Paradigme de Programare - Laboratorul 1

Utilizarea IDE-ului IntelliJ și a sistemului de versionare Git

Maniera de evaluare la această disciplină

- Participarea la orele de curs și de laborator: este recomandată la curs și obligatorie la laborator:
- Neparticiparea la mai mult de 50% din laboratoare conduce la refacerea disciplinei.
- Neparticiparea la curs conduce la probleme la examenul teoretic
- Nefacerea temelor la timp (adică săptămânal) conduce în 95% din cazuri la imposibilitatea de a realiza în timp util a subiectelor practice la examen (procent obținut pe analiza notelor din ultimii trei ani)

Evaluarea de laborator: pentru a putea lua 10 la laborator trebuie ca studentii sa fie capabili la intrebarile asistenților (minim 2-3 de student) cu privire la conținutul cursului curent (și pentru care a fost creat laboratorul) - 30% -

Examen final 70% (este o singura notă) defalcată astfel:

Proba de laborator – 40 % cu bilete (trase din teanc) și două ore maxim la dispoziție. Un subiect din două trebuie să fie îndeplinit integral pentru a se putea nota (min 5).

Proba teoretică – 40% - test docimologic - conține și întrebări cu caracter practic specifice laboratorului (min 5)

Teme acasă: 20% (atentie ca efect al acestui procent se poate întâmpla ca 5 la subiectul practiv si 5 la teorie ceea ce face 5*0.8=4!!! deci să picați din motive de teme nepredate

În caz de variantă on line (SARS-CoV-II)

la testul practic - se aleg automat doua probleme din pool și se trimit

la testul teoretic - oral cu intrebari selectate automat din pool de către aplicație

Introducere

În cadrul acestui laborator va trebui să vă obișnuiți cu mediul de dezvoltare al aplicațiilor IntelliJ și Pycharm. Pentru aceasta veți încărca o serie de proiecte deja realizate în diverse limbaje care vor fi folosite pe parcursul acestei materii, le veți analiza, compila și executa pas cu pas urmărind și ferestrele de afișare a conținutului variabilelor implicate de-a lungul execuției.

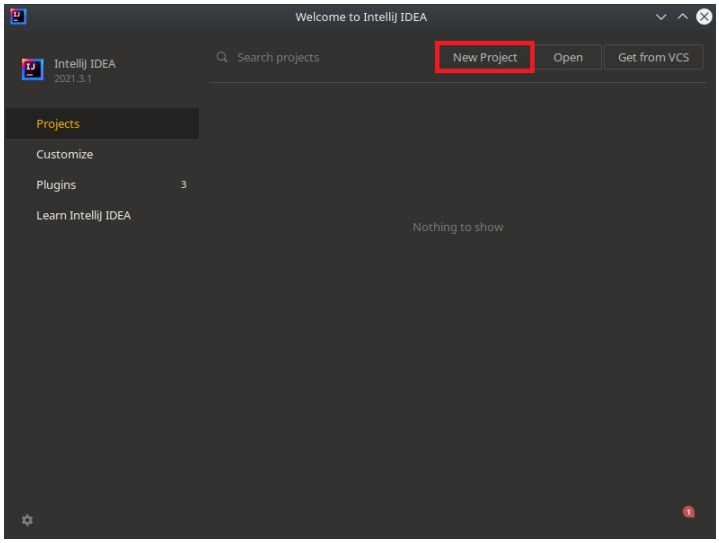
Resursele specifice sunt disponibile la http://mike.tuiasi.ro/

1. Crearea proiectelor JAVA utilizând IntelliJ IDEA

1. Crearea unui proiect nou
2. Construcția unui profil de compilare (engl. „build”)
3. Build
4. Adăugarea unui profil de execuție
5. Execuția
    1. Cum facem depanarea unui program (engl. „debug”)
    2. Cum urmărim variabilele
6. Deschiderea unui proiect existent

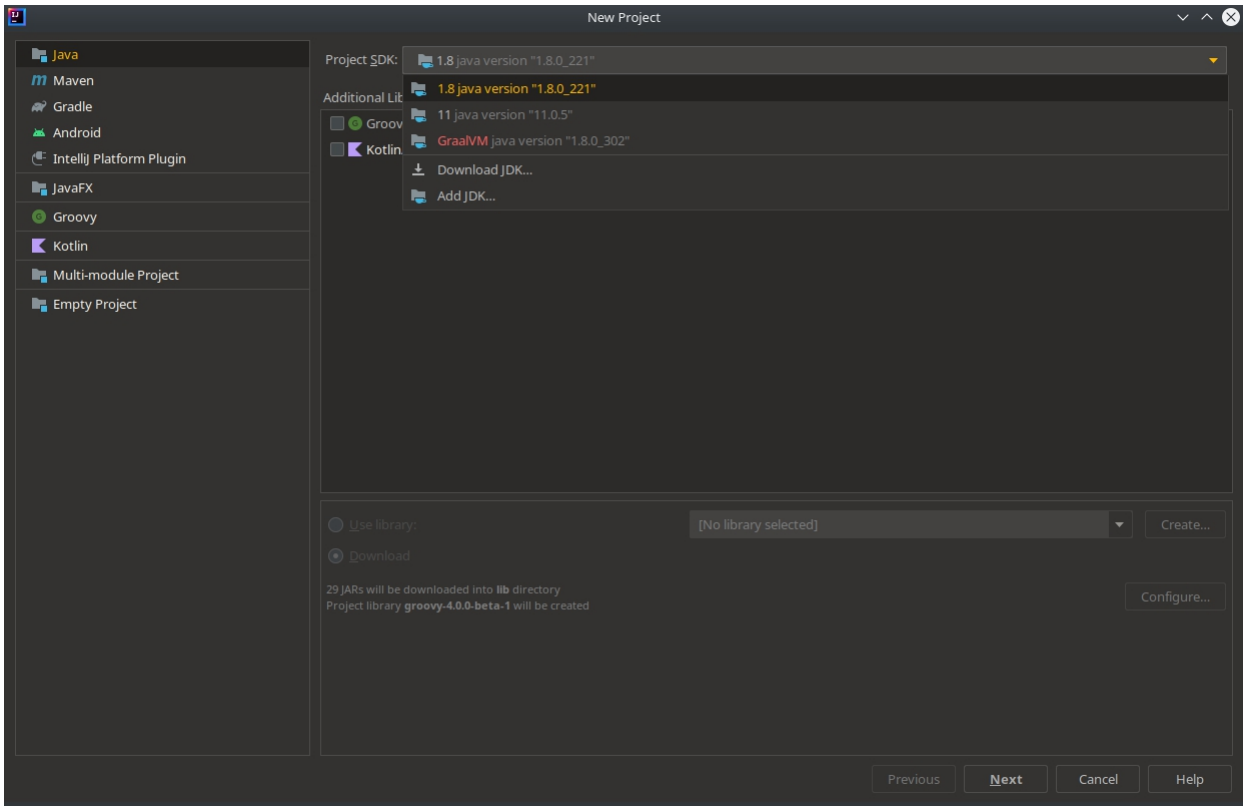
1.1. Crearea unui proiect nou

Se va deschide IDE-ul IntelliJ IDEA de pe desktop și se va crea un proiect nou.

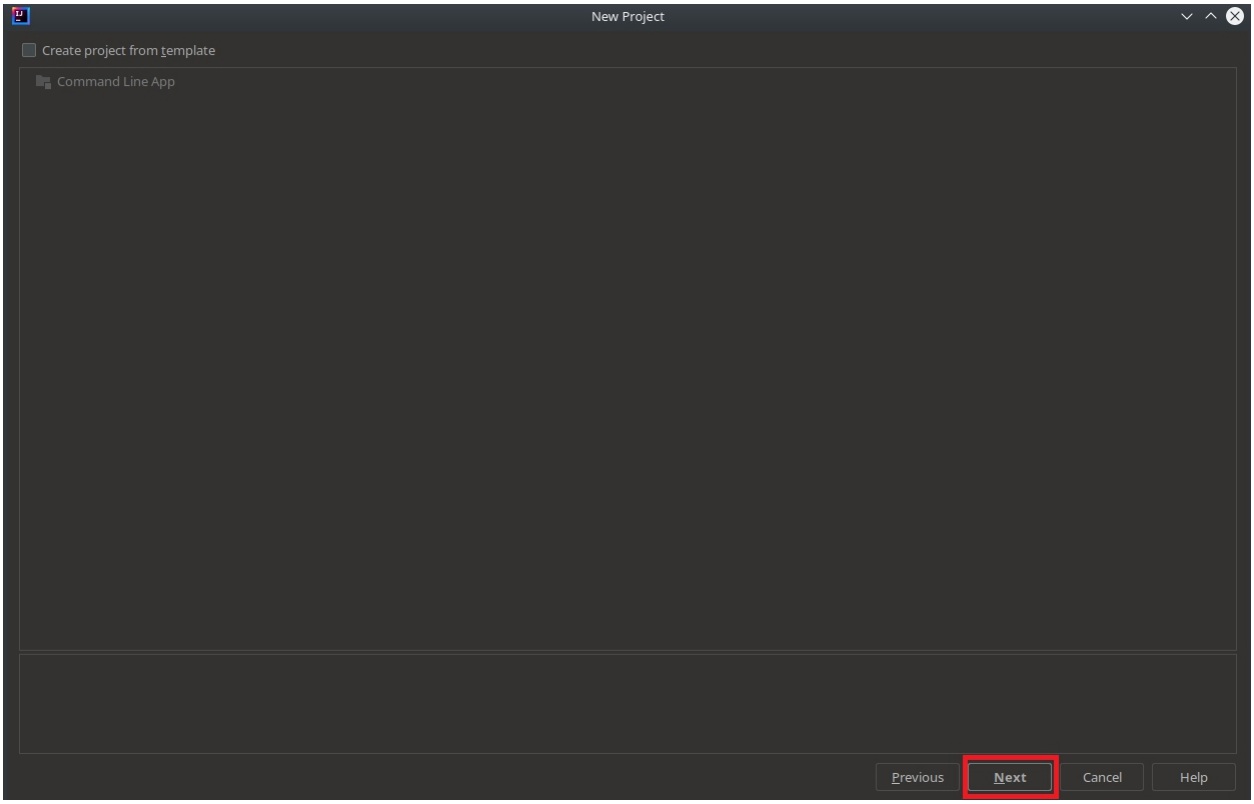


1.2. Construcția unui profil de build

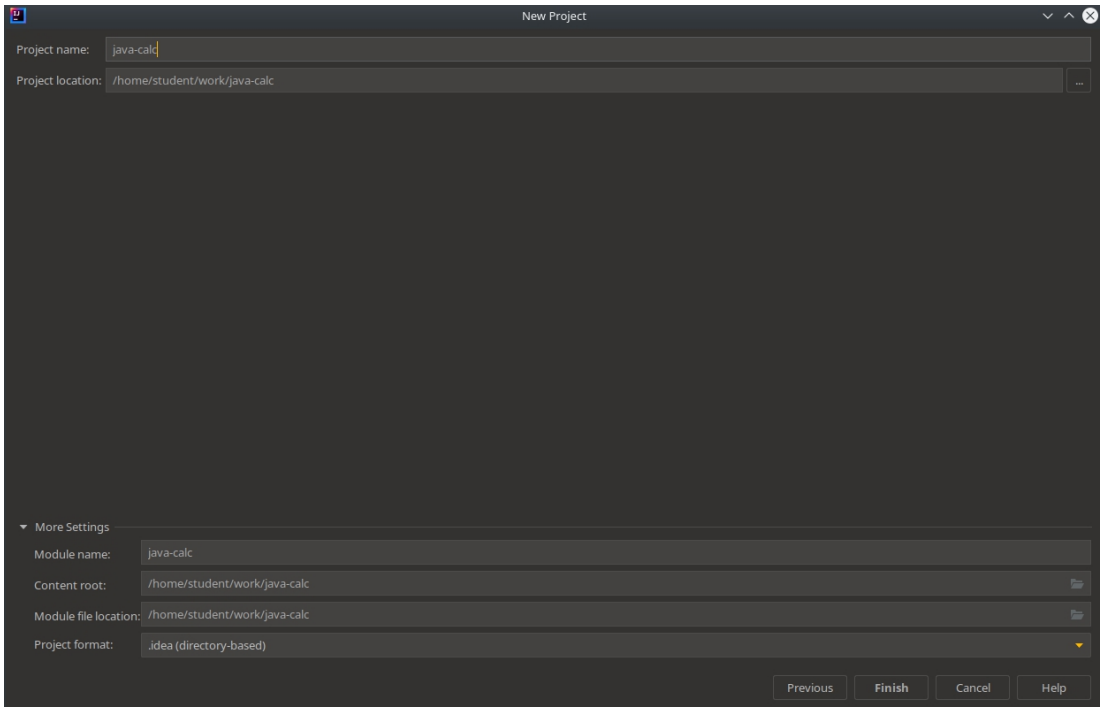
Selectăm Java în meniul din stânga și JDK 1.8 din dropdown-ul din dreapta.



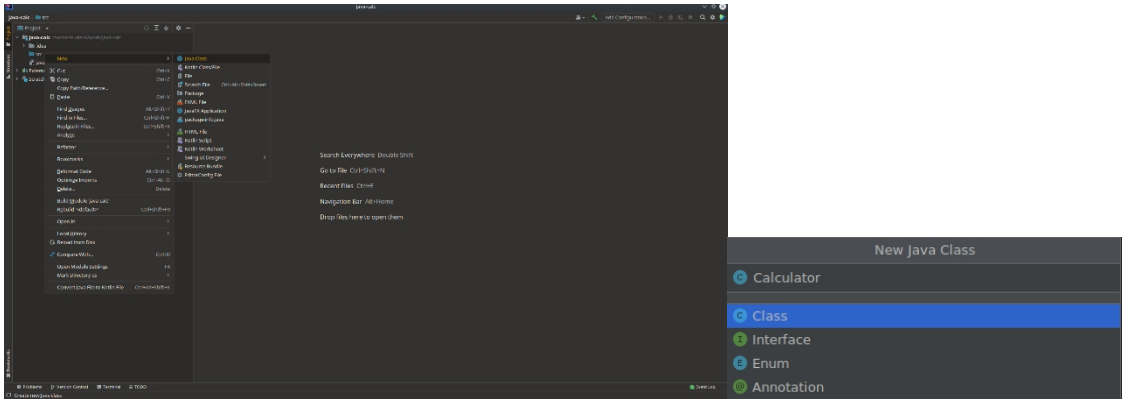
Apăsăm pe butonul **Next**



Denumim aplicația: ***java-calc***



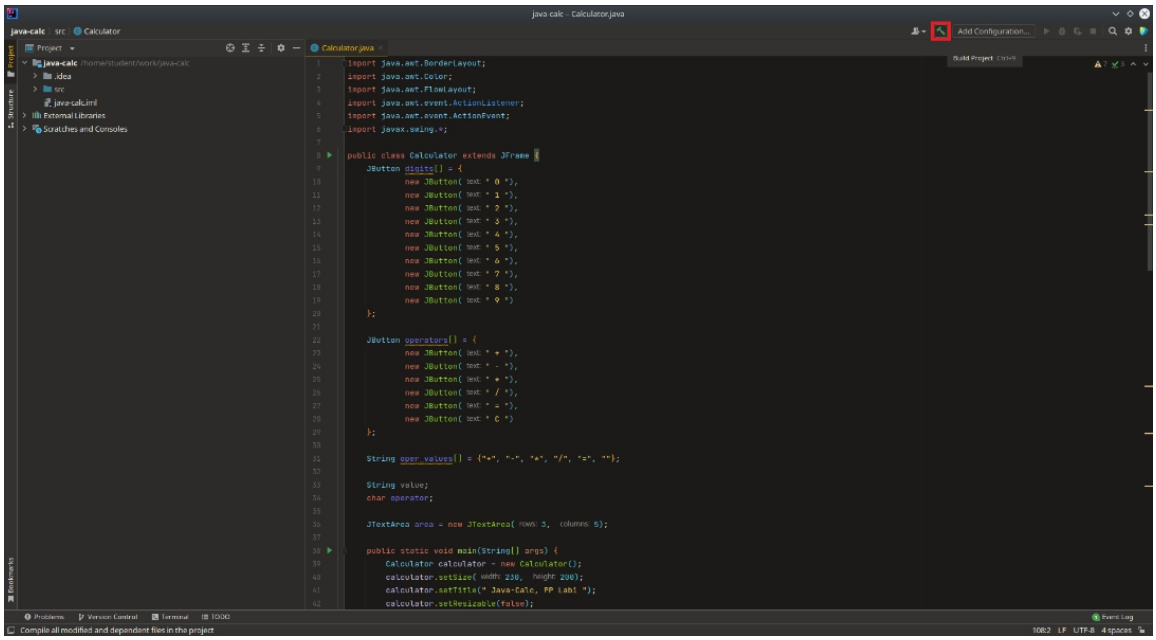
Adăugăm un fișier sursă nou proiectului:



Calculator.java

1.3. Build

Copy+paste codul din exemplul dat în laborator, apoi se face build apăsând simbolul verde din screenshot-ul de mai jos, sau în meniul de Build:

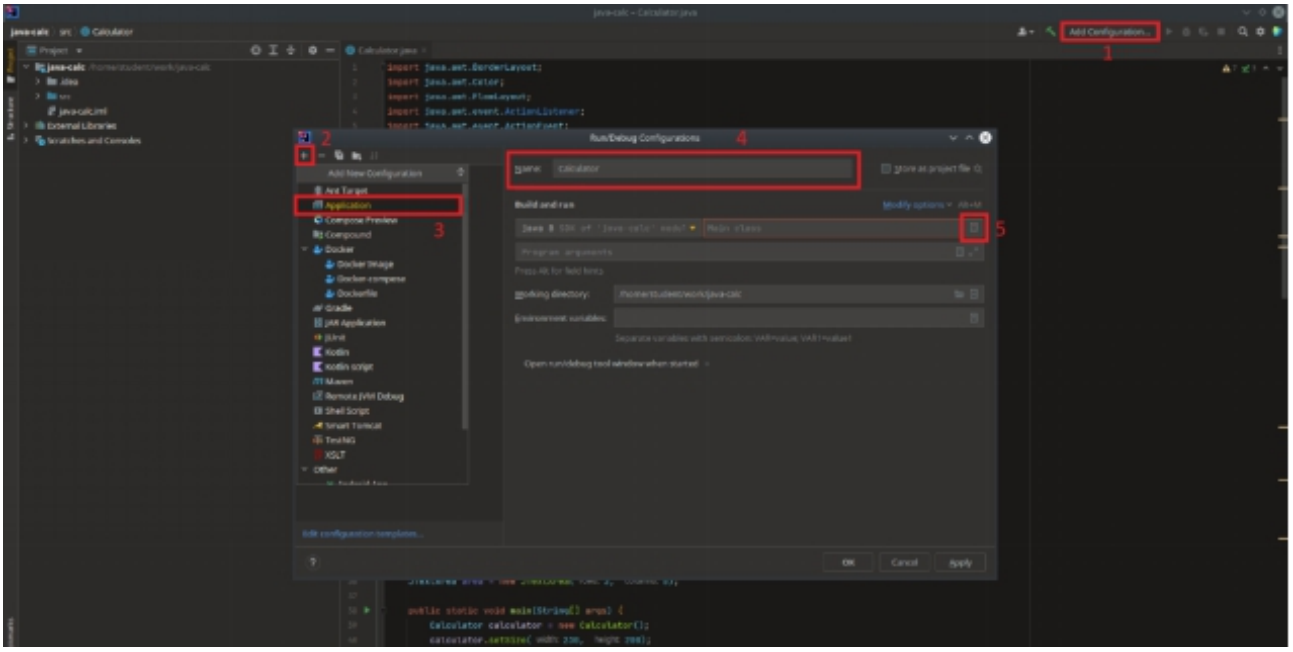


1.4. Adăugarea unui profil de execuție

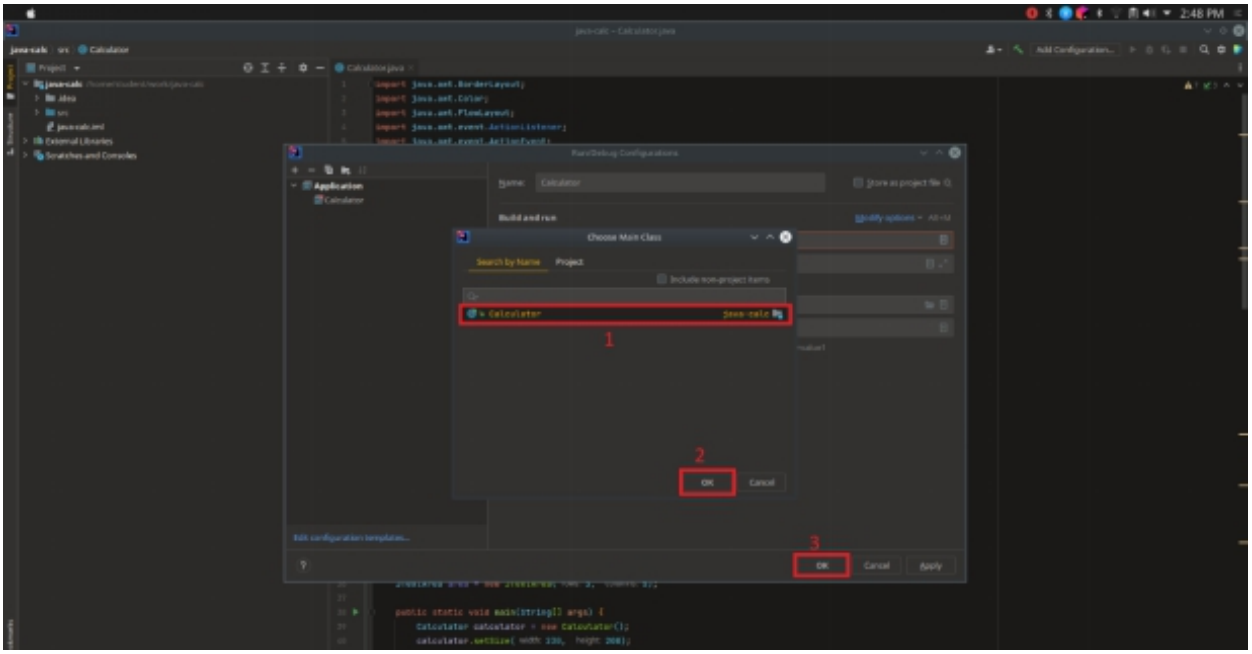
Pentru a executa aplicația, adăugăm o configurație de execuție. Acest pas este specific IntelliJ, deoarece este mai flexibil decât alte IDE-uri care vin preconfigurate pentru anumite tipuri de aplicații.

Aplicațiile JAVA pot avea și alte configurații de execuție, precum Kotlin, JAR, Applet, însă pentru acest exemplu vom alege Application.

În pasul 5 ilustrat în screenshot-ul de mai jos avem grijă să selectăm clasa care conține funcția main(). În pasul 6 se selectează versiunea corectă de JDK (1.8).

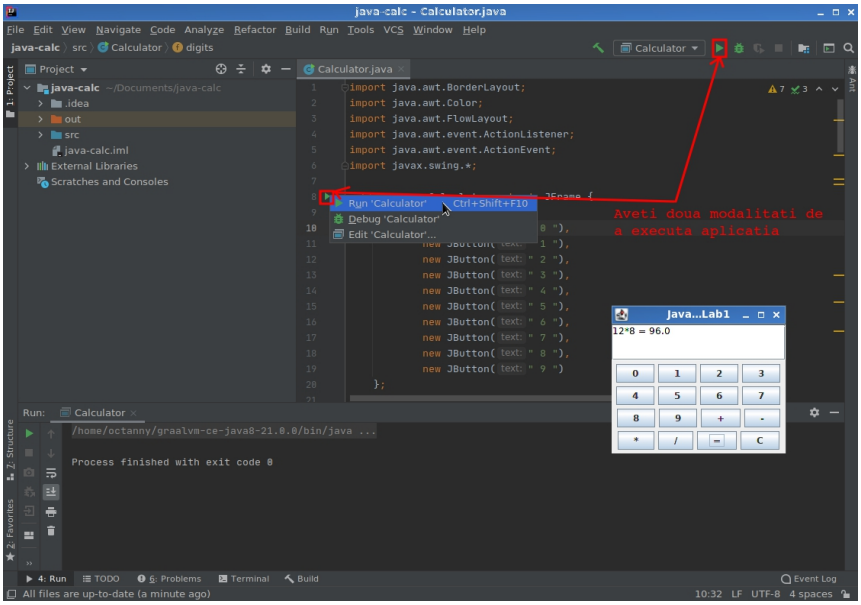


După pasul 5, va apărea urmatoatoarea fereastră:



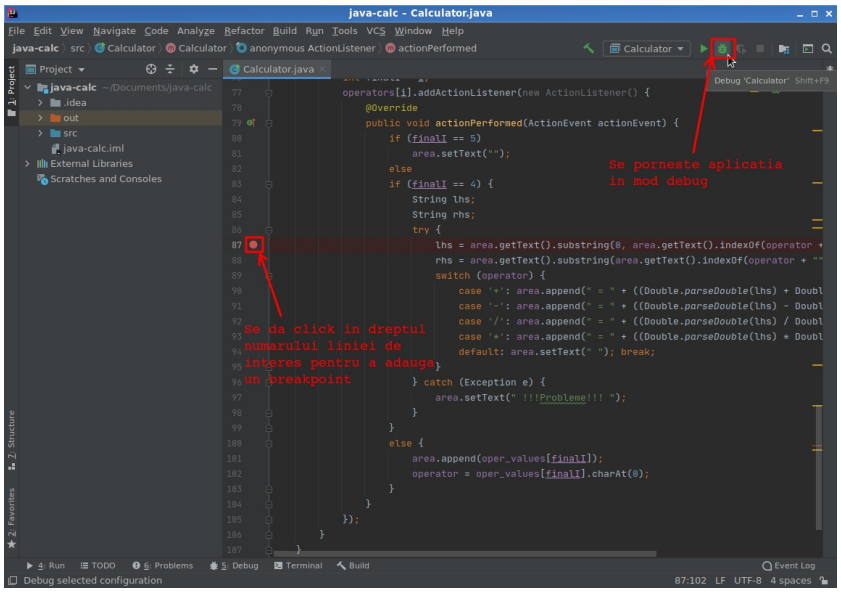
1.5. Execuţia

După crearea profilului de execuţie se aprinde butonul de execuţie (triunghiul verde) şi putem executa aplicaţia:

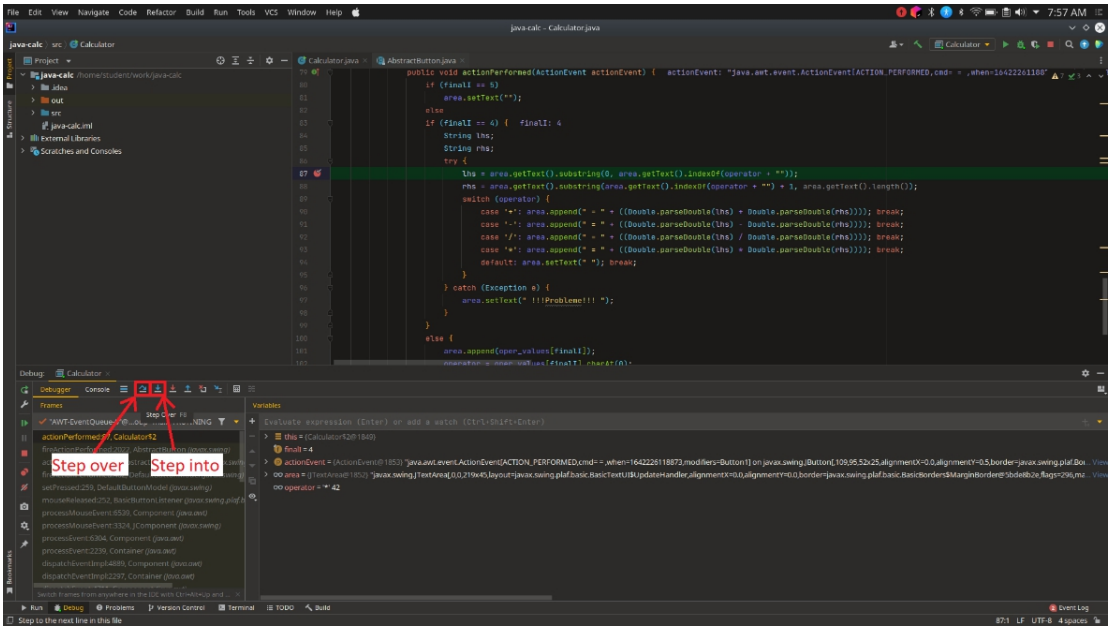


1.5.1. Cum facem debug?

Pe parcursul execuţiei, în partea stângă jos avem un panou care ne permite să facem debug, folosind puncte de oprire a execuţiei (engl. „breakpoints”) sau pas cu pas. Dat fiind că avem o aplicaţie care tratează evenimente, pentru a putea investiga ce se întâmplă în funcţia care face calculele atunci când apăsăm butonul „=”, va trebui să punem un breakpoint la linia 87.

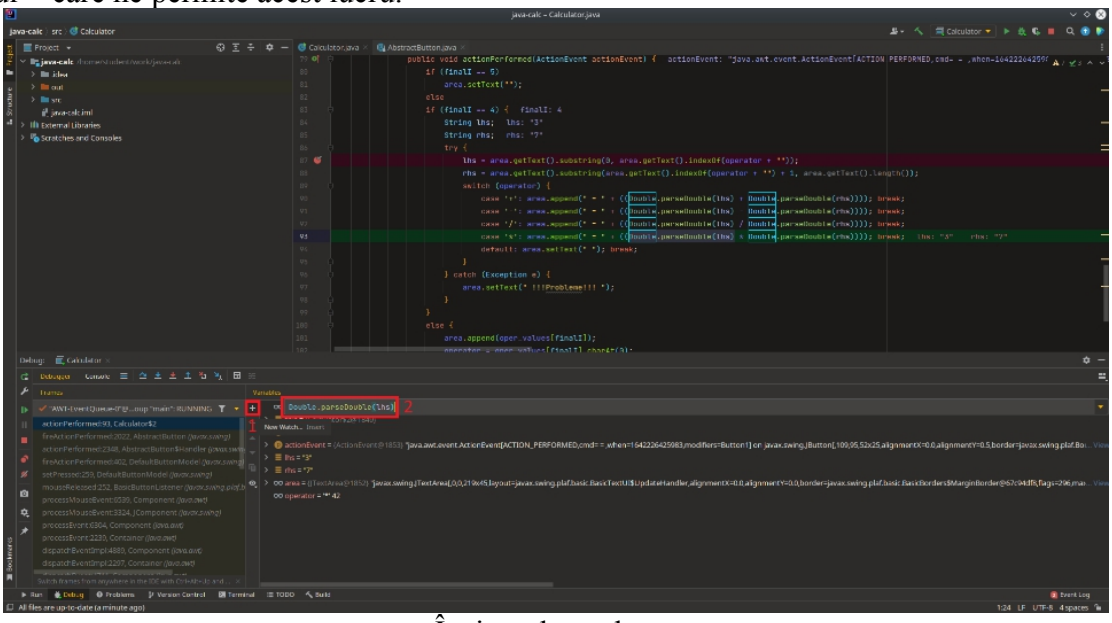


Construim calculul 3*7, iar atunci când apăsăm butonul „=”, execuţia ajunge la breakpoint. Pentru a continua procedura de debug, folosim F8=„step over” şi F7=„step into”. Diferenţa dintre cele două funcţii o face execuţia unei funcţii: F8 o va executa, F7 ne permite să intrăm şi să vedem ce se întâmplă în acea funcţie.

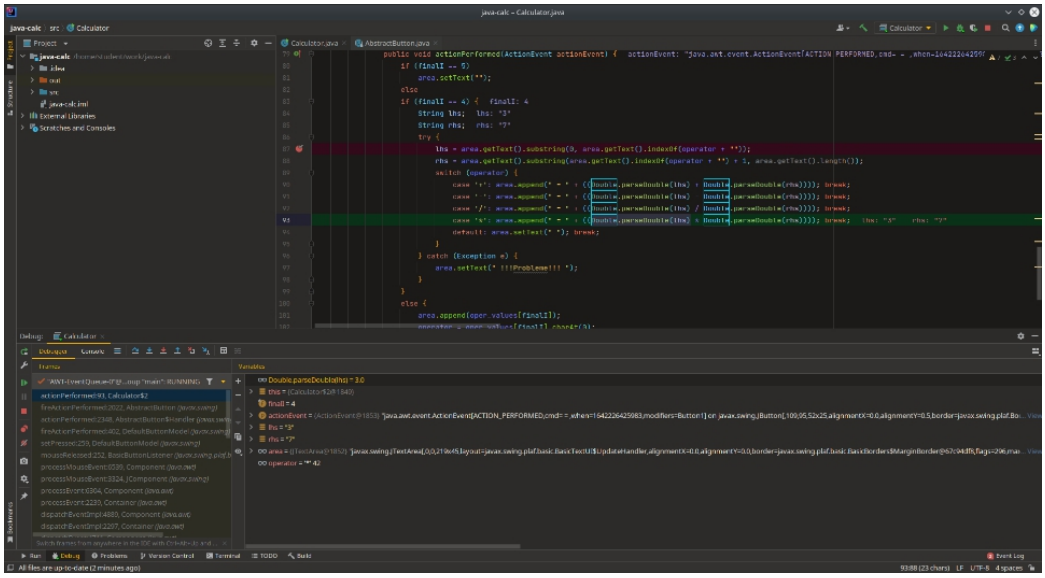


1.5.2. Cum urmărim variabilele?

În partea stângă jos sunt vizibile variabilele lhs=3, rhs=7 și operator=„*”. Switch-ul a urmat calea corectă către operația de înmulțire și ne așteptăm să obținem 21. În cazul în care se dorește adăugarea unor variabile suplimentare de urmărit, în partea stânga-sus a tab-ului cu variabile, avem simbolul + care ne permite acest lucru.



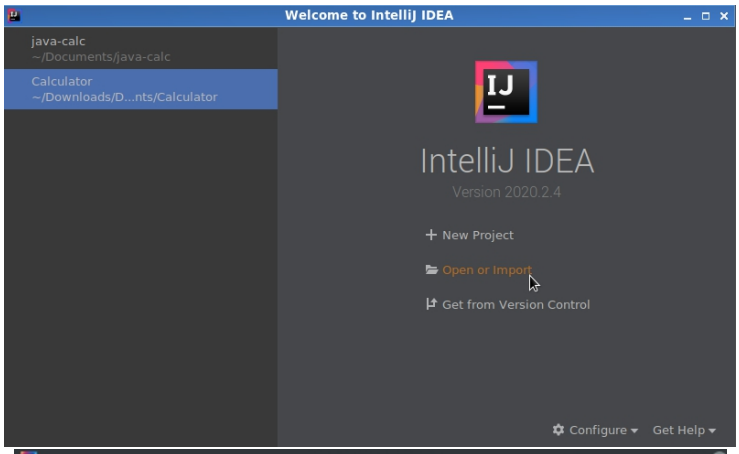
Înainte de evaluare

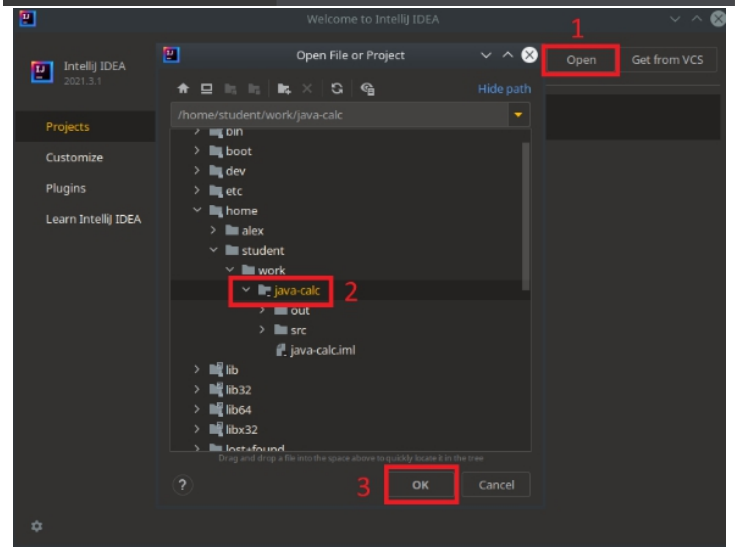


După evaluare

1.6. Deschiderea unui proiect existent

Dacă închidem proiectul, din File → Close Project, acesta se poate deschide din fereastra principală, ori din meniul din stânga, ori căutându-l pe disk apăsând pe opțiunea Open or Import.



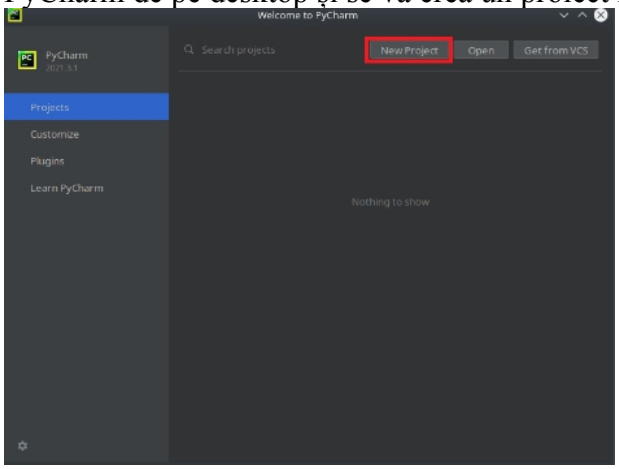


2. Crearea proiectelor Python utilizând PyCharm

În acest caz s-a folosit ca exemplu un proiect disponibil și aici.

2.1. Crearea unui proiect nou

Se va deschide IDE-ul PyCharm de pe desktop și se va crea un proiect nou (la fel ca la Java).



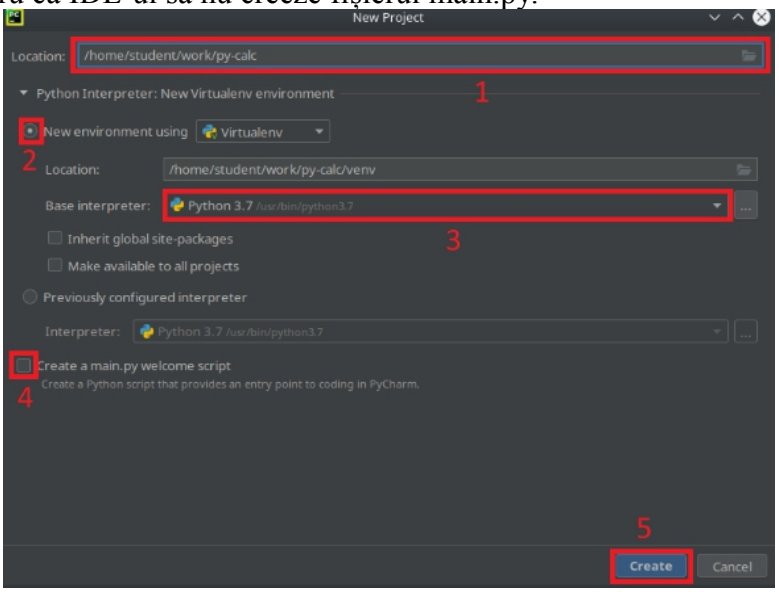
2.2. Construcția unui nou mediu virtual

1) Selectăm locația proiectului

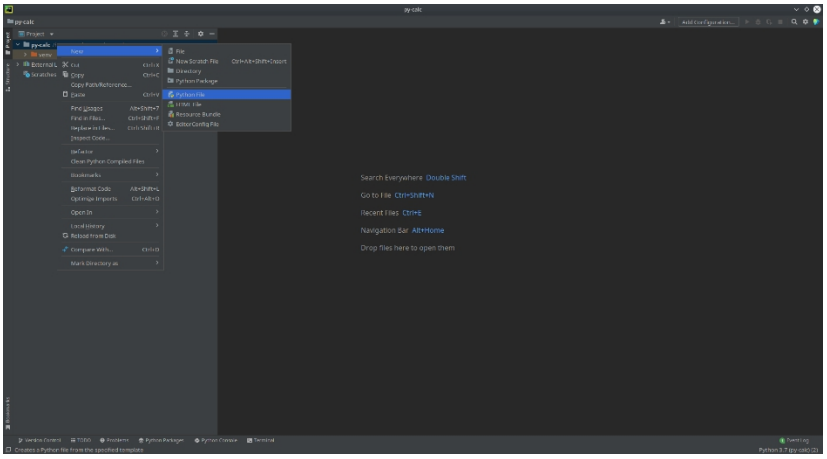
2) Construim un mediu virtual pentru utilizarea Python, în caz că vom utiliza pachete adiționale, nu afectăm distribuția principală de Python.

3) Selectăm Python 3.7

4) Debifăm pentru ca IDE-ul să nu creeze fișierul main.py.

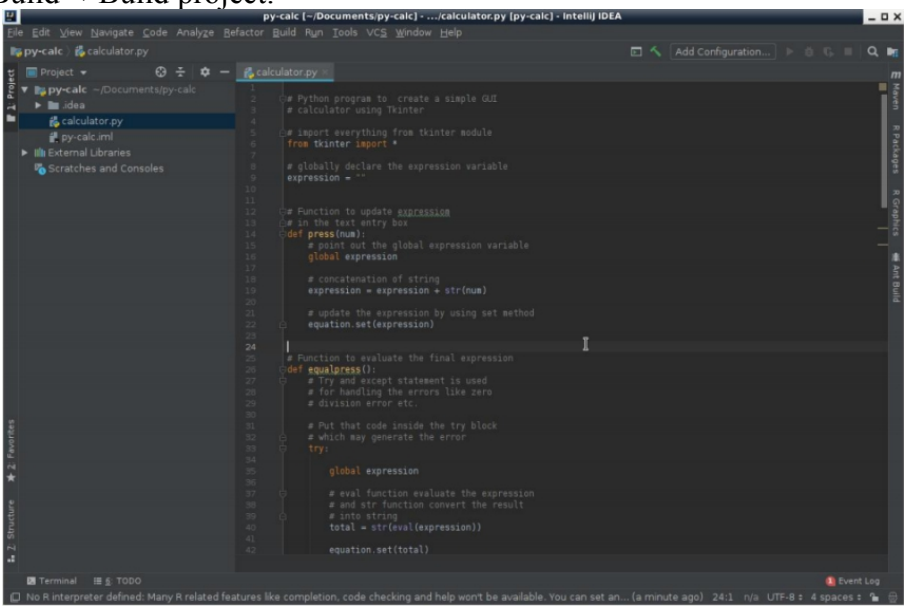


Adăugăm proiectului un fișier sursă cu click dreapta pe numele proiectului în tab-ul din stânga → New → Python File → calculator.py. Copy+paste codul din exemplul dat în laborator.



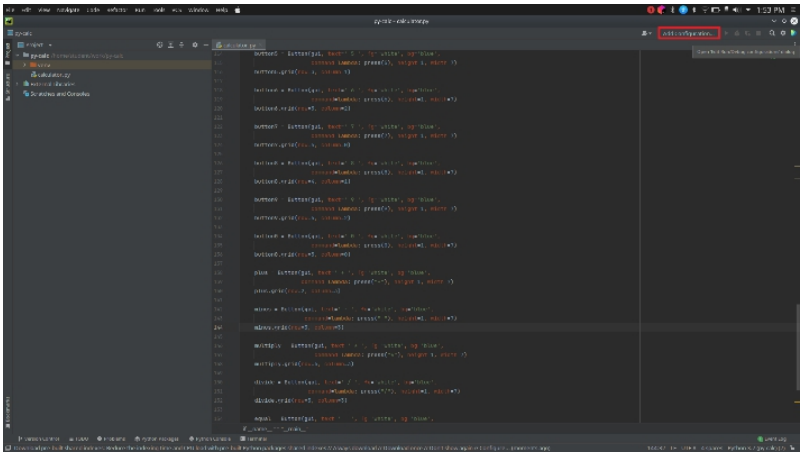
2.3. Build

Copy+paste la codul din exemplul dat în laborator, apoi se face build apăsând CTRL+F9, sau din meniul de Build → Build project:

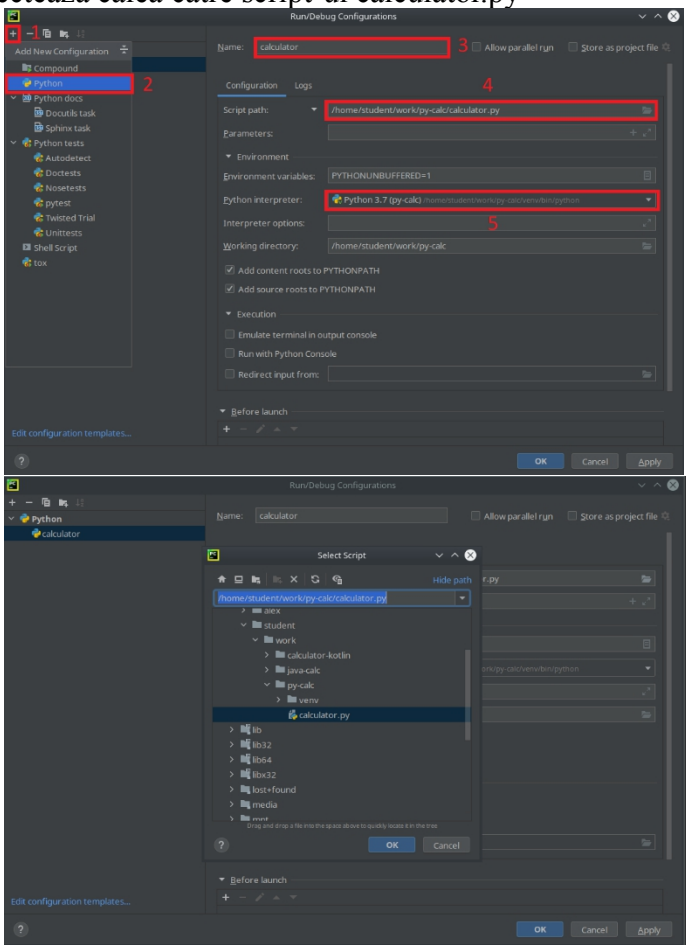


2.4. Adăugarea unui profil de execuție

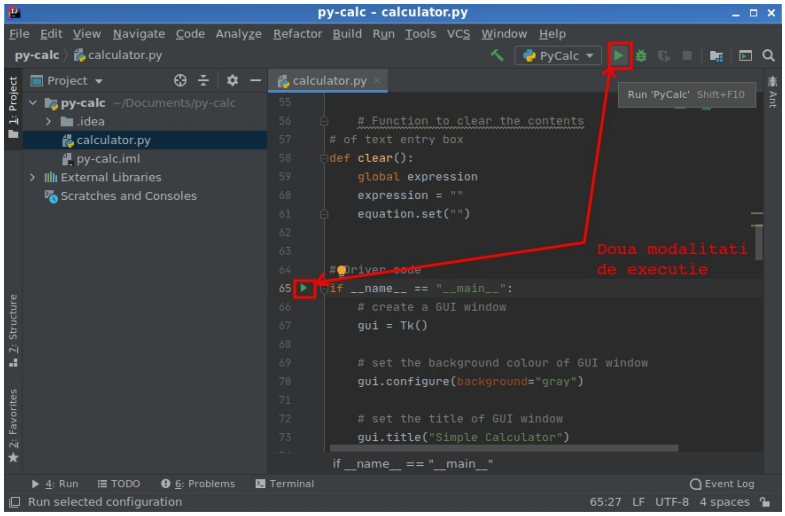
Se realizează apăsând butonul „Add Configuration”. Din pagina nouă selectăm + din stânga sus și apoi Python.



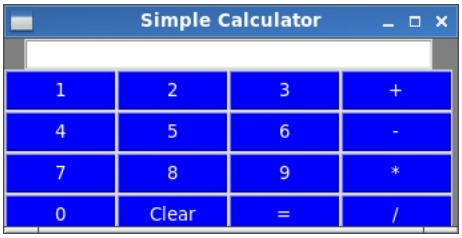
În pasul 4 se selectează calea către script-ul calculator.py



2.5. Execuția

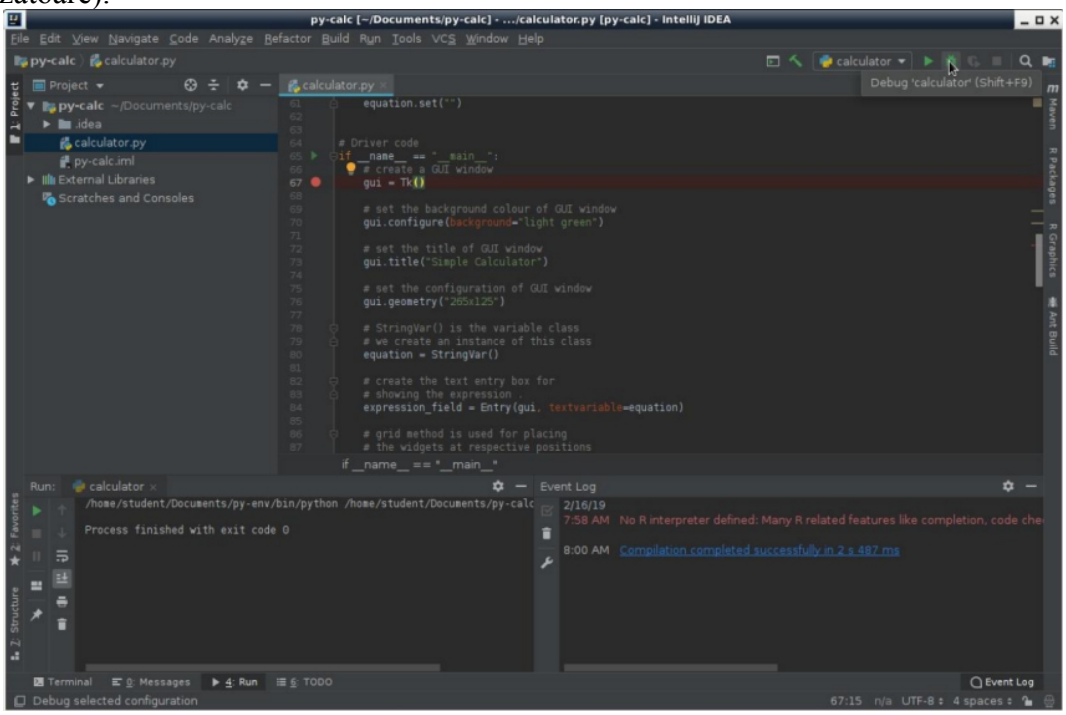


Rezultatul execuției:



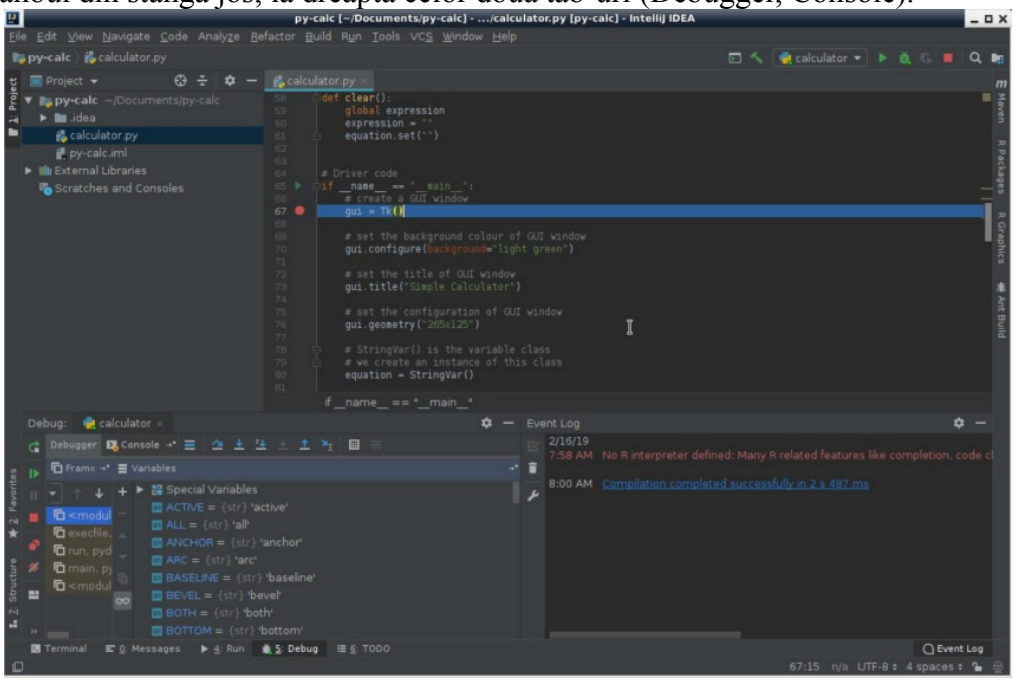
2.5.1. Cum facem debug?

În dreapta simbolului de execuție sau selectând din meniu Run → Debug „nume_proiect”, începem execuția în mod debug. În caz că dorim să accelerăm procesul și să investigăm anumite probleme, folosim breakpoints, ca execuția să se oprească aici (click lângă numărul liniei corespunzătoare).



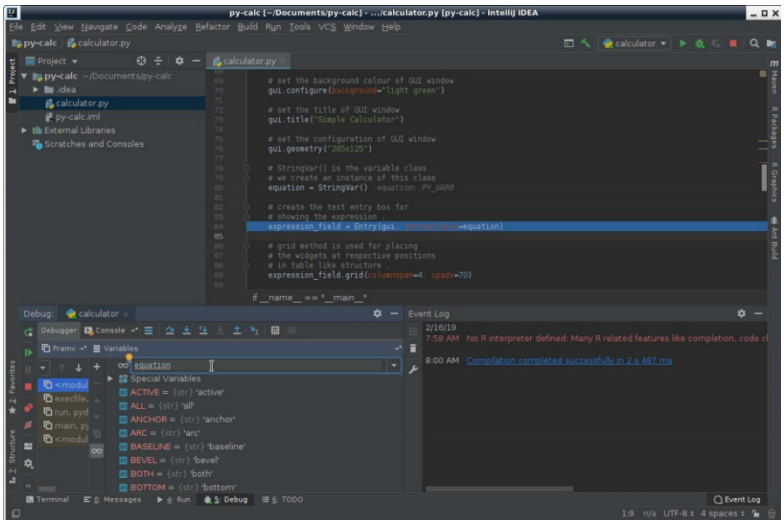
Fereastra din stânga jos ne permite să instrumentăm execuția, vizualizând valorile variabilelor în timp real. Pentru execuția pas cu pas a codului secvențial, vom folosi F8 (step over), iar pentru a

urmări execuția în interiorul unor funcții locale, folosim F7 (step in). Simbolurile corespunzătoare se găsesc în panoul din stânga jos, la dreapta celor două tab-uri (Debugger, Console).

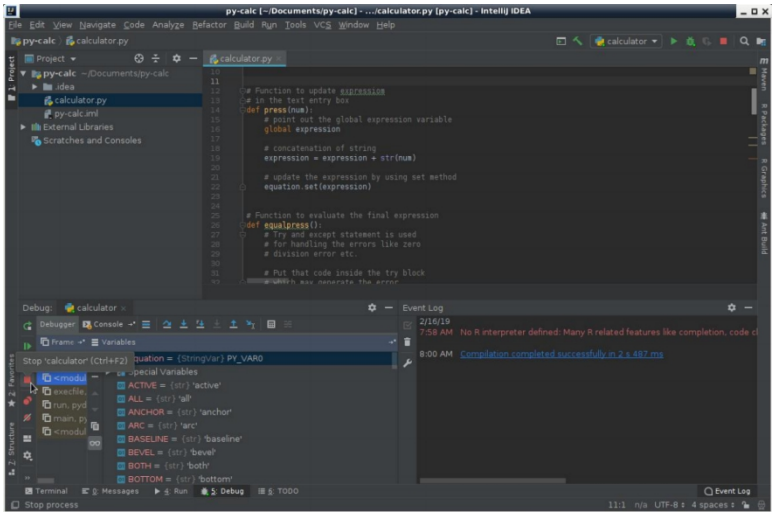


2.5.2. Cum urmărim variabilele?

Pentru a adăuga variabile care nu apar, folosim simbolul + de sub tab-ul Variables și introducem numele variabilei.



Pentru a opri execuția în mod Debug, folosim pătrățelul roșu din dreapta sus sau din stânga jos:



2.6. Deschiderea unui proiect existent

Vezi subcapitolul 1.6.

3. Crearea proiectelor Kotlin utilizând IntelliJ IDEA

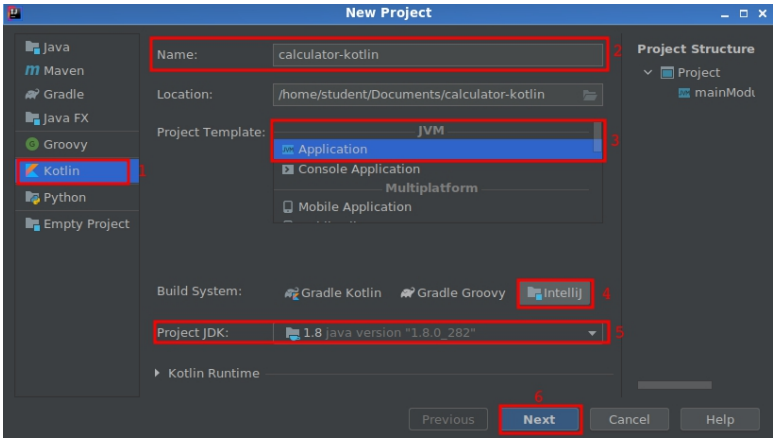
Proiectul exemplu este disponibil și https://www.programiz.com/kotlin-programming/examples/calculator-when.

3.1. Crearea unui proiect nou

Se va deschide IDE-ul IntelliJ IDEA de pe desktop și se va crea un proiect nou (la fel ca la Java).

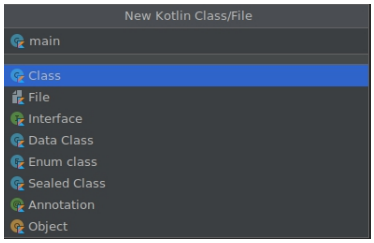
3.2. Construcția unui profil de build

Selectăm Kotlin → Kotlin/JVM



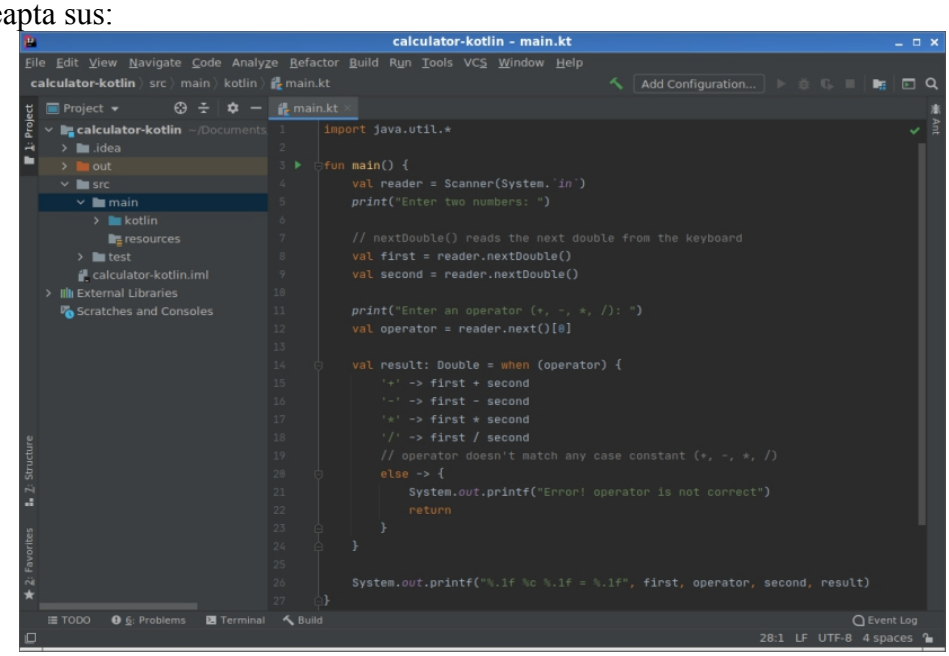
La final se apasă Next, iar în fereastra următoare Finish.

Adăugăm un fișier sursă cu click dreapta pe folder-ul kotlin (src/main/kotlin) în tab-ul proiectului din stânga, New → Kotlin File/Class → main. Fișierul main va avea pe disk extensia .kt



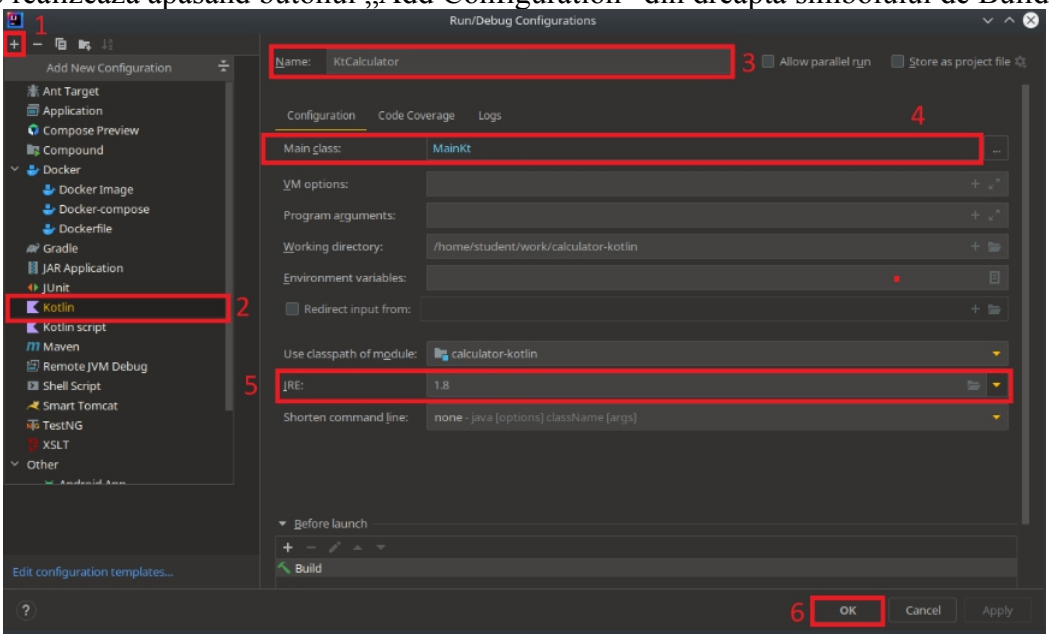
3.3. Build

Copy+paste codul din exemplul dat în laborator, apoi se face build cu click pe ciocănelul verde din dreapta sus:



3.4. Adăugarea unui profil de execuție

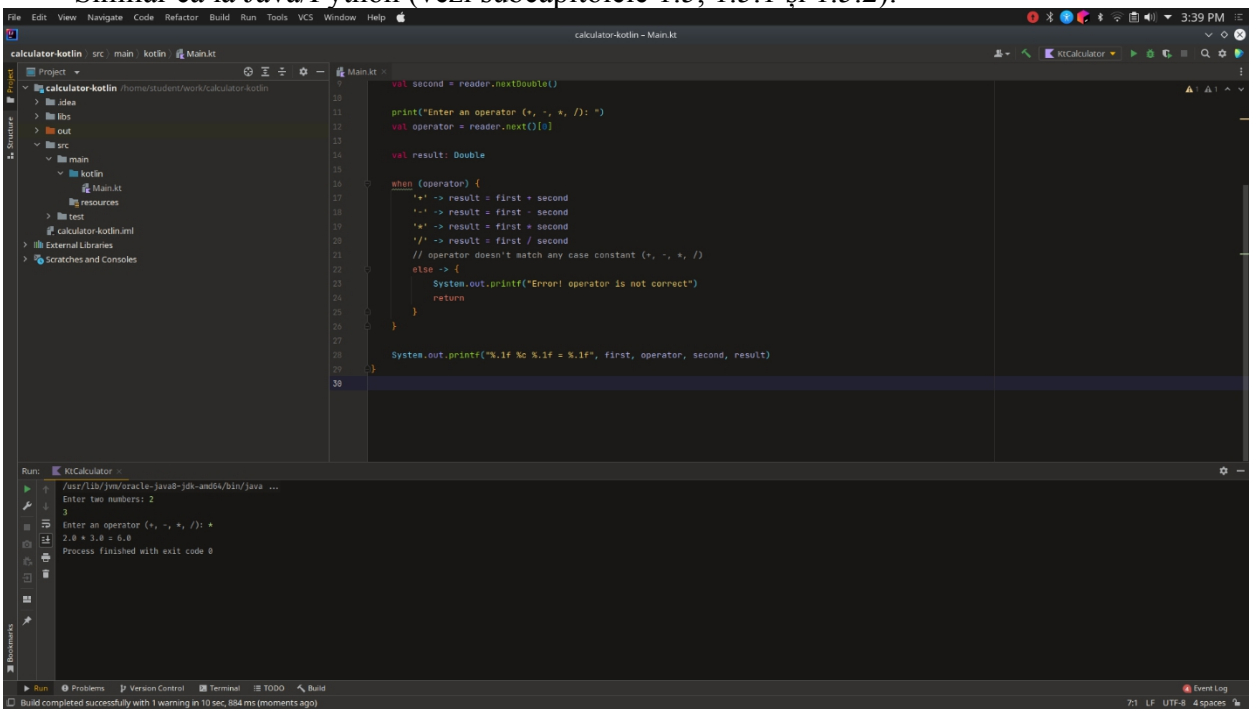
Se realizează apăsând butonul „Add Configuration” din dreapta simbolului de Build.



Deși în cazul de față nu folosim o clasă, în pasul 4, valoarea MainKt pentru Main class referă fișierul sursă cu funcția main().

3.5. Execuția

Similar ca la Java/Python (vezi subcapitolele 1.5, 1.5.1 și 1.5.2).



3.6. Deschiderea unui proiect existent

Vezi subcapitolul 1.6.

4. Crearea proiectelor Kotlin-JS utilizând IntelliJ IDEA

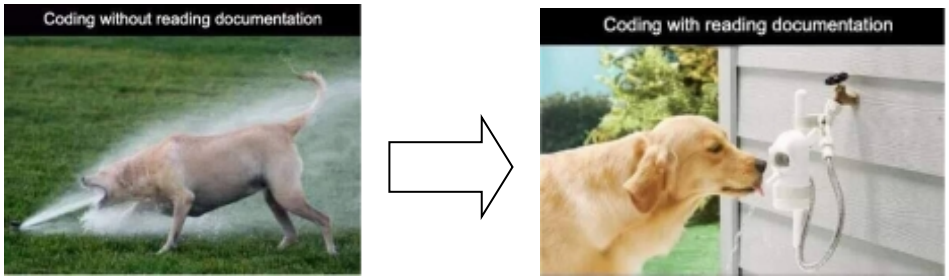
Se va urmări tutorialul disponibil la https://www.raywenderlich.com/201669-web-app-with-kotlin-js-getting-started. Pașii pentru Build / Adăugarea configurației de execuție / Debug, corespund cu cei prezentați pentru proiectele anterioare.

5. Controlul versiunilor

Se recomandă citirea cărții **„Pro Git”** scrisă de Scott Chacon și Ben Straub. Aceasta este disponibilă pe https://git-scm.com/book/en/v2

Alte resurse recomandate:

1. https://guides.github.com/introduction/flow/
2. https://nvie.com/posts/a-successful-git-branching-model/

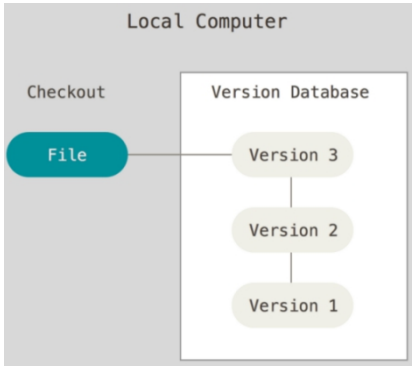


Controlul versiunilor este un sistem ce înregistrează schimbările unui fișier sau a unui set de fișiere de-a lungul timpului, pentru a putea reveni la versiuni specifice mai târziu. Cu alte cuvinte, în

cazul în care se dorește „Undo” la un anumit fișier, se poate reveni la o versiune anterioară (cu precizarea că trebuie făcut **commit**, ideal după orice funcție/funcționalitate scrisă).

5.1. Sisteme de locale pentru gestiunea versiunilor codului unei aplicații

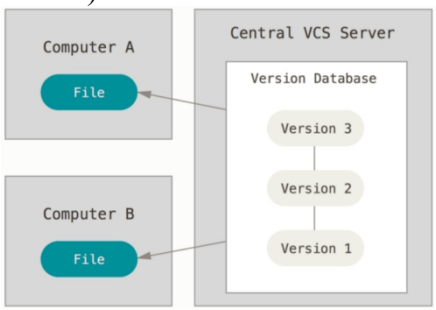
Inițial înainte de a face modificări la un proiect în desfășurare acesta se salva separat de multe ori ca o simplă arhivă (opțional cu data curentă). Această metodă e predispusă la erori, deoarece este ușor să uiți în ce folder te afli și poți modifica alt set de fișiere. Pentru a ușura munca dezvoltatorului și a scădea rata de erori nedorite s-au dezvoltat așa zisele Sisteme de control al versiunilor care sunt bazate în principal pe existența unei baze de date simplă care „reține” toate modificările pe setul de fișiere.



Sistem de versionare local

5.2. Sisteme centralizate pentru gestiunea versiunilor codului unei aplicații (de versionare)

Primele sisteme de gestiune a versiunilor erau doar la nivel local ceea ce punea probleme în cazul dezvoltării simultane bazate pe mai multe echipe care de multe ori nici nu erau aflate în aceeași locație fizică. Pentru a rezolva problema s-au introdus sistemele centralizate pentru controlul versiunilor. Acestea erau bazate pe un server central care conține toate fișierele aparținând unei versiuni a codului sursă (versionate) și un număr de clienți care copiază versiunea curentă a fișierelor respective (aplicație client server clasică).

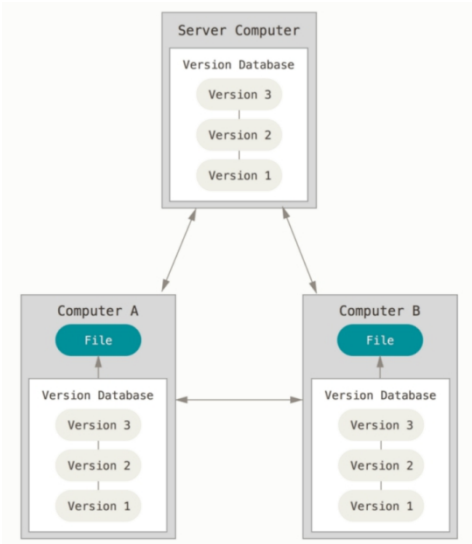


Sistem de versionare centralizat

5.3. Sisteme de versionare distribuite

Orice sistem centralizat are o serie de dezavantaje printre care și faptul că dacă serverul se blochează atunci toții clienții nu mai pot funcționa corect („single point of failure”) De asemenea, dacă hard disk-ul pe care se află repository-ul central devine corupt, se pierde tot istoricul proiectului. Acest lucru este valabil și la sistemele de versionare locale.

Pentru a rezolva problema s-au dezvoltat sistemele de versionare distribuite (**Git**, Mercurial, etc). Într-un astfel de sistem, clienții nu salvează doar ultima versiune a fișierelor, ci copiază întregul repository (inclusiv istoricul complet). Așadar, dacă vreun server „moare”, orice repository al unui client poate fi copiat înapoi pe acel server pentru a-l restaura.

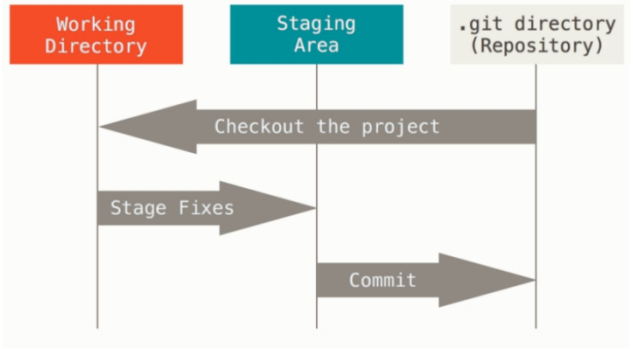


Sistem de versionare distribuit

5.4. Git

În cele ce urmează, se va utiliza Git ca sistem de versionare distribuit. Fișierele dintr-un repository git pot avea una dintre următoarele trei stări:

- **Modified -** s-au efectuat modificări asupra fișierului dar nu s-a făcut încă commit în baza de date
- **Staged -** s-a marcat un fișier modificat în versiunea lui curentă pentru a fi inclus în următorul commit
- **Committed -** datele sunt stocate în siguranță în baza de date locală



Secțiunile principale ale unui proiect Git

5.4.1. Instalare și configurare

Pentru instalare, se deschide un terminal și se execută comanda:

```
sudo apt install git
```

Pentru configurare, se vor executa următoarele comenzi:

```
git config --global user.name "Ion Popescu"
git config --global user.email "ion.popescu@gmail.com"
git config --global core.editor vim
git config --global core.pager more
git config --global help.autocorrect true
```

Se poate modifica editorul de commit cu editorul preferat, dar se recomandă utilizarea unui editor din linia de comandă.

5.5. Introducere în Git

5.5.1. Integrare Git în IntelliJ

După ce s-a instalat Git din terminal, se pornește IntelliJ, se deschide fereastra de Setări/Preferințe (CTRL+ALT+S) și se selectează **Version Control -> Git.** La „Path to Git executable“ se adaugă:

```
/usr/bin/git
```

Observație: Se poate testa dacă calea este corectă prin apăsarea butonului „Test”. Dacă nu coincide, se va deschide un terminal și se va executa comanda:

```
which git
```

Opțional: Dacă se dorește configurarea Git remote, se poate opta pentru stocarea parolei în meniul **Appearance and Behavior -> System Settings -> Passwords.**

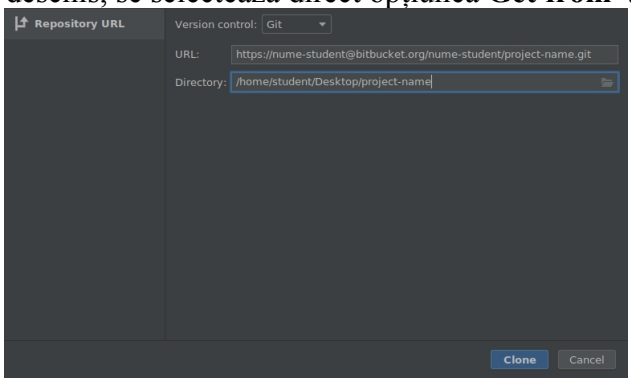
Pentru mai multe detalii cu privire la integrarea Git în IntelliJ, vezi https://www.jetbrains.com/help/idea/version-control-integration.html.

5.5.2. Utilizarea Git în IntelliJ

Există mai multe modalități de a lucra cu Git. Se poate crea un repository local, un repository pe un server (personal) remote, sau se poate opta pentru un serviciu de găzduire (hosting) cum ar fi BitBucket (https://bitbucket.org/account/signup/).

5.5.3. Clonarea unui proiect de pe un server remote

Se deschide IntelliJ și se selectează meniul **VCS -> Get from Version Control**, sau dacă nu este nici un proiect deschis, se selectează direct opțiunea **Get from Version Control.**



Clonare repository din IntelliJ

Dacă apare o fereastră „Add File to Git”, se bifează „Remember, don’t ask again” și se apasă pe butonul **Cancel.**

5.5.4. Git din linia de comandă

Vezi cheat-sheet-ul anexat lucrării de laborator.

5.6. Instalare și configurare Docker pentru Git

Mai întâi se dezinstalează versiunile vechi de docker / docker-engine:

```
sudo apt-get -y remove docker docker-engine docker.io
```

Apoi, se instalează dependențele:

```
sudo apt update
sudo apt install -y apt-transport-https ca-certificates wget
sudo apt install -y software-properties-common ssh
```

Se adaugă cheia GPG pentru repository-ul docker pe sistemul propriu:

```
wget https://download.docker.com/linux/debian/gpg
sudo apt-key add gpg
```

Se adaugă repository-ul Docker oficial, executând comanda (e o singură linie):

```
echo "deb [arch=amd64] https://download.docker.com/linux/debian
$(lsb_release -cs) stable" | sudo tee -a
/etc/apt/sources.list.d/docker.list
```

Observație: comanda de mai sus e pe o singură linie în terminal.

```
sudo apt update
sudo apt-cache policy docker-ce
sudo apt -y install docker-ce
```

Pentru a porni docker, se execută următoarele comenzi:

```
sudo systemctl start docker  # to start Docker
sudo systemctl enable docker # enable Docker service to autostart
```

Verificare instalare corectă:

```
sudo docker run hello-world
```

Permiterea utilizatorilor non-root să folosească Docker:

```
sudo groupadd docker
sudo usermod -aG docker <your_username>
```

Dintr-un terminal deschis pentru user-ul configurat, se execută:

```
docker run hello-world  # fara sudo
```

În cele ce urmează se va utiliza o imagine Git Server Docker cu Alpine Linux. Pentru aceasta, se va executa în terminal:

```
docker run -d -p 2222:22 -v ~/git-server/keys:/git-server/keys -v ~/git-
server/repos:/git-server/repos jkarlos/git-server-docker
```

Observație: comanda de mai sus e pe o singură linie în terminal.

În urma executării comenzii anterioare, s-au creat directoarele *~/git-server/keys* și *~/git-server/repos*. Este posibil ca directoarele *keys* și *repos* să aparțină utilizatorului root, caz în care se execută următoarele două comenzi:

```
sudo chown -R <your_username>:<your_username> ~/git-server/keys
sudo chown -R <your_username>:<your_username> ~/git-server/repos
```

Se generează o cheie ssh:

```
ssh-keygen -b 4096
```

La *„Enter file in which to save the key (/home/<your_username>/.ssh/id_rsa):“* se apasă tasta ENTER.

La *„Enter passphrase (empty for no passphrase):”* și *„Enter same passphrase again:”* se lasă liber, apăsând tasta ENTER pentru a continua.

Se copiază cheia publică în directorul ~/git-server/keys:

```
cp ~/.ssh/id_rsa.pub ~/git-server/keys
```

Acum este nevoie să fie repornit containerul docker. Pentru aceasta, se execută întâi comanda:

```
docker ps
```

Se copiază CONTAINER ID-ul, apoi se execută comanda:

```
docker restart <container-id>
```

Spre exemplu:

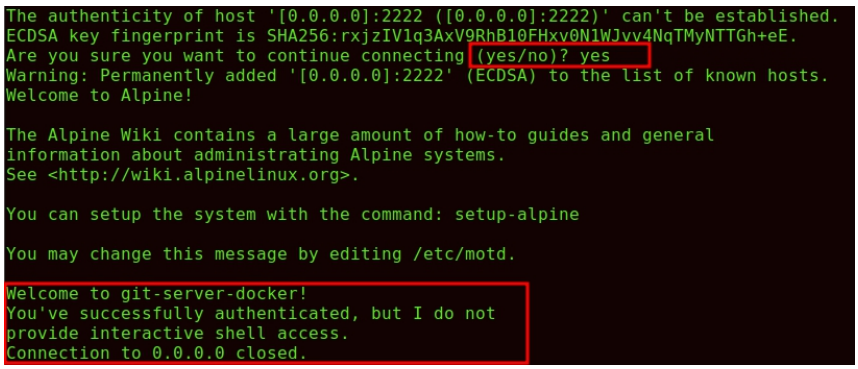
```
docker restart 25860a4feabb
```

Pentru a testa funcționarea corectă a containerului, se copiază adresa IP din coloana PORTS afișată de comanda **docker ps** și se execută:

```
ssh git@<ip-docker-server> -p 2222
```

Spre exemplu:

```
ssh git@0.0.0.0 -p 2222
```



Creare repository nou:

```
mkdir project_name
cd project_name
git init --shared=true
echo "env/" > .gitignore
git add .
git commit -m "Initial commit"
cd ..
git clone --bare project_name ~/git-server/repos/project_name.git
```

Pentru clonarea repository-ului:

```
git clone ssh://git@<ip-docker-server>:2222/git-
server/repos/project_name.git
```

Spre exemplu:

```
git clone ssh://git@0.0.0.0:2222/git-server/repos/project_name.git
```

După ce se efectuează alte modificări și commit-uri, se poate face direct push (URL-ul de remote origin fiind deja setat:

```
git push
```

Pentru mai multe detalii privind comenzile Git din terminal, vezi anexa de la finalul laboratorului.

Se recomandă utilizarea unui server personal remote, sau configurarea unei mașini virtuale care să se comporte ca un server remote. Detaliile de configurare sunt disponibile la adresa: https://git-scm.com/book/en/v2/Git-on-the-Server-Setting-Up-the-Server

Aplicaţii şi teme

Aplicaţii de laborator:

1. Pentru acomodarea cu IDE-ul IntelliJ, parcurgeți primele 4 capitole pas cu pas.
2. Creați un repository Git local și adăugați în acesta unul dintre exemplele prezentate.

Teme pe acasă:

1. Alegeți unul dintre proiecte (deci cu un limbaj anume) și extindeți calculatorul astfel încât acesta să poată analiza și expresii matematice compuse (de exemplu: 2*(4-5/6)/456). Pentru aceasta se va utiliza **evaluarea poloneză explicit implementată (nu versiunea cu stive simulate din vectori)** . Se poate opta și pentru implementarea cu arbori.
2. Să se inițializeze un repository git într-un container docker și să se implementeze o aplicație (într-un limbaj la alegere) care să citească un fișier text, să proceseze conținutul astfel încât să elimine semnele de punctuație, să execute alte două procesări la alegere (eliminare spații multiple, lower/upper case, filtrare cuvinte cu un anumit număr de litere, filtrare numere, etc) și să afișeze rezultatul la consolă. Pentru fiecare funcționalitate nou adăugată (citirea unui fișier, procesarea conținutului, afișarea rezultatului, etc), se execută git add, git commit și git push. Se prezintă pe laptop sau on remote.

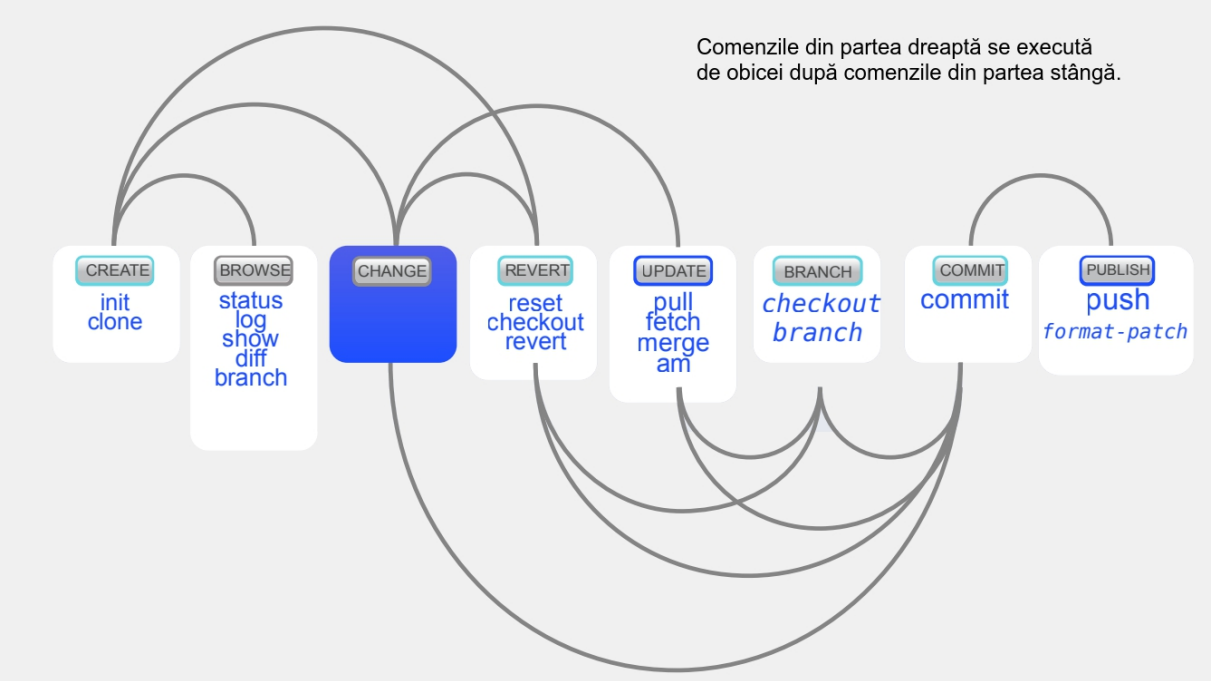
Anexe

Cele mai bune practici pentru sistemele de versionare

- **Commit related changes**: Commit-uri cât mai mici (rezolvarea a două erori diferite = două commit-uri diferite)
- **Commit often:** Commit-uri cât mai dese
- **Do NOT commit half-done work:** Nu se face commit dacă spre exemplu te-ai oprit din a lucra la proiect la jumătatea unei funcții
- **Test code before you commit**: Nu se face commit la o porțiune de cod netestată (alți developeri pot face pull și să sesizeze unele erori)

- **Write good commit messages:** Se evită mesajele care nu explică exact ce s-a întâmplat în acel commit. Mesajele de commit trebuie să fie cât mai descriptive și preferabil scurte.
- **Use branches**: Când se lucrează la un feature nou, acesta trebuie creat pe o ramură (branch) separată de cea principală, fiind integrată la final în branch-ul master.

Git Cheat Sheet



Flow-ul comenzilor Git

SETUP

Configuring user information used across all local repositories

`git config --global user.name "[firstname lastname]"`

set a name that is identifiable for credit when review version history

`git config --global user.email "[valid-email]"`

set an email address that will be associated with each history marker

`git config --global color.ui auto`

set automatic command line coloring for Git for easy reviewing

SETUP & INIT

Configuring user information, initializing and cloning repositories

`git init`

initialize an existing directory as a Git repository

`git clone [url]`

retrieve an entire repository from a hosted location via URL

STAGE & SNAPSHOT

Working with snapshots and the Git staging area

`git status`

show modified files in working directory, staged for your next commit

`git add [file]`

add a file as it looks now to your next commit (stage)

`git reset [file]`

unstage a file while retaining the changes in working directory

`git diff`

diff of what is changed but not staged

`git diff --staged`

diff of what is staged but not yet committed

`git commit -m "[descriptive message]"`

commit your staged content as a new commit snapshot

IGNORING PATTERNS

Preventing unintentional staging or commiting of files

```
logs/
*.notes
pattern*/
```

Save a file with desired patterns as .gitignore with either direct string matches or wildcard globs.

`git config --global core.excludesfile [file]`

system wide ignore pattern for all local repositories

SHARE & UPDATE

Retrieving updates from another repository and updating local repos

`git remote add [alias] [url]`

add a git URL as an alias

`git fetch [alias]`

fetch down all the branches from that Git remote

`git merge [alias]/[branch]`

merge a remote branch into your current branch to bring it up to date

`git push [alias] [branch]`

Transmit local branch commits to the remote repository branch

`git pull`

fetch and merge any commits from the tracking remote branch

BRANCH & MERGE

Isolating work in branches, changing context, and integrating changes

`git branch`

list your branches. a * will appear next to the currently active branch

`git branch [branch-name]`

create a new branch at the current commit

`git checkout`

switch to another branch and check it out into your working directory

`git merge [branch]`

merge the specified branch's history into the current one

`git log`

show all commits in the current branch's history

INSPECT & COMPARE

Examining logs, diffs and object information

`git log`

show the commit history for the currently active branch

`git log branchB..branchA`

show the commits on branchA that are not on branchB

`git log --follow [file]`

show the commits that changed file, even across renames

`git diff branchB...branchA`

show the diff of what is in branchA that is not in branchB

`git show [SHA]`

show any object in Git in human-readable format

TRACKING PATH CHANGES

Versioning file removes and path changes

`git rm [file]`

delete the file from project and stage the removal for commit

`git mv [existing-path] [new-path]`

change an existing file path and stage the move

`git log --stat -M`

show all commit logs with indication of any paths that moved

REWRITE HISTORY

Rewriting branches, updating commits and clearing history

`git rebase [branch]`

apply any commits of current branch ahead of specified one

`git reset --hard [commit]`

clear staging area, rewrite working tree from specified commit

TEMPORARY COMMITS

Temporarily store modified, tracked files in order to change branches

`git stash`

Save modified and staged changes

`git stash list`

list stack-order of stashed file changes

`git stash pop`

write working from top of stash stack

`git stash drop`

discard the changes from top of stash stack