a "src" directory containing "e1" and "YearUtilities.java".
- Code Editor:** The main window shows two files: "YearUtilities.java" and "YearUtilitiesTest.java". "YearUtilities.java" contains Java code for determining if a year is leap. A tooltip is shown over the code at line 14, indicating a change in the changelist.
- Change History:** A tooltip at the bottom of the editor shows the current state of the code, including the addition of a condition for years divisible by 400.
- Status Bar:** At the bottom, the status bar shows "16:1 CRLF UTF-8 4 spaces Git: master".

Editing code

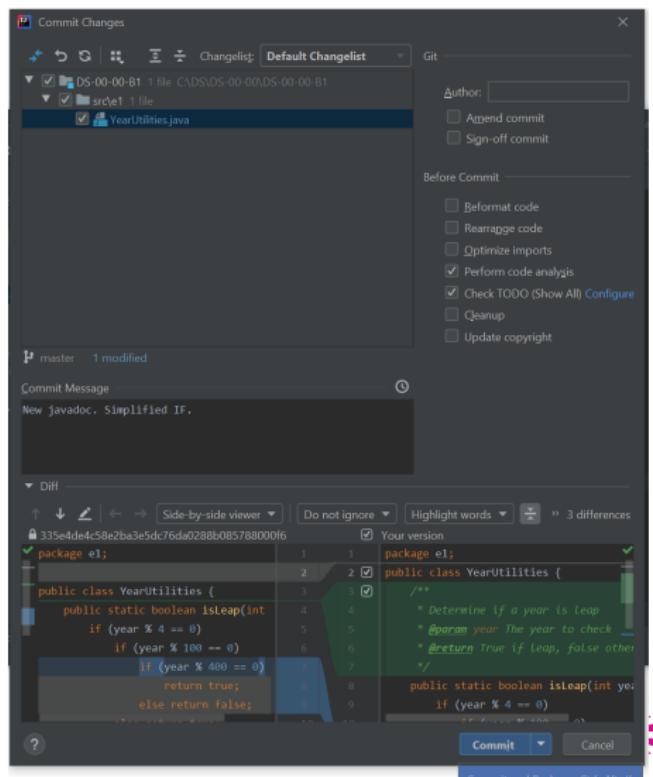
- We can see the changes for the entire file selecting *diff* from the *Version Control* window.

The screenshot shows the IntelliJ IDEA interface with the following details:

- File Menu:** File, Edit, View, Navigate, Code, Analyze, Refactor, Build, Run, Tools, VCS, Window, Help.
- Project Bar:** DS-00-00-B1, src, e1, YearUtilities.java.
- Toolbars:** Side-by-side viewer, Do not ignore, Highlight words.
- Code Editor:** Shows the `YearUtilities.java` file with a diff view. The left column shows the original code, and the right column shows the modified code with changes highlighted in green. The code implements a leap year determination algorithm.
- Status Bar:** 2:1 CRLF UTF-8 4 spaces Git: master.
- Bottom Bar:** Compare files or revisions, Show Diff Ctrl+D, Version Control, Terminal.

Sending changes

- Performing a new *commit* will submit the changes to the repository.
- To make it faster, we can *commit* and *push* at the same time.



Sending changes

- If everything went as expected, two *pop-ups* will appear, one for the *commit* and another for the *push*.

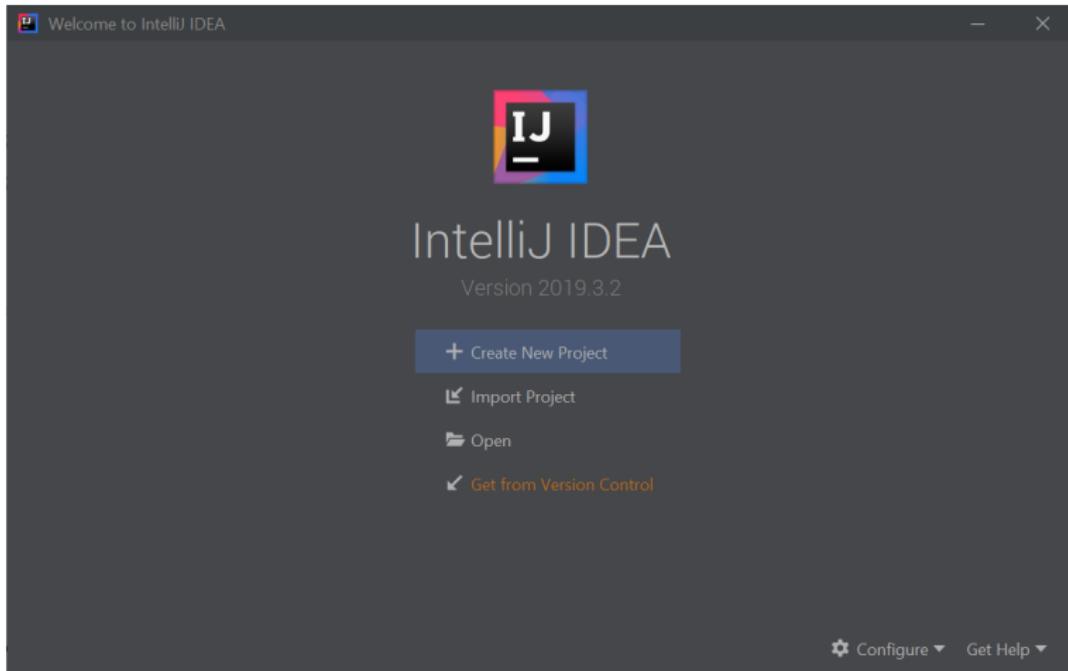
The screenshot shows the IntelliJ IDEA interface with the following details:

- File Structure:** The left sidebar shows a project named "DS-00-00-B1" containing "src" and "out" directories, with "src" expanded to show "e1" and "YearUtilities".
- Code Editor:** The main window displays "YearUtilities.java" with the following code:

```
1 package e1;
2 public class YearUtilities {
3     /**
4      * Determine if a year is Leap
5      * @param year The year to check
6      * @return True if Leap, false otherwise
7      */
8     public static boolean isLeap(int year) {
9         if (year % 4 == 0)
10             if (year % 100 == 0)
11                 return year % 400 == 0;
12             else return true;
13         else return false;
14     }
15 }
```
- Toolbars and Status Bar:** The bottom status bar shows "1 file committed: New javadoc. Simplified IF." and "Pushed 1 commit to origin/master".
- Bottom Right:** A decorative circular logo with a pink/purple gradient.

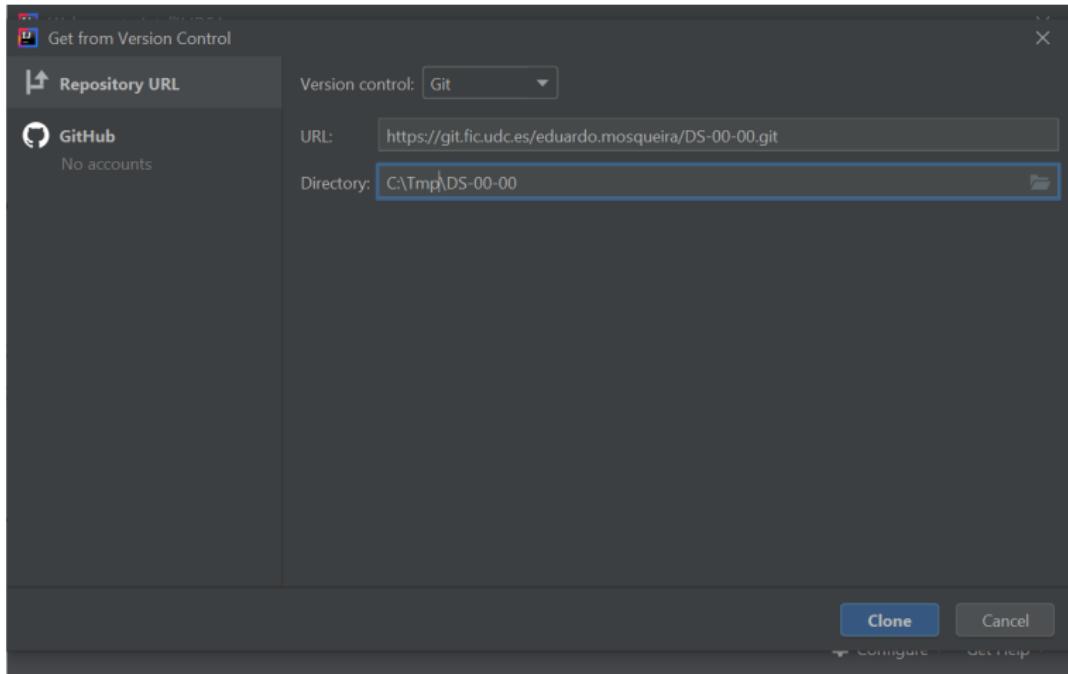
Downloading a fresh copy of the project

- In order to download a fresh copy of the project into our computer, we can select *Get from Version Control* when launching IntelliJ IDEA.



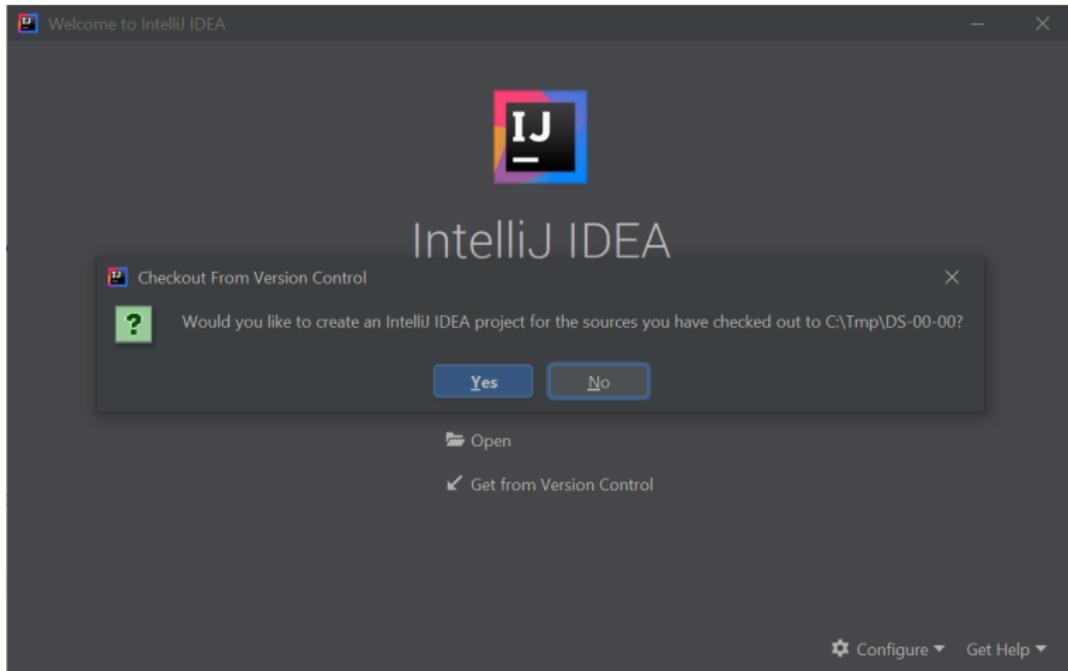
Downloading a fresh copy of the project

- We choose Git, enter the remote repository's location and the local folder where it will be downloaded. Then we click *Clone*.



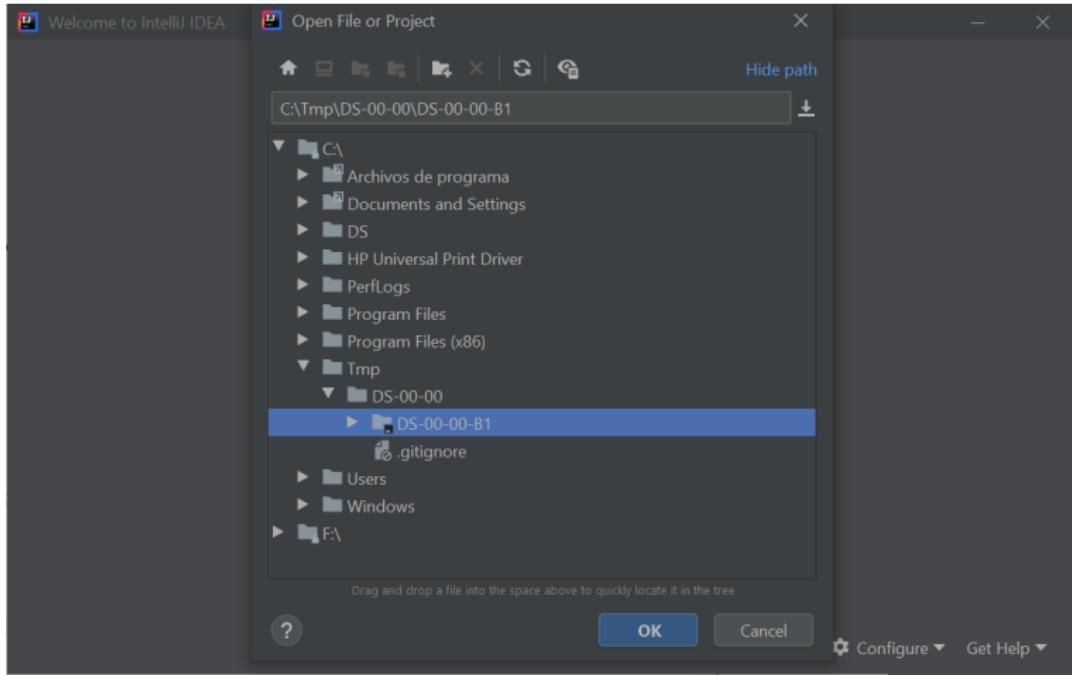
Downloading a fresh copy of the project

- It is not necessary to state whether you want to create a new IntelliJ IDEA project for those sources, since we have already created one. So we choose no.



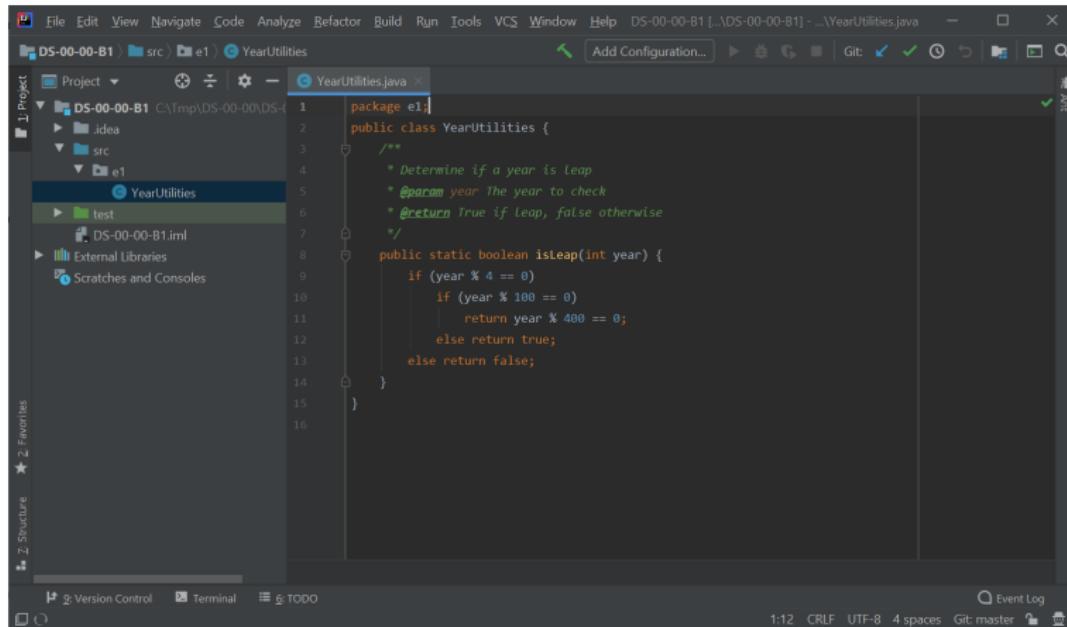
Downloading a fresh copy of the project

- The next step is to open the recently downloaded project.



Downloading a fresh copy of the project

- We finally obtain a new copy of our project in another computer.



The screenshot shows the IntelliJ IDEA interface with the following details:

- Project View:** Shows the project structure under "DS-00-00-B1". The "src" folder contains "e1" which has "YearUtilities".
- Editor:** The "YearUtilities.java" file is open. The code implements a static method `isLeap` that checks if a year is leap. It handles the standard rule (year % 4 == 0) and the exception rule (year % 100 == 0 and year % 400 == 0).

```
1 package e1;
2
3 /**
4 * Determine if a year is leap
5 * @param year The year to check
6 * @return True if Leap, false otherwise
7 */
8 public static boolean isLeap(int year) {
9     if (year % 4 == 0)
10         if (year % 100 == 0)
11             return year % 400 == 0;
12         else return true;
13     else return false;
14 }
15 }
```

- Status Bar:** Shows the current time (1:12), file encoding (CRLF, UTF-8), number of spaces (4), and the Git status (master).

Resolving conflicts

- From this new copy we add a new conversion method, from Gregorian years to Muslim years...

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Bar:** DS-00-00-B1, src, e1, YearUtilities.
- Toolbars:** File, Edit, View, Navigate, Code, Analyze, Refactor, Build, Run, Tools, VCS, Window, Help.
- Status Bar:** DS-00-00-B1 [...\DS-00-00-B1] - YearUtilities.java, 16:52 CRLF UTF-8 4 spaces, Git: master.
- Editor:** The code for `YearUtilities.java` is displayed. The class contains two static methods: `isLeap` and `GregorianToMuslim`. The `isLeap` method checks if a year is a leap year based on the rules: divisible by 4, not divisible by 100, or divisible by 400. The `GregorianToMuslim` method returns the year minus 622.

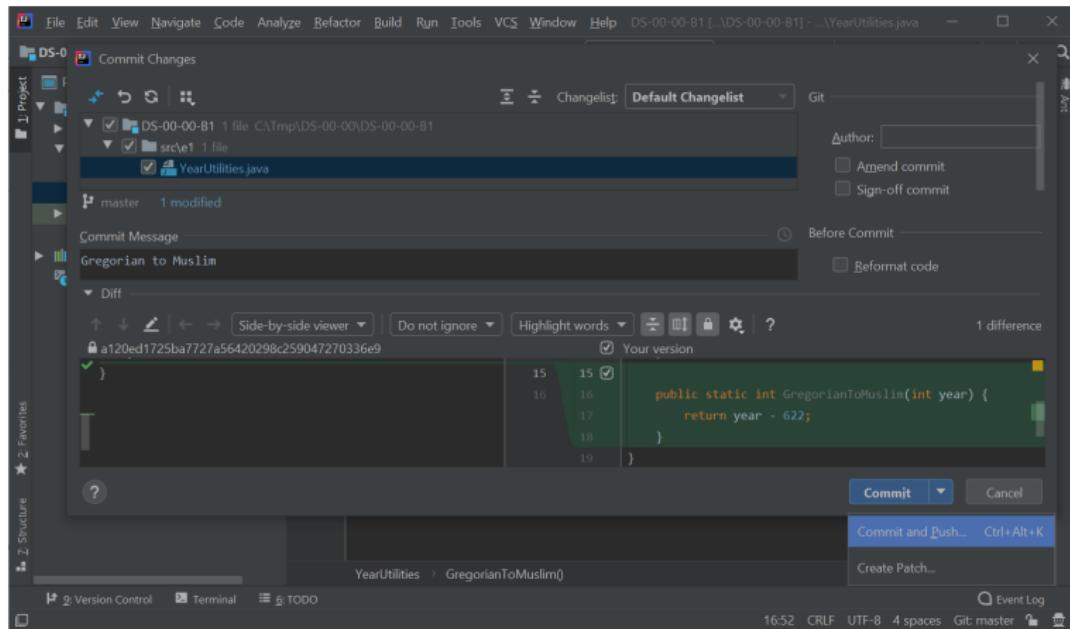
```
package e1;
public class YearUtilities {
    /**
     * Determine if a year is Leap
     * @param year The year to check
     * @return True if Leap, false otherwise
     */
    public static boolean isLeap(int year) {
        if (year % 4 == 0)
            if (year % 100 == 0)
                return year % 400 == 0;
            else return true;
        else return false;
    }
    public static int GregorianToMuslim(int year) {
        return year - 622;
    }
}
```

- Sidebar:** Project (DS-00-00-B1), External Libraries, Scratches and Consoles.
- Bottom Navigation:** Version Control, Terminal, TODO.
- Event Log:** Event Log icon.



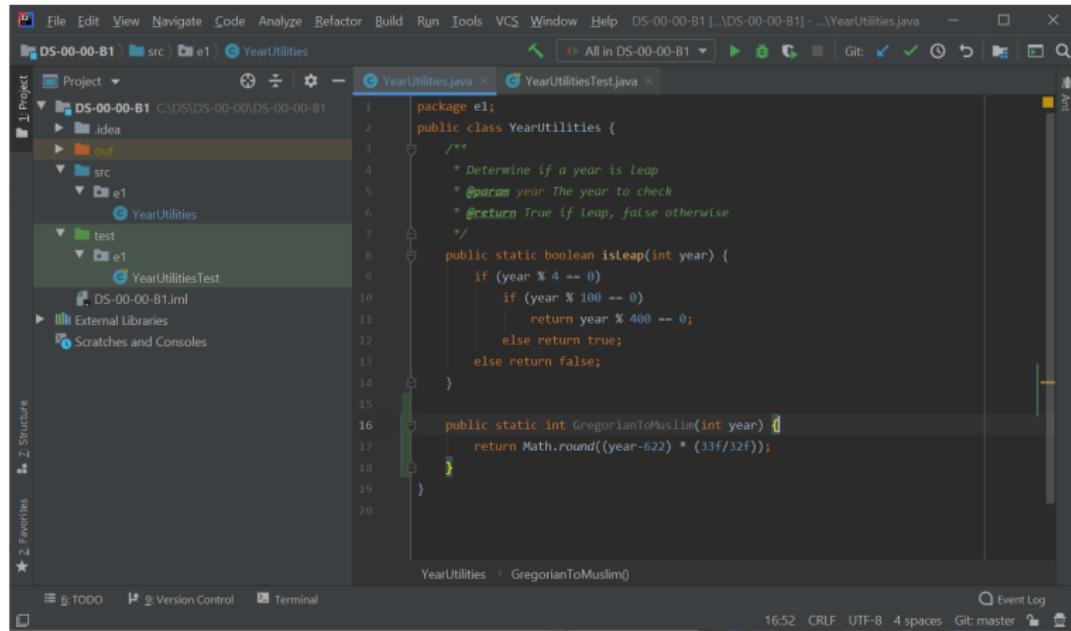
Resolving conflicts

- ...and *commit* and *push*.



Resolving conflicts

- Due to a coordination error, our partner had simultaneously created a different, more succinct version of the GregorianToMuslim method.



The screenshot shows the IntelliJ IDEA interface with the following details:

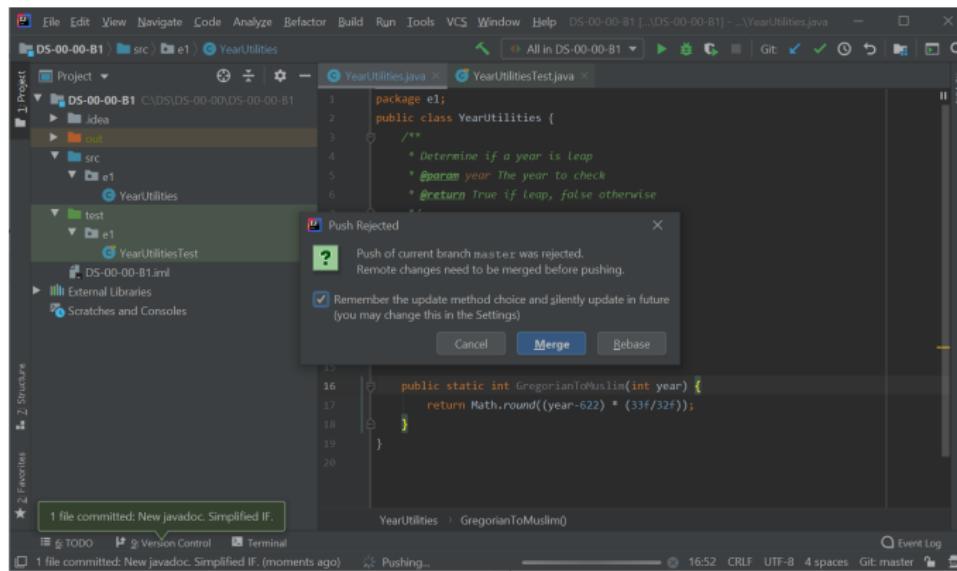
- File Menu:** File, Edit, View, Navigate, Code, Analyze, Refactor, Build, Run, Tools, VCS, Window, Help.
- Project Bar:** DS-00-00-B1, src, e1, YearUtilities.
- Toolbars:** All in DS-00-00-B1, Git.
- Left Sidebar:** Project (DS-00-00-B1), External Libraries, Scratches and Consoles.
- Right Sidebar:** Structure, Favorites.
- Code Editor:** YearUtilities.java (selected), YearUtilitiesTest.java. The code editor shows a conflict between two versions of the GregorianToMuslim method. The first version is the original, and the second is a newer, more concise version. The newer version has been merged into the original, with the original's logic removed.

```
1 package e1;
2 public class YearUtilities {
3     /**
4      * Determine if a year is leap
5      * @param year The year to check
6      * @return True if leap, false otherwise
7      */
8     public static boolean isLeap(int year) {
9         if (year % 4 == 0)
10             if (year % 100 == 0)
11                 return year % 400 == 0;
12             else return true;
13         else return false;
14     }
15
16     public static int GregorianToMuslim(int year) {
17         return Math.round((year-622) * (33f/32f));
18     }
19 }
```

- Bottom Status Bar:** TODO, Version Control, Terminal, Event Log, 16:52, CRLF, UTF-8, 4 spaces, Git: master.

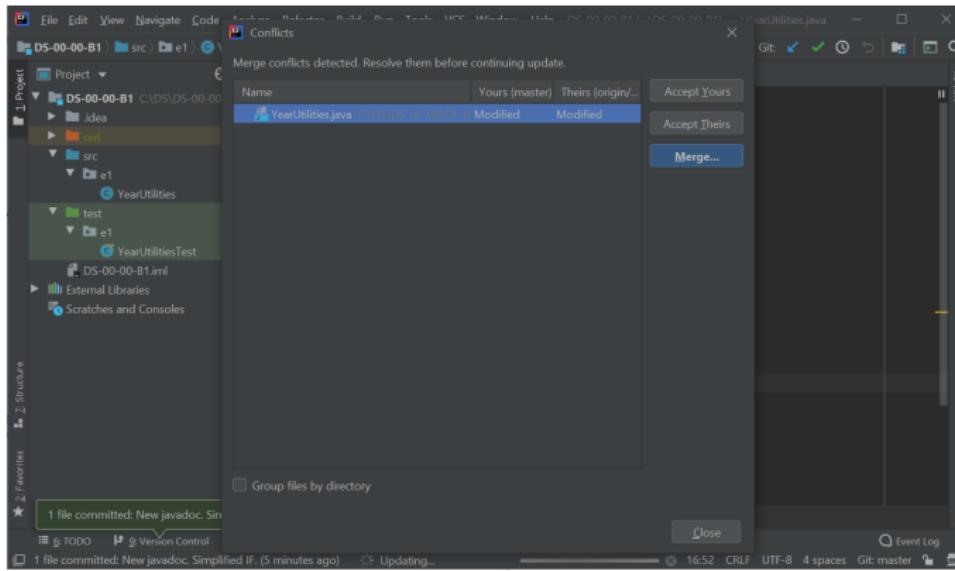
Resolving conflicts

- If we try to *commit & push*, we'll see that the first command succeeds (it is a local operation) but the second one fails because the repository has another version which is newer than the current one.
- It tells us to **Merge** the contents of the repository with the local copy. **Do not choose Rebase**, always *Merge*.



Resolving conflicts

- When we try to merge *merge*, a conflict is created, since we have changed the same lines of code in two different local copies.
- A conflict resolution window appears, in which we can resolve it easily by choosing *Accept Yours* or *Accept Theirs...*



Resolving conflicts

- ...but if we are unsure, we can click on *Merge* to see which lines are in conflict.

The screenshot shows the 'Merge Revisions' dialog for a Java file named YearUtilities.java. The dialog displays three panes: 'Your version' (left), 'Result' (center), and 'Changes from server (revision 2d4901b044360069692035b38b9a07f6d...)'. The 'Result' pane shows the merged code with line numbers. A conflict is indicated at line 15, where both sides have changes: 'Accept Left' and 'Accept Right' buttons are visible at the bottom left. The 'Apply' and 'Cancel' buttons are at the bottom right.

```
package e1;
public class YearUtilities {
    /**
     * Determine if a year is leap
     * @param year The year to check
     * @return True if Leap, false otherwise
     */
    public static boolean isLeap(int year) {
        if (year % 4 == 0)
            if (year % 100 == 0)
                return year % 400 == 0;
            else return true;
        else return false;
    }
    public static int GregorianToMuslim(int year) {
        return Math.round((year-622) * (334/32f));
    }
}
```

```
package e1;
public class YearUtilities {
    /**
     * Determine if a year is leap
     * @param year The year to check
     * @return True if Leap, false otherwise
     */
    public static boolean isLeap(int year) {
        if (year % 4 == 0)
            if (year % 100 == 0)
                return year % 400 == 0;
            else return true;
        else return false;
    }
    public static int GregorianToMuslim(int year) {
        return year - 622;
    }
}
```

Resolving conflicts

- The Merge windows shows that our version (on the left) is more complete, so we select *Accept Left*.
- If we click on the >> or << arrows, we can accept changes from either side (assuming there are multiple conflicts).

Merge Revisions for C:\DS\DS-00-00\DS-00-00-B1\src\et1\YearUtilities.java

Your version

```
✓ package et1;
public class YearUtilities {
    /**
     * Determine if a year is leap
     * @param year The year to check
     * @return True if Leap, false otherwise
     */
    public static boolean isLeap(int year) {
        if (year % 4 == 0)
            if (year % 100 == 0)
                return year % 400 == 0;
            else return true;
        else return false;
    }

    public static int GregorianToMuslim(int year) {
        return Math.round((year-622) * (33f/32f));
    }
}
```

Result

```
package et1;
public class YearUtilities {
    /**
     * Determine if a year is leap
     * @param year The year to check
     * @return True if Leap, false otherwise
     */
    public static boolean isLeap(int year) {
        if (year % 4 == 0)
            if (year % 100 == 0)
                return year % 400 == 0;
            else return true;
        else return false;
    }

    public static int GregorianToMuslim(int year) {
        return year - 622;
    }
}
```

Changes from server (revision 2d4901b044360069692035b38b9a7fd...)

No changes. 1 conflict

> Apply non-conflicting changes >> Left << All << Right ⌂ Do not ignore ⌂ Highlight words ⌂ ? ⌂

Accept Left Accept Right Apply Cancel

Resolving conflicts

- When it finishes, it reminds us that the *Push* failed. So, once the conflicts have been resolved, we must try it again...

The screenshot shows the IntelliJ IDEA interface with a Java project named "DS-00-00-B1". The "src" directory contains "e1" and "YearUtilities" packages. "YearUtilities" contains "YearUtilities.java" and "YearUtilitiesTest.java". "YearUtilitiesTest.java" is currently selected. The code in "YearUtilities.java" is:

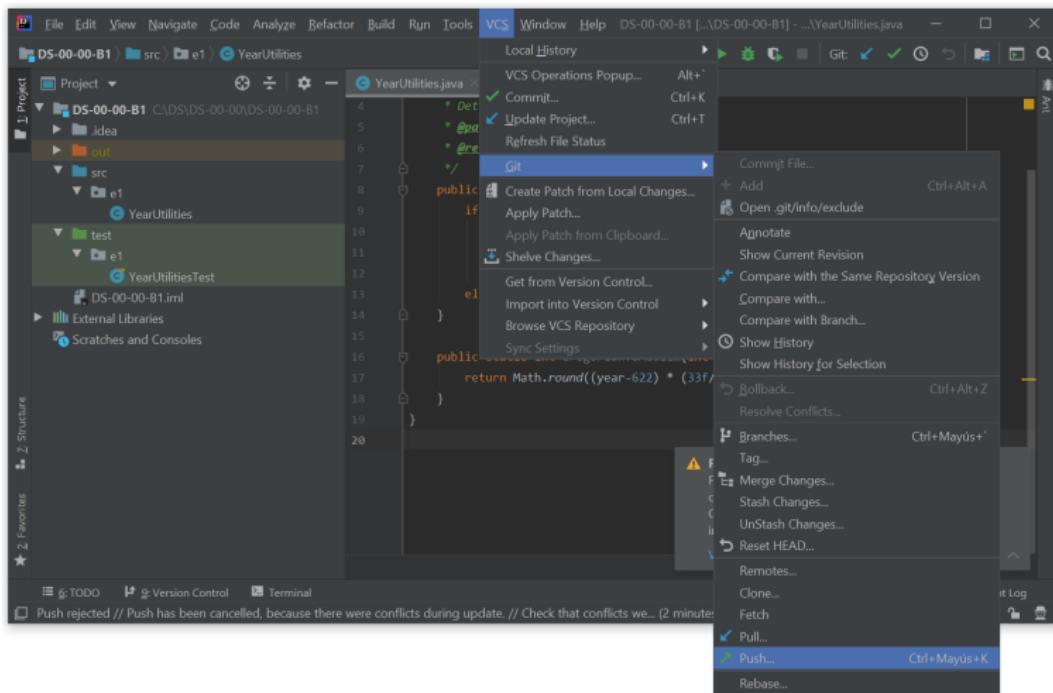
```
* Determine if a year is leap
 * @param year The year to check
 * @return True if leap, false otherwise
 */
public static boolean isLeap(int year) {
    if (year % 4 == 0)
        if (year % 100 == 0)
            return year % 400 == 0;
        else return true;
    else return false;
}

public static int GregorianToMuslim(int year) {
    return Math.round((year-622) * (33f/32f));
}
```

A tooltip at the bottom right says "Push rejected" with the message: "Push has been cancelled, because there were conflicts during update. Check that conflicts were resolved correctly, and invoke push again." Below the tooltip, there's a link "View received commit".

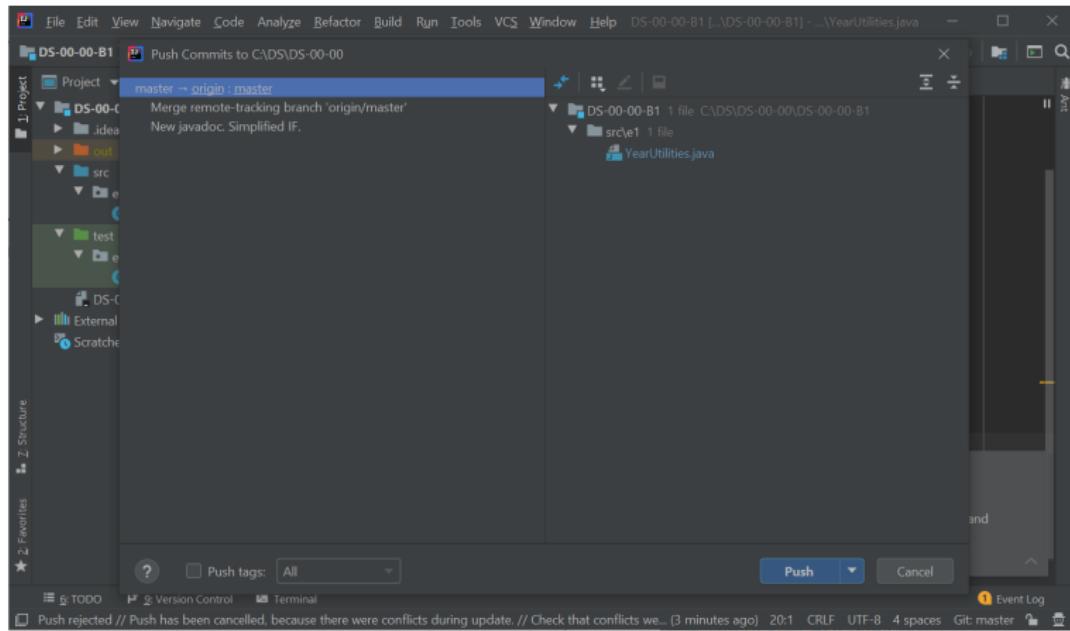
Resolving conflicts

- ...So we *Push* again...



Resolving conflicts

- ...and see that the *commit* message is *Merge remote-tracking branch...*



Resolving conflicts

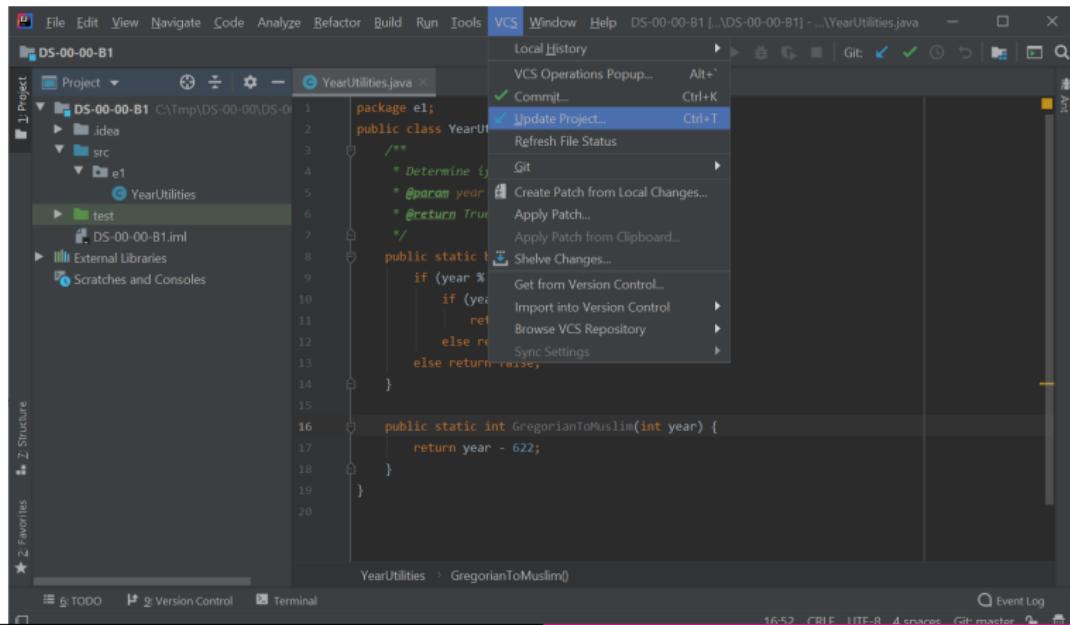
- When it's done, a *pop-up* appears, showing that two *commits* have been made – the original that we intended and the result of the *merge*.

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Structure:** The left sidebar shows the project structure for "DS-00-00-B1". It includes a "src" folder containing "e1" and "YearUtilities.java", and a "test" folder containing "YearUtilitiesTest.java".
- Code Editor:** The main window displays the content of "YearUtilities.java". The code defines two static methods: `isLeap` and `GregorianToMuslim`. A conflict is visible in the `isLeap` method where the code has been merged from two different sources.
- Toolbars and Status:** The top bar shows various menu options like File, Edit, View, Navigate, Code, Analyze, Refactor, Build, Run, Tools, VCS, Window, Help. The status bar at the bottom shows "Pushed 2 commits to origin/master" and other git-related information.
- Bottom Right Corner:** A decorative logo consisting of overlapping colored bands.

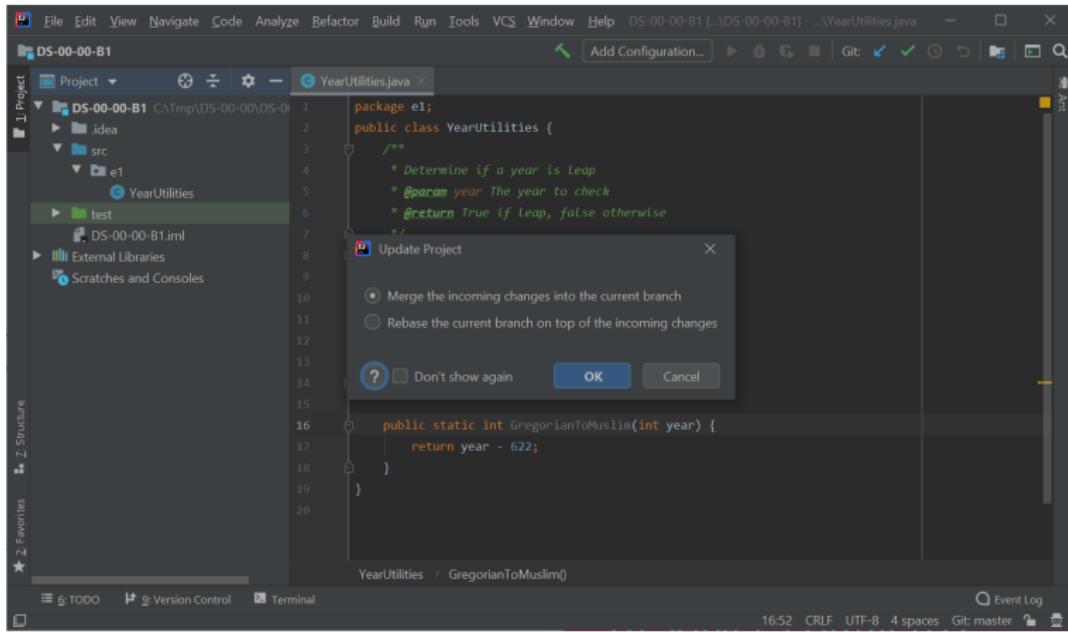
Resolving conflicts

- Now, the other member of the team, the one who had uploaded a simplified version of GregorianToMuslim is not up-to-date with the final version...
- ...and has to *Update Project* to download the remote changes.



Resolving conflicts

- Again, it asks whether we want to *Merge* or *Rebase*. We choose *Merge*.



Resolving conflicts

- In this case there are no conflicts, and the remote and local versions are correctly synchronized.

The screenshot shows the IntelliJ IDEA interface with the following details:

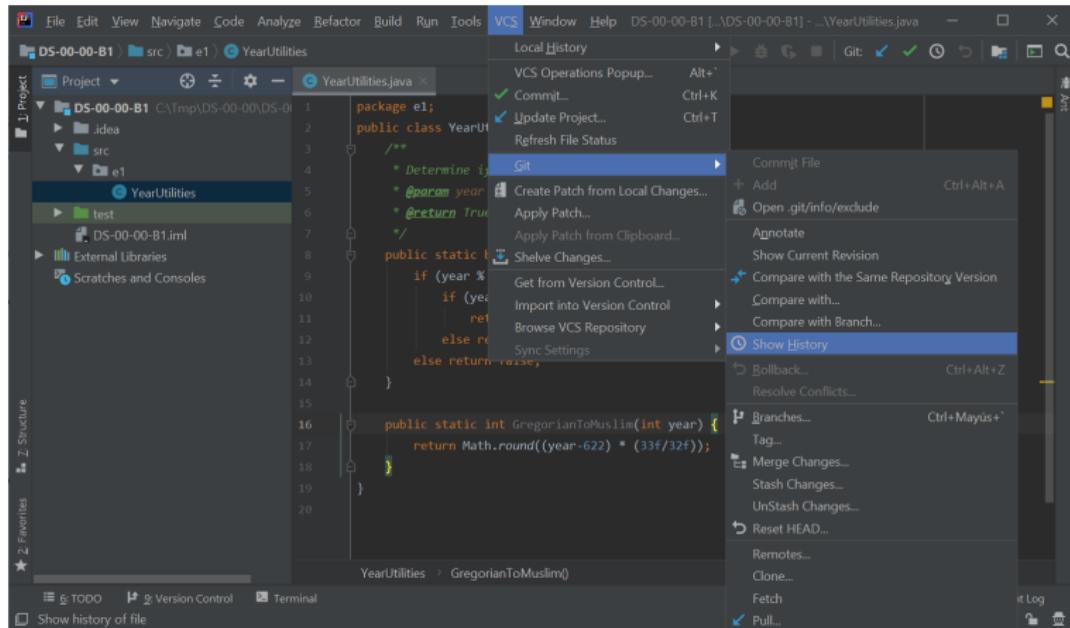
- File Menu:** File, Edit, View, Navigate, Code, Analyze, Refactor, Build, Run, Tools, VCS, Window, Help.
- Toolbar:** DS-00-00-B1, Add Configuration..., Git: master.
- Project Structure:** DS-00-00-B1, C:\Tmp\DS-00-00\DS-01, idea, src, e1, YearUtilities, test, DS-00-00-B1.iml, External Libraries, Scratches and Consoles.
- Code Editor:** YearUtilities.java, showing Java code for determining leap years and converting Gregorian to Muslim dates.
- Status Bar:** 1 file updated in 2 commits // View Commits (moments ago), 16:52, CRLF, UTF-8, 4 spaces, Git: master.
- Bottom Status:** 1 file updated in 2 commits // View Commits (moments ago).

A tooltip at the bottom right indicates "1 file updated in 2 commits" and "View Commits".



History

- At any point, we can review the history of any file...



History

- ...see a list of all the changes and mark two version for comparison...

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project View:** Shows the project structure for "DS-00-00-B1".
- Code Editor:** Displays the file "YearUtilities.java" with the following code:else return false;
}
public static int GregorianToMuslim(int year) {
 return Math.round((year-622) * (33f/32f));
}
- History Tool Window:** Titled "History: YearUtilities.java", it lists the commit history:

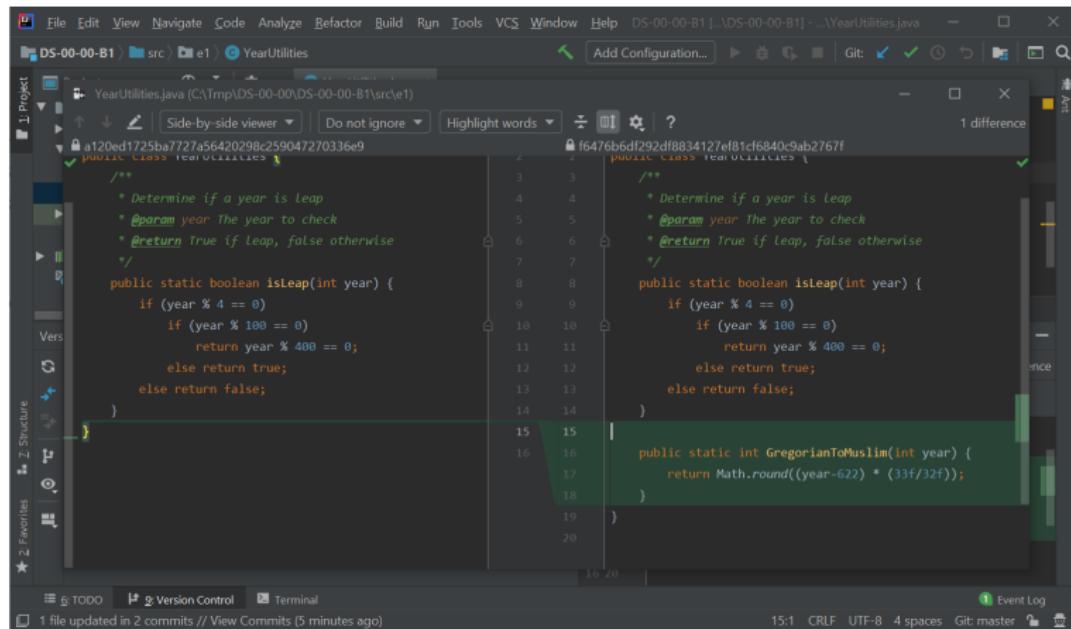
Author	Date	Commit Message
Eduardo Mosqueira Rey	24/01/2020 20:27	Merge remote-train > origin / master
Eduardo Mosqueira Rey	24/01/2020 20:13	New javadoc. Simplified IF.
Eduardo Mosqueira Rey	24/01/2020 20:09	Gregorian to Muslim
Eduardo Mosqueira Rey	24/01/2020 19:45	New javadoc. Simplified IF.
Eduardo Mosqueira Rey	24/01/2020 19:22	First commit of exercise 1: leap years

The commit at index 4 is selected.

- Status Bar:** Shows "1 file updated in 2 commits // View Commits (4 minutes ago)" and "16:52 CRLF UTF-8 4 spaces Git: master".


History

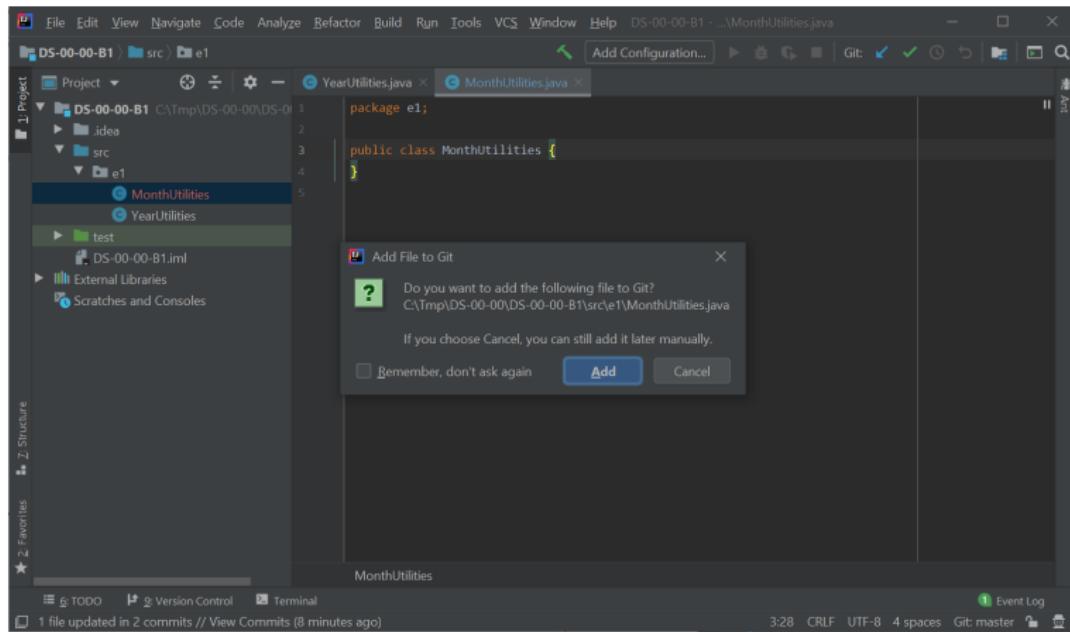
- ...and see their differences in a *diff* window (*Ctrl+D*).



```
diff --git a/120ed1725ba7727a56420298259047270336e9 b/f6476b6df29d1f8834127e81cf68409ab2767f
--- a/120ed1725ba7727a56420298259047270336e9
+++ b/f6476b6df29d1f8834127e81cf68409ab2767f
@@ -1,14 +1,16 @@
 /**
  * Determine if a year is leap
  * @param year The year to check
  * @return True if leap, false otherwise
  */
-public static boolean isLeap(int year) {
-    if (year % 4 == 0)
-        if (year % 100 == 0)
-            return year % 400 == 0;
-        else return true;
-    else return false;
-}
+public static boolean isLeap(int year) {
+    if (year % 4 == 0)
+        if (year % 100 == 0)
+            return year % 400 == 0;
+        else return true;
+    else return false;
+}
+
+public static int GregorianToMuslim(int year) {
+    return Math.round((year-622) * (33f/32f));
+}
```

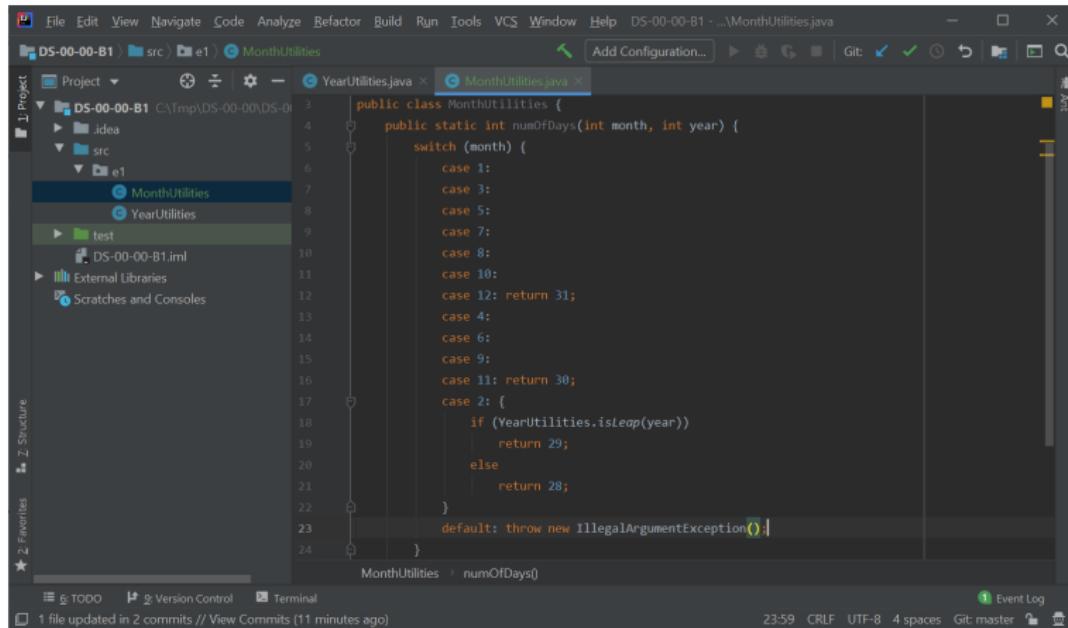
Adding new files to the repository

- Finally, every time a new class is created in our project, the version control system will ask if we want to include it.
- We must choose *Add* in order to include the new file.



Adding new files to the repository

- When it is included, it will show up in green...



The screenshot shows the IntelliJ IDEA interface with the following details:

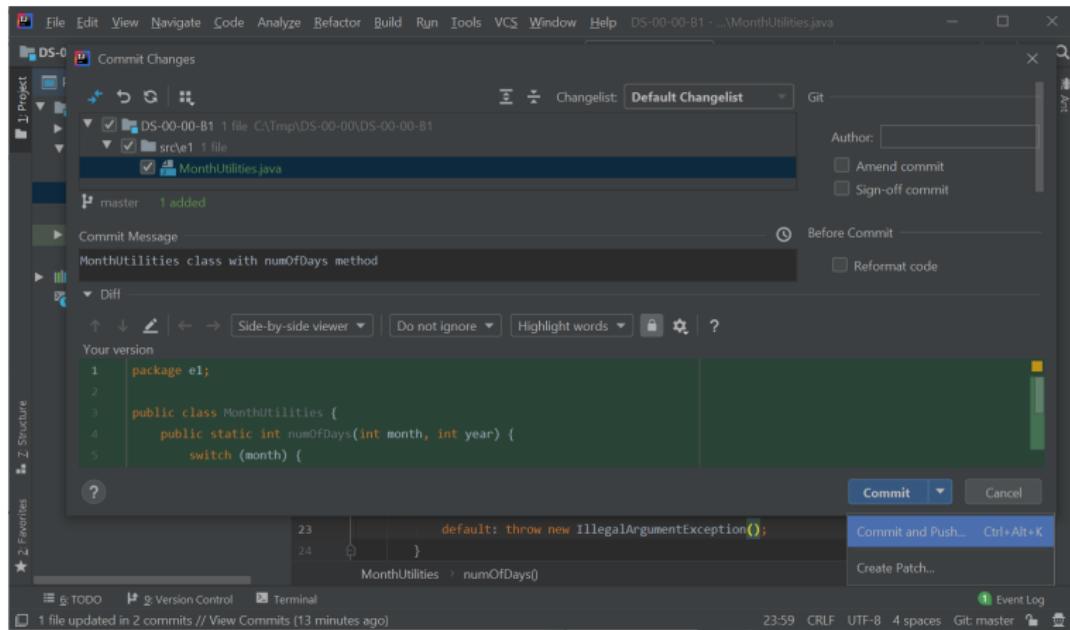
- Project Bar:** Shows "DS-00-00-B1" as the current project.
- Toolbars:** Standard IntelliJ toolbars for File, Edit, View, Navigate, Code, Analyze, Refactor, Build, Run, Tools, VCS, Window, Help.
- Status Bar:** Shows "1 file updated in 2 commits // View Commits (11 minutes ago)" and "23:59 CRLF UTF-8 4 spaces Git: master".
- Code Editor:** Displays the `MonthUtilities.java` file with the following code:

```
public class MonthUtilities {  
    public static int numOfDays(int month, int year) {  
        switch (month) {  
            case 1:  
            case 3:  
            case 5:  
            case 7:  
            case 8:  
            case 10:  
            case 12: return 31;  
            case 4:  
            case 6:  
            case 9:  
            case 11: return 30;  
            case 2: {  
                if (YearUtilities.isLeap(year))  
                    return 29;  
                else  
                    return 28;  
            }  
            default: throw new IllegalArgumentException();  
        }  
    }  
}
```
- Sidebar:** Shows the "Project" view with the file structure: DS-00-00-B1, .idea, src, e1, MonthUtilities, YearUtilities, test, DS-00-00-B1.iml, External Libraries, Scratches and Consoles.
- Bottom:** Shows the "Event Log" tab.



Adding new files to the repository

- ...and the code will be uploaded in the next *commit & push*.



Final recommendations

- **Do commit & push frequently**, do not wait for changes to pile up.
- **Do also pull frequently** to be always up-to-date.
- Try to give a purpose to the **commits**. Do not *commit* random changes to multiple unrelated classes.
- **The commit message must be clear** and state the goal unambiguously. Long and detailed messages are preferable to simple “some changes” messages.
- **Arrange the responsibilities in order to minimize conflicts** when *merging*. If you do *pair programming*, no conflicts will occur, since both of you will be working on the same local copy.
- **Do not leave commits for the last moment**, computers sometimes fail at the worst possible moment.
- **Always check that changes have been successfully uploaded on the web page**. Even better, download a fresh copy to check it thoroughly.



Practice 4: Git

Diseño Software (614G01015)

Eduardo Mosqueira Rey (Coordinador)

Departamento de Ciencias de la Computación y Tecnologías de la Información
Facultad de Informática



UNIVERSIDADE DA CORUÑA