Workshop 1: Maven and Git

The workshop will introduce Maven and Git. After this workshop you will be able to set up your project development environment.

If using your own computer this assumes you have Java 8 installed!

Introducing Maven

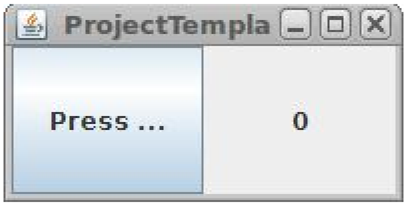
If using your own computer you may need to install Maven locally on your computer.

We will use Maven from the command line for now (a shell/terminal). Later on it's possible to do this from inside any IDE (in particular NetBeans, NetBeans has built in Maven support).

Linux and Mac already have decent shells. The Windows shell is very primitive. Possibly the Powershell is better?

1. When Maven installed, try this (command "tree" is useful in Linux/Mac-terminal).

2. Now download the project template from the course page. Do the following
    a. Inspect the project in particular the pom.xml
    b. Execute: mvn compile in same directory as pom.xml (libraries will be downloaded from the central repository). Inspect project directories (output will show up in terminal).
    c. Try to run the project (the *.class files) from the command line. Should look like (or similar):



    d. Execute: mvn clean and inspect.
    e. Execute: mvn package again and <u>really</u> inspect everything (also the HTML-pages).
    f. Previous step created an executable jar-file (i.e a program), try to run it from the command line.

Add a Dependency

The main reason for us to use Maven is the dependency handling capabilities. Always as a first step try to find useful libraries (don't reinvent the wheel).

1. Create another my-app project as in the first point above.

2. We'll create a class Monster (for some game) but we're bored with the boilerplate coding. We decide to use Lombok (inspect features).

3. Add a dependency on the lombok library into the pom.xml (this is simplified if using NetBean, just right click on Dependencies > Add dependency …)

4. Using your favorite IDE, code the Monster class and use the lombok annotations to reduce the boilerplate code (use your imagination and as much as possible from Lombok).

5. Use Maven (possibly from inside IDE) to compile and package the program (Lombok should be downloaded). Try to run the (useless) program to see if it works. If using an IDE, the generated setters/getters will show up in code completion!

6. Inspect the generated jar file, is lombok packaged with it (easy to inspect in NetBeans)?

7. Inspect your local Maven repository, the .m2 directory. Locate Lombok in it.

Introducing Git

You need some reading before starting with this. Start here and give it a quick look. Try to grasp the basic ideas (later go back for clarification).

A Git Workflow

Read about Git branches, then read this and finally exercise it (further down)

When using Git we need an ordered process, a **workflow**. Our workflow uses three branches:

- The remote branch "master" (default name), also known as origin/master. Created when remote repo is created. Always exists.

- The local branch master. The local copy of the remote master. Created when project is cloned from the remote repo. Could be up to date with origin/master or not. Always exists.
- A local branch named as “the task to be accomplished”. This branch is created from the local master for each task. This branch is strictly private for the contributor. When task is fulfilled the branch is merged into local master (the remote master is subsequently updated with the changes in the local master). Finally the branch is deleted.

Workflow Steps

Assume we have a remote repo and have cloned it locally. We now start working with the local clone.

1. git checkout master // You are probably already on local master
2. git pull // (= git pull origin master) To update local master
3. git checkout -b myTask // Create and switch to branch myTask
4. Optional: Use git branch -a to show all branches and git status to see what branch you’re on.
5. Do some coding (some well defined (sub) task), after max 30 min. … go to 6.
6. git commit -a -m “blablabla” // Commit on branch myTask (use better message)
7. Goto 5. until task finished (max 2 hours before next point)
8. Ok, assume task finished. Project should be executable, all test should pass.
9. Now we start to integrate our changes into the project (Tip: zip current code)
    a. git checkout master
    b. git pull // Update local master again
    c. git checkout myTask // Back to our private branch
    d. git rebase master // Integrate myTask on top of master
    e. git checkout master
    f. git merge myTask // Fast forward
    g. git push // (= git push origin master) Push to remote repo
    h. git branch -d myTask // Delete branch

10. Now everybody should be able to see our contribution (i.e. git pull).
11. Continue with next task, go to 2.

Exercising the Workflow

To exercise the workflow (locally) do the following

1. We need three Git directories. Select a top level directory to host them all (I call it “tmp” from now)

2. We will simulate a remote repo on your local machine. Create a directory ooproj.git in tmp.

3. Step into the created directory, execute

   ```
   git --bare init
   ```

4. Go back to tmp. Clone the remote repo twice (will simulate two different users local repos)

   ```
   git clone ooproj.git ooprojA.src
   git clone ooproj.git ooprojB.src
   ```

5. There should now be three directories in tmp: ooproj.git, ooprojA.src and ooprojB.src.

6. Create a Java-file with some simple code in ooprojA.src, then in same directory, execute:

   ```
   git status      (just to check)
   git add .       (must add new files)
   git status
   git commit -a -m init
   git status
   git push
   ```

   Let ooprojB.src execute: git pull. Both should now have the same code.
   Use

   ```
   gitk --all (for both A and B)  // Or other graphical tool
   ```

7. The clones to represent two different users. Open one terminal for ooprojA.src and one for ooprojB.src. Let users work on the code using the workflow. Use commit message to identify the changes. Use gitk --all (or similar) for both users after each step in the workflow.
   All contributions should appear in a linear fashion when inspecting in gitk. If not, you messed up. Remove everything and start from 1.

   If getting a conflict, see and also below

Merge Conflicts

Now we'll create a conflict and then resolve it. Continue with the Git-repos from above.
We assume both users have the same and latest versions of the code and uses the workflow.

1. Let user A modify a code row (note which one), commit and push it to origin (a full workflow cycle).

2. Let user B modify same row and commit, later in workflow when ...

3. … B does the rebase step, a message should show up:

   ```
   ...
   CONFLICT (content): Merge conflict in ....java
   Failed to merge in the changes.
   Patch failed at 0001 …
   ...
   ```

4. Open an editor in ooprojB and resolve the conflict (edit the code, there are markings like <------- and more, decide what to keep) then execute git add . to mark as resolved.

5. Execute git rebase --continue in same directory. Conflict hopefully solved.

Setup GitHub

We prefer GitHub as our remote repository for the project (if you own a server it's also possible to host the remote repository on the server).

1. All group member must register on GitHub (note name and password!).

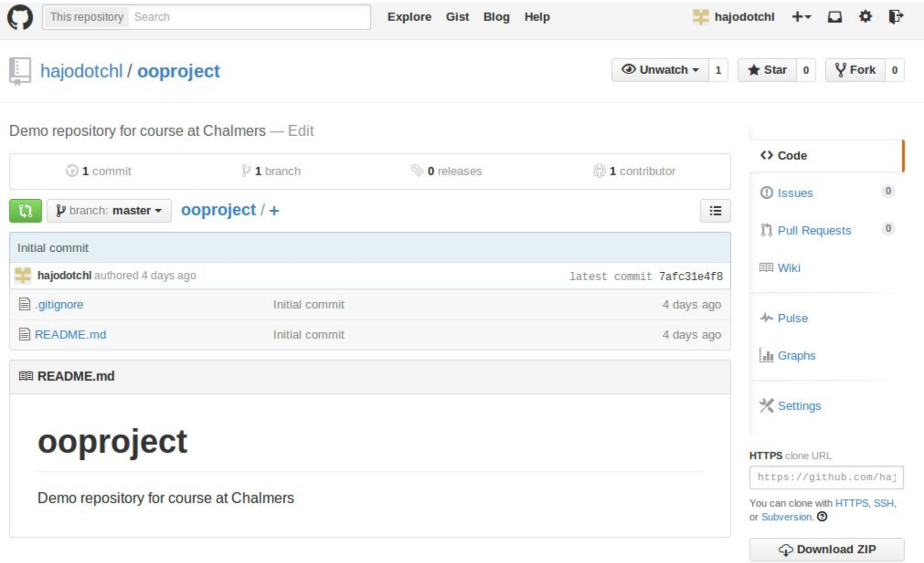
2. All should read and do step 1, Setup Git, on GitHub Bootcamp. To check use:

   ```
   git config -l
   ```

   Also to see how to cache password (don't need to login for every push)

3. Select one group member as repository owner. Let owner create a repository, follow step 2 on Bootcamp (use a good name!). Choose to create an initial gitgnore file for Java and a README-file. WARNING: gitignore should be adapted to your IDE (not pushing IDE specific files to the repo). See Web.

4. When finished the repository owner should have a page similar to the one below:



hajodotchl / **ooproject**

Demo repository for course at Chalmers — Edit

Initial commit

hajodotchl authored 4 days ago `latest commit 7afc31e4f8`

.gitignore	Initial commit	4 days ago
README.md	Initial commit	4 days ago

README.md

ooproject

Demo repository for course at Chalmers

HTTPS clone URL

`https://github.com/haj`

You can clone with HTTPS, SSH, or Subversion.

5. Add collaborators for the repository (collaborators must be registered on GitHub).

Clone the Remote Repository

Let (remote) repository owner do the following:

1. Select a local directory where you would like to have your local repository.

2. Step into directory and execute (copy your URL from GitHub, last part is name of local directory for the local repository):

```
git clone https://github.com/hajodotchl/ooproject.git ooproject.src
```

3. Inspect directory for the local repo! Files from remote should be in the local repository. There should be a hidden .git-directory and a .gitignore-file, check!

4. Execute: git remote -v to check that the remote is ok. Should list (like):

```
origin   https://github.com/hajodotchl/ooproject (fetch)
origin   https://github.com/hajodotchl/ooproject (push)
```

5. Copy the content of the Maven project template to the local repository directory. Execute: git status. Should look like (delete directory “target” if present):

```
Untracked files:
(use "git add <file>..." to include in what will be committed)

findbugs-exclude.xml
pmd-rules.xml
pom.xml
src/
```

6. Rename the project content to your taste (name, artifactId ... in pom.xml, packages in src, etc). Open project in your IDE and refactor/inspect (this is easily accomplished in NetBeans). Make it really beautiful!

7. When satisfied, add and commit everything to the local repo, execute

```
$ git add .          (yes, dot last)
$ git status         (a list of "new" files should appear)
$ git commit -a -m initial
$ git status  (...ahead of 'origin/master' by 1 commit)
$ git log
```

8. Now push the local repo content (project) to the remote repo, execute: git push and then git status

9. Go to GitHub (possibly refresh page). Magic (hopefully)!

10. Check GitHub contributors (should be one and only one, the repo owner).

11. Let everyone clone the repo and start working using the workflow introduced earlier.

Pair programming

If working in pairs you have to share the commits, commit from different machines and/or use the --author when committing.

If you don't distribute commits evenly you will fail the course because of too little contribution to the project!

Also use the @author annotation in Java classes to verify you work.
Short list of useful Git commands here (also other links on course page)