# Java Fundamentals

## Lesson 2: Version Control Systems. Git

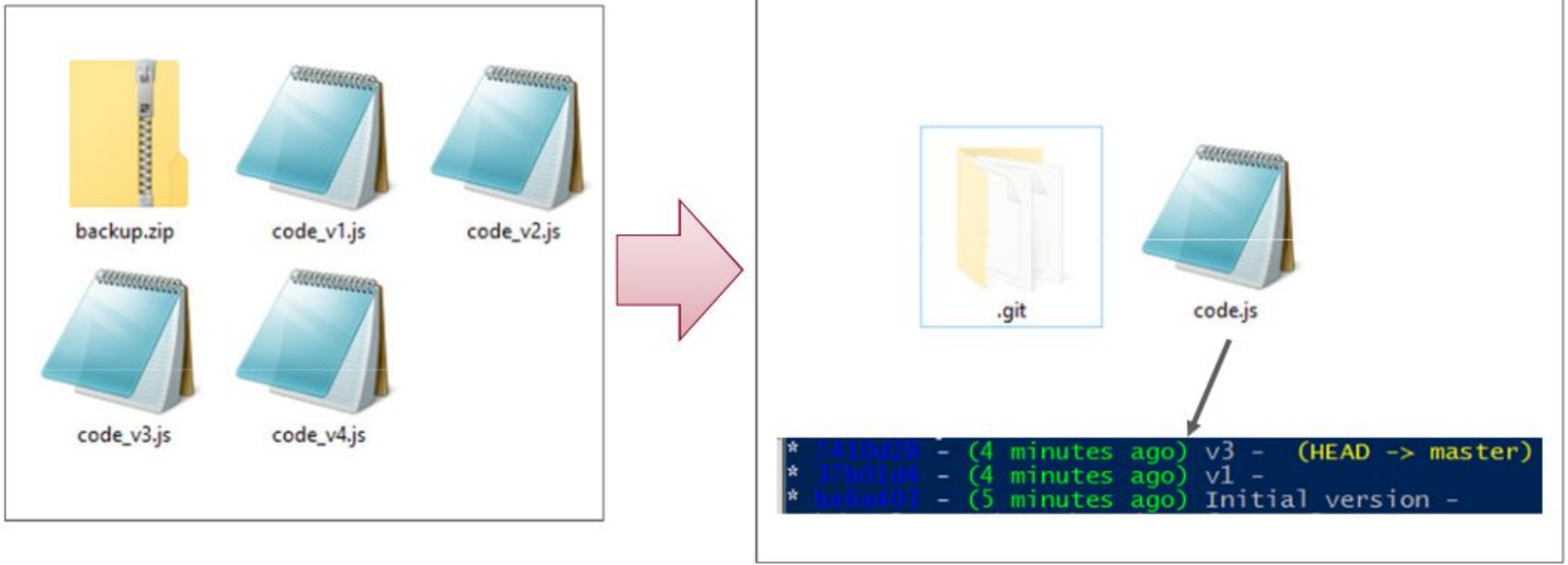
Speaker: Nicolae Sîrbu
Alexandru Umaneț

# Lesson Objectives

- Introduction to Version Control System
- Introduction to Git
- Performing basic Git operations

# Introduction to Version Control Systems

# Version Control Basics



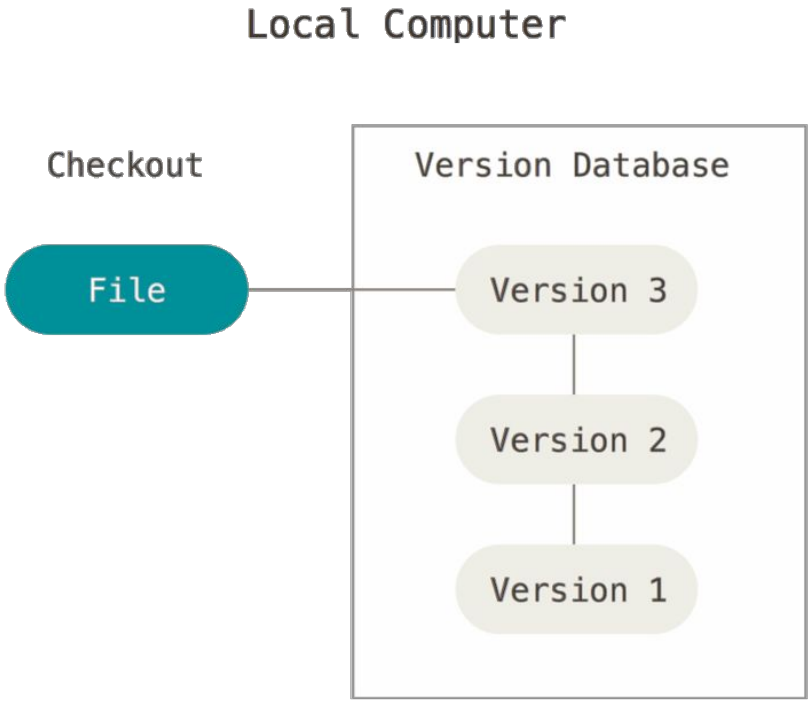
# Version Control Systems

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

There are three types of version control systems:

- Local
- Centralized
- Distributed

# Local Version Control System

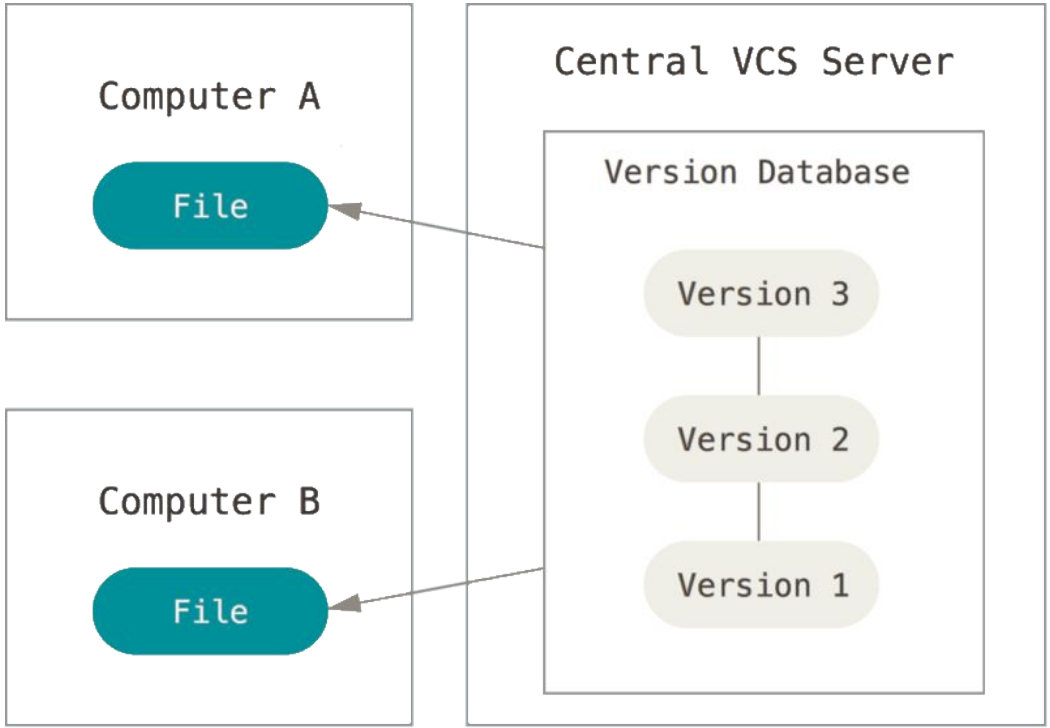


# Local Version Control System

- \+ Simple
- − Error prone
- − Useful to only a single developer at a time
- − No collaboration

Ex: Revision Control System on MacOS

# Centralized Version Control System

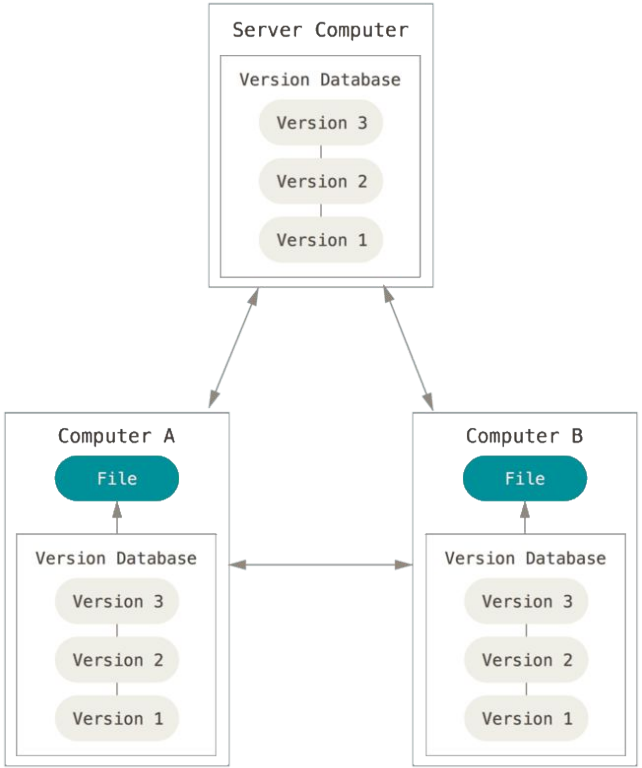


# Centralized Version Control System

- \+ Easier to administrate a central database instead of several local databases.
- − Single point of failure if the database goes down, gets corrupted, and so on.
- − Central server is prone to hacking.
- − It’s a very bad idea to store all changes to code in a single place.

Ex: CVS, Subversion, Perforce

# Distributed Version Control Systems



# Distributed Version Control Systems

- Enables developers to work with several remote repositories.
- Each clone is a full backup of all of the data.
- Possible to work offline.

Ex: Git, Mercurial.

# Introduction to Git

# States of Files in Git

In Git, your files can be in one of the following states:

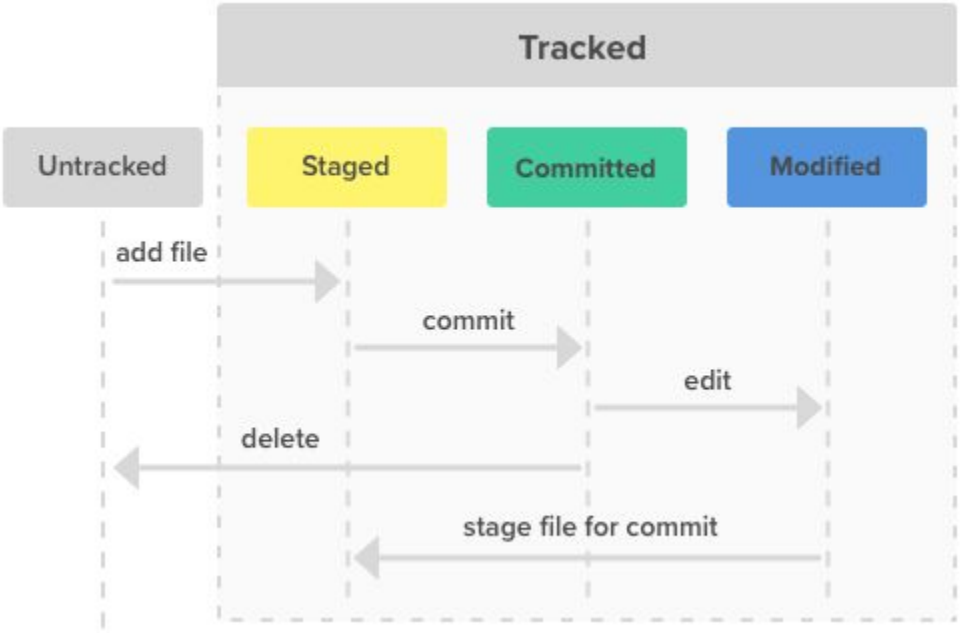
- **Untracked**: These are any files that are not being tracked, Git isn't aware of the existence of these files.

- **Staged**: You have marked the file in its current version to go into your next commit snapshot.

- **Modified**: You have made changes to the file but have not committed it to your database yet.

- **Committed**: The data is safely stored in your local database.

# States of Files in Git



# Sections of a Git Project

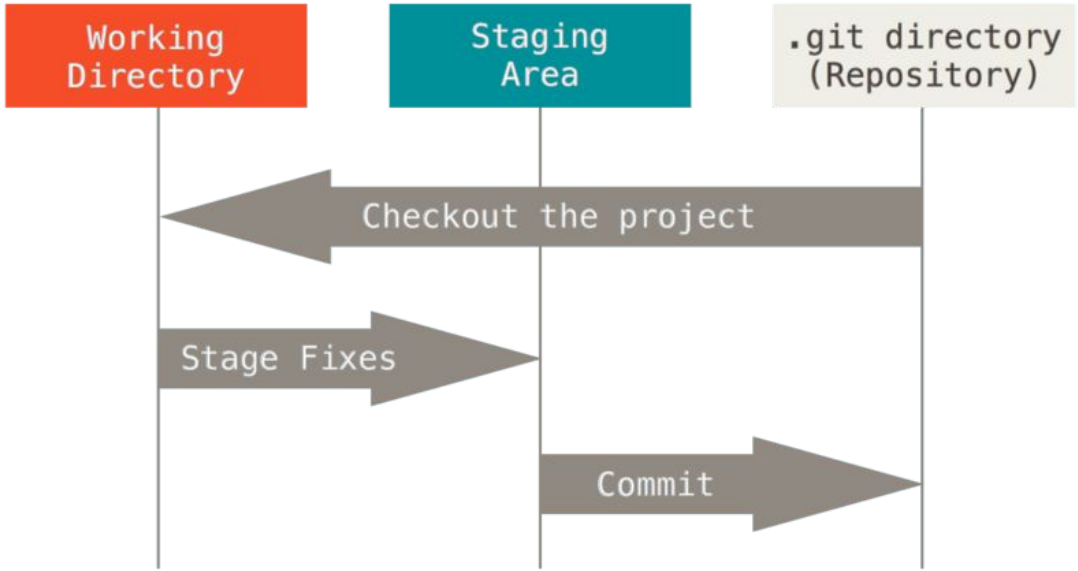
There are three main sections of a Git project:

- **Git directory**: Contains metadata and the object database for your project or what is copied when you clone a repository from another computer.

- **Working directory**: A single checkout of one version of the project, where files are pulled from the compressed database in the Git directory and put on disk to use or modify.

- **Staging area**: A file that stores information about what will go into your next commit.

# Sections of a Git Project



# Exercise #2.2 Git installation

1. Download Git from **https://git-scm.com/downloads** and install it.
2. Run Git on Windows, use the provided "git bash" or "git shell" application to open a command line with Git integration.
3. Define your identity:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

4. Check your Git version:

```
$ git --version
```

5. Display the Git help screen:

```
$ git –help
```

# Basic Git operations

# Basic Git operations

| Command | Operation |
| --- | --- |
| `git init` | Create a new repository in an existing project or directory |
| `git add` | Start file tracking and stage changes |
| `git commit` | Commit changes |
| `git status` | View changes since last commit |
| `git rm` | Stop file tracking |
| `git mv` | Rename a file |
| `git clone` | Clone an existing repository from another server |
| `git pull` | Pull changes from remote server and merge with files in working directory |
| `git push` | Push local changes to a remote server |

# Using the `git init` command

Create a new Git repository in an existing project or directory:

```
$ git init
```

This creates a `.git` subdirectory that contains the repository files in the project directory, but nothing in your project is tracked yet.

# Using the `git add` command

Begin tracking a new file or all files in a directory:

```
$ git add .
```

- The command accepts a path name for a file or for a directory.
- If the path is for a directory, the command adds all files and all subdirectories in that directory.
- Git stages a file exactly as it is when `git add` is run.
- If the file is modified after the command is run, the command must be rerun to **stage** those modifications.
- Use `git status` to list files that have changes staged for the next commit.

# Using the `git commit` command

Store the current content of the index in a new commit with a message describing the changes:

```
$ git commit -m "commit message"
```

Use `-m "<msg>"` as the commit message.

# Using the `git status` command

Shows the status of the working tree:

```
$ git status
```

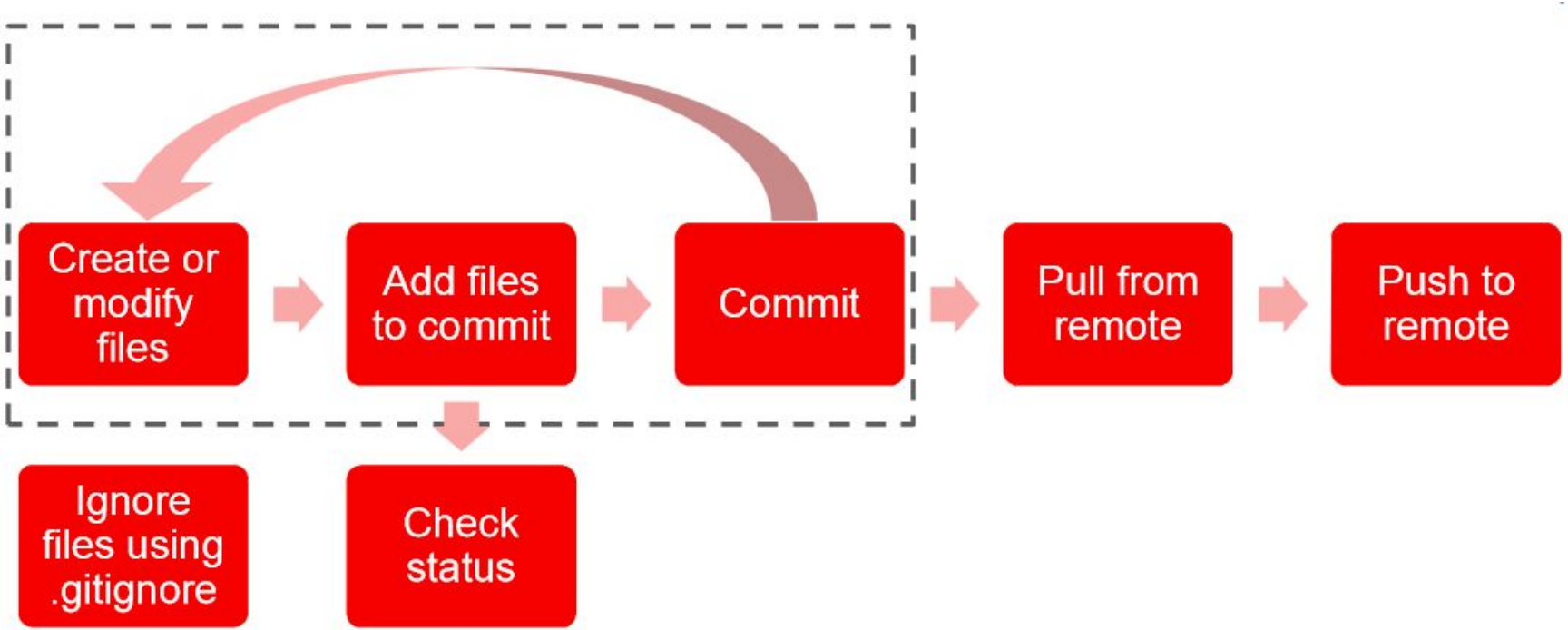
Answers two questions:

- What have you changed but not yet staged?
- What have you staged that you are about to commit?

Use the `-s` or `--short` option to see input in short-format.

Use the `-v` or `--verbose` option (or the `git diff` command) to also see file level changes.

# Git Workflow



# Exercise #2.3 Create a Git repository

1. Create a Git repository on your machine.
2. Create  a file in the working directory.
3. Check the status of the working directory.
4. Stage the file.
5. Check the status of the working directory.
6. Make a commit.
7. Check the status of the working directory.

# HM. Exercise #2.4 Git practice

1. Create a directory on your machine called `git_practice`.
2. Enter the `git_practice` directory and initialize a Git repository.
3. Create 3 new files.
4. Check the status of your Git repository.
5. Start tracking these files by adding them to Git. Use "`git add .`" to stage all the files.
6. Check the status of your Git repository.
7. Commit your changes.
8. Edit a couple of your files with some new content.
9. Check the status of the repository.
10. Stage your files for the next commit.
11. Commit your new changes.
12. View the history of commits using the "`git log`" command.

# Ignoring files that should not be committed

A `.gitignore` file tells Git which files and directories to ignore, before you make a commit.

The `.gitignore` file must reside in the working directory, not in the `.git` directory.

Use patterns to delineate files to ignore, rather than explicitly listing files and directories.

# Ignoring files that should not be committed

The rules for the patterns you can put in the `.gitignore` file are as follows:

- Blank lines or lines starting with # are ignored.
- Start patterns with a forward slash (/) to avoid recursivity.
- End patterns with a forward slash (/) to specify a directory.
- An asterisk (*) matches zero or more characters.
- [abc] matches any character inside the brackets (a, b, or c in this case).
- A question mark (?) matches a single character.
- Brackets enclosing characters separated by a hyphen ([0-9]) match any character between them (0 through 9 in this case).
- Two asterisks match nested directories, for example: a/**/z would match a/z, a/b/z, a/b/c/z, and so on.

# `.gitignore` file content example

```
# Build results
[Dd]ebug/
[Dd]ebugPublic/
[Rr]elease/
[Rr]eleases/
x64/
build/
bld/
[Bb]in/
[Oo]bj/
```

# Removing files from the repository

Remove files from the working tree:

```
$ git rm [file]
```

It will not remove modified files that have been staged, unless the removal is forced with `-f`.

# Renaming files in Git

Move or rename a file or a directory.

```
$ git mv [old-file-name] [new-file-name]
```

The renaming is finalized at the next commit.

# Using the `git clone` command

Sets up your local master branch to track the remote master branch on the server you cloned from.

Get a copy of an existing Git repository:

```
$ git clone [url]
```

Get a copy of an existing Git repository with a name that is different from the original repository name:

```
$ git clone [url] [new-repository-name]
```

A cloned remote repository is automatically added under the name “origin.”

# Using the `git pull` command

Fetches data from the server you cloned, and automatically tries to merge files into the code that you are currently working on.

Fetch from and integrate with another repository or a local branch:

```
$ git pull
```

# Using the `git push` command

Push your changes upstream to a remote server and update remote refs and associated objects:

```
$ git push [remote-name] [branch-name]
```

If you cloned your repository, use this command to push your master branch and all its commits to your origin server:

```
$ git push origin master
```

Proactively do a `git pull` before doing a `git push` to avoid most conflicts.

# Exercise #2.5 Working with remote repositories

1. Register on GitHub, if you don't have an account yet: **https://github.com**.

2. Create a repository on GitHub.
   (https://help.github.com/articles/create-a-repo/)

3. Link the repository that you created in *Exercise #2.3* with the repository from GitHub.
   (https://help.github.com/articles/adding-an-existing-project-to-github-using-the-command-line/)

4. Push your local changes to the remote repository. Use `git push` command.

# Exercise #2.6 Cloning a repository

1. Clone this repository from GitHub:
   **https://github.com/nsirbu/Tekwill-Java-Fundamentals-nov2019**
2. Add a new file to your working copy (the repository you've just cloned), inside *Lesson_2/StudentFiles/* folder. Put your initials in the file name, e.g. `Nicolae Sirbu -> nsirbu`.
3. Make a commit.
4. Push your local changes to the remote repository.

* **Note**: Before pushing your local changes, don't forget to first pull changes from the remote repository. Use `git pull` command.

# Exercise #2.7 Ignoring things

1. Create a repository on your GitHub account called *ProjectSample*.
2. Clone the repository on your computer.
3. Copy the *ProjectSample* folder from *Lesson_2/ProjectSample* to your working copy.
4. Create a `.gitignore` file with the following configurations:
    a. Ignore the `build` directory;
    b. Ignore the `dist` directory;
    c. Ignore all `.log` files from `log` directory;
    d. Ignore the `dat` folder from project resources (`src/main/resources` ).
5. Push your local changes to the remote repository.
6. Check out your remote repository on GitHub. Notice the differences from your working copy.

# Using the `git checkout` command

Replaces your local changes with the ones from the remote repository, basically cancels the changes that are not committed yet.

Usage:

```
$ git checkout .

$ git checkout <filename>
```

# Using the `git revert` command

This command can be considered an 'undo' type command, however, it is not a traditional undo operation.

Instead of removing the commit from the project history, it figures out how to invert the changes introduced by the commit and appends a new commit with the resulting inverse content.

Usage:

```
$ git revert <commit_ID>
```

To see the IDs of the commits use `git log` command.

# Exercise #2.8: Git Basics review

1. Create a repository on your GitHub account called `git_basics`.
2. Clone this repository on your computer.
3. Navigate to the folder that contains the repository which you've just cloned.
4. Create a file called `basics.txt` inside it.
5. Add `basics.txt` to the staging area.
6. Commit with the message `"adding basics.txt"`.
7. Check out your commit with `git log`.
8. Open the `basics.txt` file and add some lines of text. Save and close the file.
9. Add `basics.txt` to the staging area.
10. Commit with the message `"updating basics.txt"`.
11. Push the changes from your local repository to the remote repository.

# HM. Exercise #2.9: Git Advanced review

1. Create a folder on your computer called `git_advanced`.
2. Enter the `git_advanced` folder.
3. Create a file called `one.txt`.
4. Initialize an empty git repository inside `git_advanced` folder.
5. Add `one.txt` to the staging area.
6. Commit with the message `"adding one.txt"`.
7. Check out your commit with `git log`.
8. Create another file called `two.txt`.
9. Add `two.txt` to the staging area.
10. Commit with the message `"adding two.txt"`.
11. Remove the `one.txt` file.
12. Add this change to the staging area.
13. Commit with the message `"removing one.txt"`.
14. Create a repository on your GitHub account called `git_advanced`.
15. Link your local repository with the remote repository `git_advanced`.
16. Push the changes from your local repository to the remote repository.

# HM. Exercise #2.10.1 Ignore unwanted files

It is often a good idea to tell Git which files it should track and which it should not. Developers almost always do not want to include generated files, compiled code or libraries into their project history.

Your task is to create and commit a configuration that would ignore:

- all files with `exe` extension
- all files with `o` extension
- all files with `jar` extension
- the whole `libraries` directory

# HM. Exercise #2.10.2 Ignore unwanted files

1. Create a repository on your GitHub account called `git_ignore_unwanted_files`.
2. Clone this repository on your computer.
3. Navigate to the folder that contains the repository which you've just cloned.
4. Create a file called `.gitignore` inside it with the configurations from the previous slide.
5. Perform some tests and make sure the configurations that you've defined are correct.
6. Push your local changes to the remote repository.

# Resources

An introduction to version control
(https://www.codebasehq.com/blog/an-introduction-to-version-control)

git - the simple guide
(http://rogerdudler.github.io/git-guide/)

Git Explained: For Beginners
(https://dzone.com/articles/intro-git)

# Java Fundamentals

## Lesson 2: Version Control Systems. Git

End.

Speaker: Nicolae Sîrbu
Alexandru Umaneț