Software Technology Lab

GEIAL316-B2

Lab.

Tamas Tompa, PhD
assistant professor
Department of Information Technology

Miskolc, 2026

1.

What will be needed?

- Any development environment
  - Eclipse / IntelliJ (preferred)
- **JUnit** framework (*junit-version.jar*)
  - we will write unit tests
  - adding a code library to the project
    - Eclipse → installation
    - or add a Jar file to the project (we don't do that...)
    - or using Maven (we do this more often...)
      - setting dependency in pom.xml

HelloTestWorld

- Let's write a class called **HelloTestWorld**
  - **2 methods:**
  - **int square(int x)**
    - returns the square of the value "x" given in its parameter
  - **int countA(String word)**
    - returns the number of letters "a" in the "word" specified in its parameter

HelloTestWorld

- **HelloTestWorld** class

```
1.public class HelloTestWorld {
2.
3.    public int square( int x) {
4.        return x * x;
5.}
6.
7.    public int countA(String word) {
8.        int count = 0 ;
9.
10.        for ( int i= 0 ;i<word.length();i++) {
11.            if (word.toLowerCase().charAt(i) == 'a')
12.count++;
13.}
14.
15.        return count;
16.}
17.}
```

HelloTestWorld

- **Let's test the operation of the methods**
  - Project → New → Junit Test Case
  - test class name: xTest
    - x: refers to the operation of the test, which method we are writing a test for
  - 2 methods → minimum 2 test cases (could there be more?)
    - **countATest class**
    - **squareBody class**

```
1.@Test
2.public void test() {
3.    fail("Not yet implemented");
4.}
```

HelloTestWorld

- **countATest**

```
public void countATest () {
    // given
    HelloTestWorld helloTestWorld = new HelloTestWorld ();
    beckon result = 0;
    beckon expected = 1;
    String inputString = "monkey" ;

    // when
    result = helloTestWorld.countA (inputString);

    // then
    assertEquals(expected, result);
}
```

HelloTestWorld

- **countANegativeTest**

```
public void countANegativeTest () {
    // given
    HelloTestWorld helloTestWorld = new HelloTestWorld ();
    beckon result = 0;
    beckon expected = 2;
    String inputString = "monkey" ;

    // when
    result = helloTestWorld .countA (inputString);

    // then
    assertNotEquals (expected, result);
}
```

HelloTestWorld

- **squareBody**

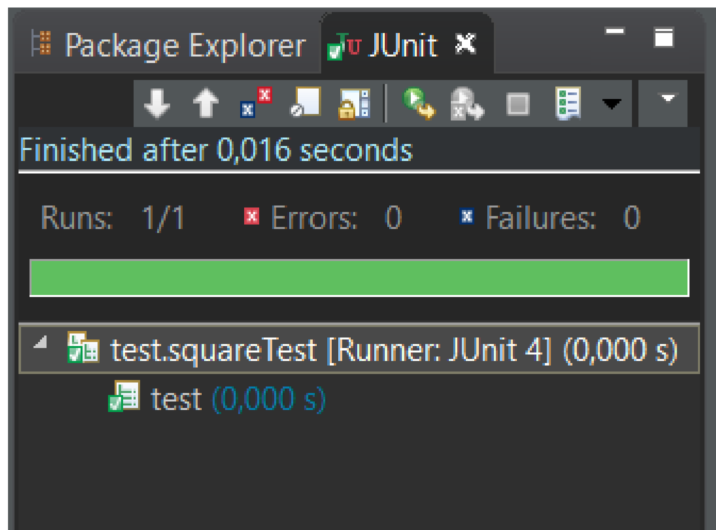
```
public void squareTest () {
    // given
    HelloTestWorld helloTestWorld = new HelloTestWorld ();
    beckon result = 0;
    beckon expected = 25;

   // when
   result = helloTestWorld.square(5);

   // then
   assertEquals (expected, result);
}
```

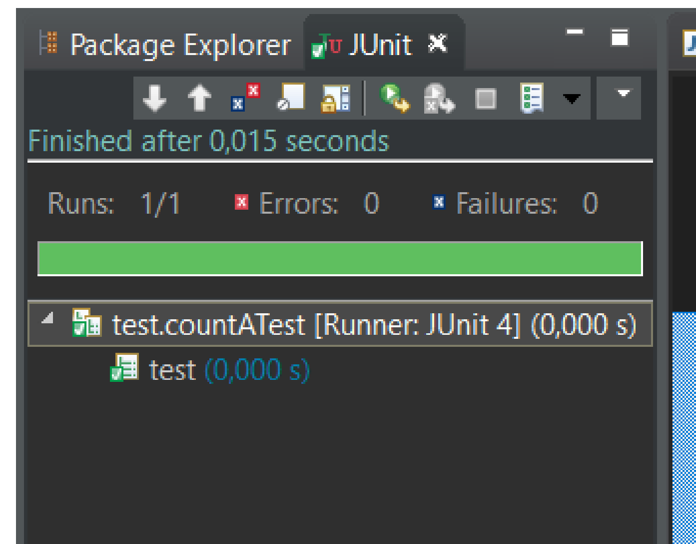
HelloTestWorld

- Always need a class variable, an object to create test (given):
  - `Hello Test World test = new Hello Test World();`
- Saving result in a variable (when):
  - `int output = test.square (5);`
- `assertEquals(expected, actual):` (then)
  - It has 2 parameters (for now)
  - examines that the expected and the actual returned value does it match
  - `assertEquals (25, output);`

HelloTestWorld

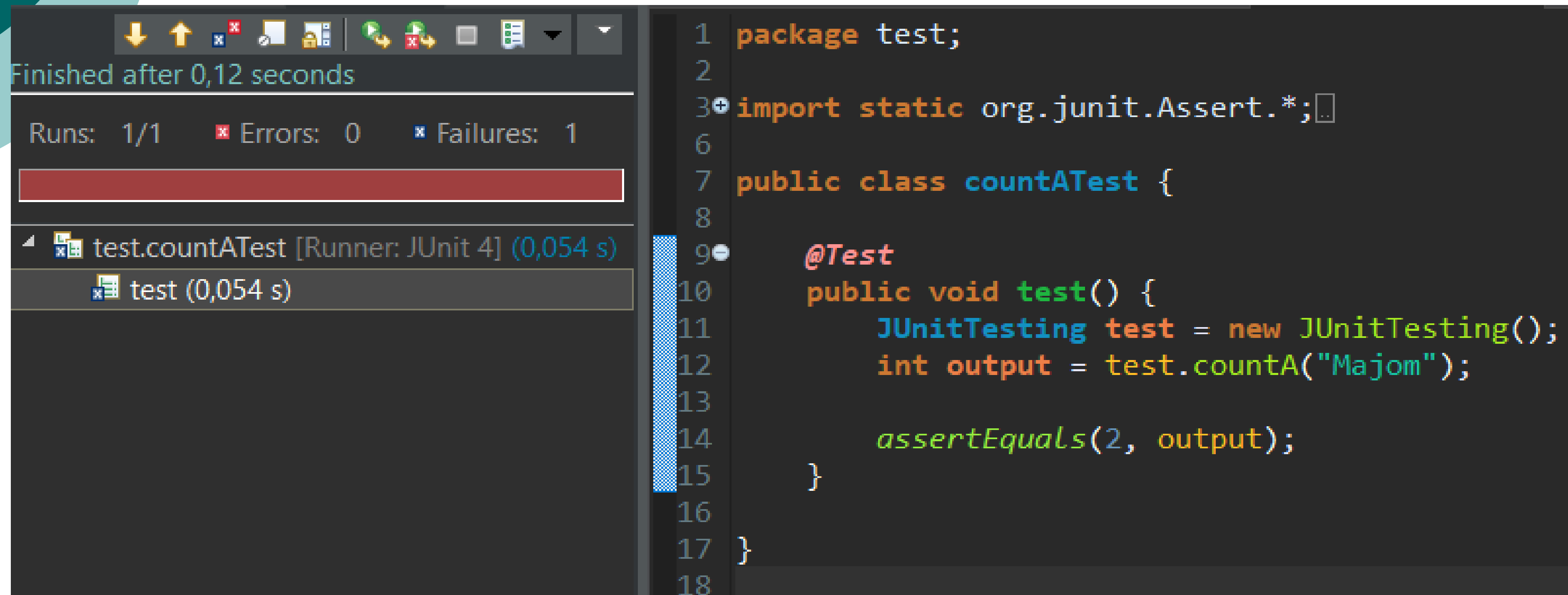
- **Successful test**





HelloTestWorld

- **Failed test**
  - Why?



HelloTestWorld

- Running multiple test cases
  - Creating a Test Suit class

```
1.@RunWith (Suite.class)
2.@SuiteClasses ({squareTest.class, countATest.class})

3.public class AllTests {
4.
5.}
```

HelloTestWorld

- Running multiple test cases
  - A test class, multiple methods
    - `@Test` annotation !

```
1.class AllTestOwn {
2.
3.    @Test
4.    void testSquare() {
5.        HelloTestWorld htw = new HelloTestWorld();
6.        int result = htw.square(5);
7.
8.        assertEquals(25, result);
9.    }
10.
11.    @Test
12.    void testCountA() {
13.        HelloTestWorld htw = new HelloTestWorld();
14.        int result = htw.countA("Monkey");
15.
16.        assertEquals(1, result);
17.    }
18.}
```

2.

Exercise 2.

- Let's write a class named „`PeopleList`” that can store people (people , String , ArrayList)
- Have an „`add`” method that allows a new person to be added to the list
- Have a „`remove`” method that can be used to remove an existing person from the list
  - if the person to be deleted is not on the list → `NoSuchElementException` to throw
- Have a „`size`” method that returns the number of people in the list (list size)
- Have an „`isEmpty`” method that returns whether the list is empty or not
- Have a „`removeAll`” method that empties the list named „people”

Exercise 2.

- Let's write a class called „`ListTest`” that tests the methods of the previously implemented class named „`PeopleList`”
- Before running all tests, the storage should be populated with people (init)
  - `@Before`
- Let's write a test case to check the size of the list (size)
- Let's write a test case to check if the list is empty (isEmpty)
  - `assertTrue / assertFalse`
- a test case for adding a new element (person, String)
- Let's write a test case for the method that empties the list (removeAll)
  - delete then size check

Exercise 2.

- Let's write a test case that involves deleting a non-existing person
  - `exception!`
- Write a method that runs after each test and empties the list (destroy)
  - `@After`

3.

Exercise 3.

- Let's create an interface called „`DatabaseConnection`”
  - "`isUserExisting`" method
  - parameters: username (String) and password (String)
  - return boolean
    - user exists or not

- Let's create a class called „`User`”
  - have a method called „`sum`” with parameters
    - username (String)
    - user password (String)
    - two numbers (int, int)
    - if the given user exists with the given password then return the sum of the 2 numbers, otherwise null

Exercise 3.

- After that, create a „`DatabaseConnection`" class variable in the „`User`" `class.`
  - „User" class should have a constructor which sets the DatabaseConnection
  - we don't implement the interface, then someone else will, we don't know who... It's not even the responsibility of this class!
  - the „`sum`" method, save the return value of DatabaseConnection to a variable
    - only if it returns true should the two numbers be added

Exercise 3.

- Let's test the operation of the methods of the „`User`” class!
  - it is necessary to implement the „`DatabaseConnection`” interface
    - it must be passed in the constructor of the „`User`” class
    - always return with true
      - → Fake object

- Let's test the functionality of the methods of the „`User`” class using mockito!
  - Stub object
  - Mock object

Exercise 3.

- Let's create a test class called „`MockitoTest`”
  - it should contain a list (Collection, List) that we mock out
  - add 2 Strings to this mock list
    - e.g. „hello”, „world”
  - we check with the verify method whether the given String has been added
    - `verify(mockedList).add("hello");`
  - check if a given String has been added only once
    - `verify(mockedList, times (1)).add("world");`
  - check if a given String was not added
    - `verify(mockedList, never()).add("bye");`
    - or `verify(mockedList, times(0)).add("bye");`

4.

Exercise 4.

- Installing Git
  - https://git-scm.com/downloads

- Git commands:
  - `git init` - initialize current directory as repo
    - **.git** folder (this will create a repo)
  - `git init nameOfRepo` - initialize the current directory as a repo but with the given name nameOfRepo
  - `git config --global init.defaultBranch main` - rename the default branch, in this example it is „main”

Exercise 4.

- Git commands:
  - `git branch -m main` – name the master branch „main”
  - `git config --global user.name “nameHere”` – setting username
    - check: `git config --global user.name`
  - `git config --global user.email “emailHere”` – set email for user
    - check: `git config --global user.email`

Exercise 4.

- Git commands:
  - `git status` - repo status, branch, commits, changes, etc.
  - `git log` - view the commit history of the repo
  - `git clone url` – download the contents of a remote repo from a given url
    - eg: git clone https://github.com/ttspeaker88/oop_2022.git
  - `git commit -m "messageHere"` – create a commit with message messageHere, it will include all changes that are in the staging area

Exercise 4.

- Git commands:
  - `git command --help` – „command" command manual
  - `git diff` / `git diff fileName` – what is the difference (change) between the committed file and the current staged file
  - `git push` – upload changes from local repo to remote repo
  - `git pull` – pull the contents of a remote repo into the local repo

Exercise 4.

1. Git practice using cmd

- 1. create a list and enter it
  - `mkdir myFirstRepo`
  - `cd myFirstRepo`
- 2. initialize as repo
  - `git start`
- 3. set the username and email
  - `git config -- global user.name “nameHere”`
  - `git config --global user.email “emailHere”`

Exercise 4.

- 4. create a text file in this directory
  - notepad ++, nano , pico , etc.
- 5. check git status
- 6. mark the file to be versioned
  - `git add fileName` – the file „fileName" is added to the statement to area
  - `git rm -- cached fileName` - if we don't want to version track this file, we need to remove the staging from area (returns to working directory)
- 7. check git status again

Exercise 4.

- 8. create a commit message that will trigger the staging goes to
  - `git commit -m “messageHere”`
- 9. check git status again and git log
- 10. push this change
- 11. modify the contents of the text file
- 12. git Let's look at the differences with diff
- 13. commit the change
- 14. then push again

Exercise 4.

- 15. Revert to a commit
  - git checkout " commit" identifier ”
  - `git checkout 1fb07ce54ad6f14ddf8449fae1ccae21127f2f9f`

Exercise 4.

2. Git practice using cmd

- 1. New Git Create a repository
  - `mkdir my-git-repo`
  - `cd my-git-repo`
  - `git start`
- 2. Add a new file
  - `echo "Hello, Git !" > hello.txt`
  - `git add hello.txt`
- 3. Commit
  - `git commit -m "Add hello.txt with initial content"`

Exercise 4.

- 4. Create a new branch
  - `git branch new-feature`
  - `git checkout new-feature`

- 5. Modify and commit a file on the new branch
  - `echo "This is a new feature." >> hello.txt`
  - `git add hello.txt`
  - `git commit -m "Add new feature to hello.txt"`

- 6. Reverting and merging
  - `git checkout main`
  - `git merge new-feature`

- 7. Check for changes in hello.txt

Exercise 4.

3. Git practice using cmd

- 1. Create a new folder called "git-practice"
- 2. Navigate to this folder
- 3. Initialize Git repository
- 4. Create a new text file named "hello.txt" in the folder
- 5. Write some text in the "hello.txt" file (e.g. "Hello, Git!")
- 6. Add the file to the staging to area

Exercise 4.

- 7. Commit the changes with the following message: "Add hello.txt file"
- 8. Create a new branch called "feature-branch"
- 9. Switch to the "feature-branch" branch
- 10. Modify the contents of the "hello.txt" file (e.g. change the text to "Hello, Git ! This is a feature branch change.")
- 11. Add the modified file to the staging to area
- 12. Commit the changes with the following message: "Modify hello.txt in feature branch"

Exercise 4.

- 13. Switch back to the main branch (main/ master)
- 14. Modify the contents of the "hello.txt" file again (e.g. change the text to "Hello, Git ! This is a change tin the main branch.")
- 15. Add the modified file to the staging to area
- 16. Commit the changes with the following message: "Modify hello.txt in main branch "
- 17. Merge the "feature-branch" branch into the main branch
- 18. Check Git logo

Exercise 4.

3. Git practice using cmd – solution

- Creation and navigation

```
mkdir git practice
cd git -exercise
```

- Initializing Git

```
git start
```

- Create a new file and add content

```
echo "Hello, Git !" > hello.txt
```

- Staging add area

```
git add hello.txt
```

Exercise 4.

- Commit

```
git commit -m "Add hello.txt file"
```

- Create a new branch

```
git branch feature-branch
```

- Branch change

```
git checkout feature-branch
```

- Modify file contents

```
echo "Hello, Git ! This is a feature branch change ." >
hello.txt
```

Exercise 4.

- Staging add area

```
git add hello.txt
```

- Commit to the feature branch

```
git commit -m " Modify hello.txt in feature branch "
```

- Switch back to the main branch

```
git checkout main/ master
```

- Modify the contents of a file on the master branch

```
echo "Hello, Git ! This is a change tin the main branch
." > hello.txt
```

Exercise 4.

- Staging add area

```
git add hello.txt
```

- Commit on the master branch

```
git commit -m " Modify hello.txt in main branch "
```

- Merge feature branch into master branch

```
git to merge feature-branch
```

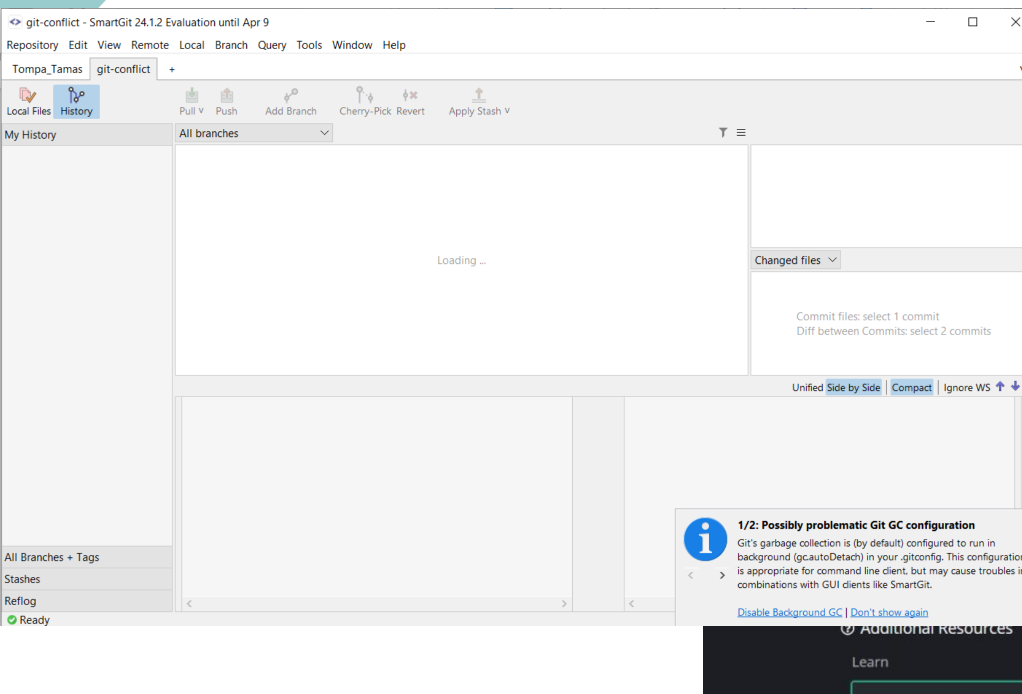
- Check Git log

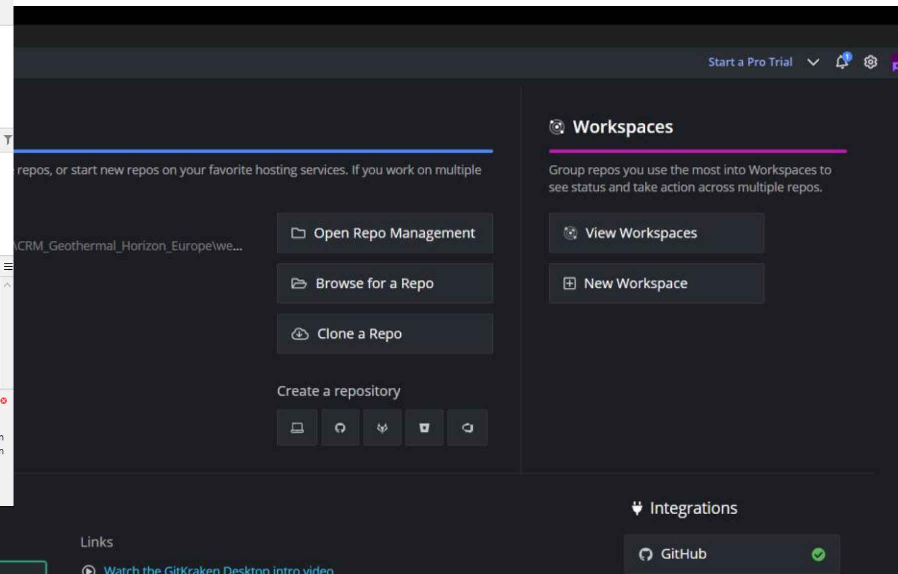
```
git log
```

Exercise 4.

3. Git practice

- Let's see what happens with git clients too
  - SmartGit
  - GitKraken





Exercise 4.

4. Git practice from cmd (conflict resolution)

- 1. Create a new folder and initialize it with Git as repository
  - `mkdir conflict-demo`
  - `cd conflict-demo`
  - `git start`
- 2. Create and commit a file
  - `echo "Hello, Git !" > hello.txt`
  - `git add hello.txt`
  - `git commit -m "Add hello.txt with initial "content "`

Exercise 4.

- 3. Create a new branch , modify a file, commit
  - `git branch feature-branch`
  - `git checkout feature-branch`
  - `echo "This is a feature branch." >> hello.txt`
  - `git add hello.txt`
  - `git commit -m "Add feature branch content to hello.txt"`

- 4. Revert to the main branch and then modify the file
  - `git checkout master`
  - `echo "This is the main branch." >> hello.txt`
  - `git add hello.txt`
  - `git commit -m "Add main branch content to hello.txt"`

Exercise 4.

- 5. Merge resulting in conflict
  - `git merge feature-branch`

- 6. Resolve conflict by merging content (manually)

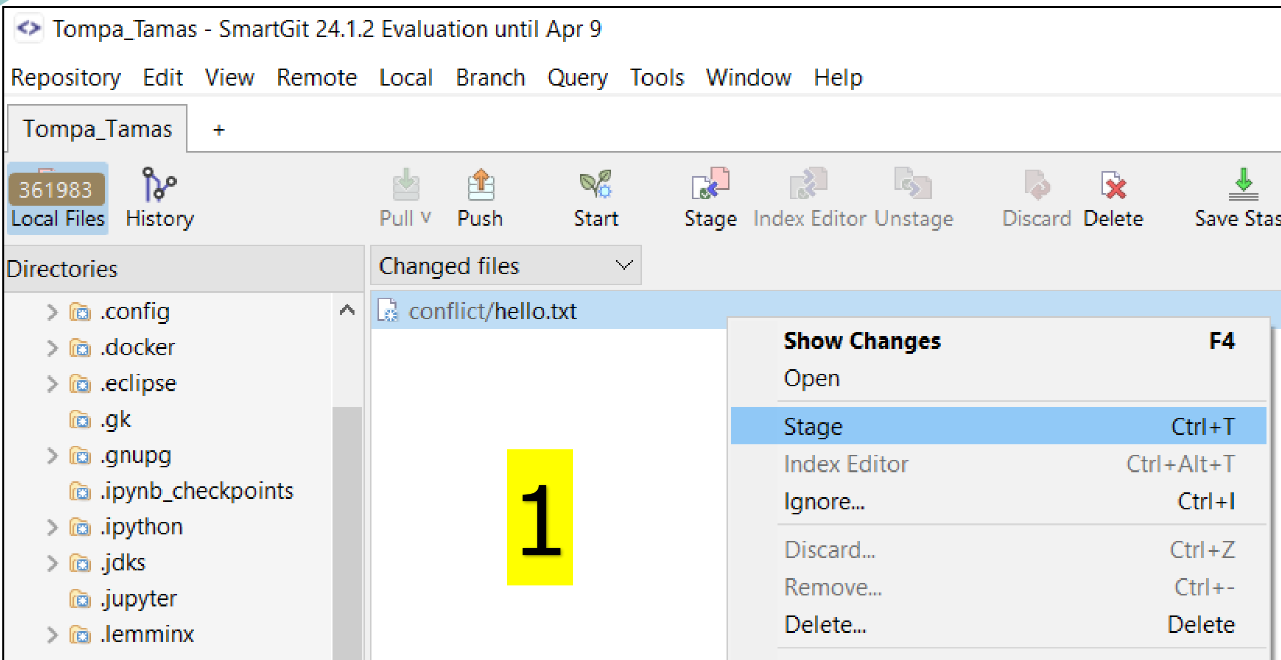
```
Hello, Git!
This is the main branch.
This is a feature branch.
```

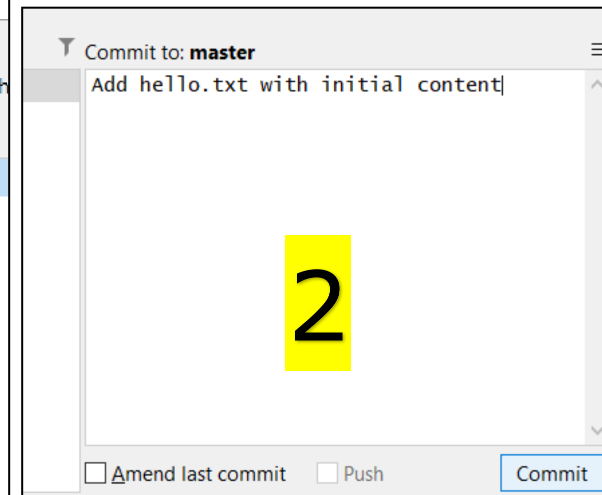
- 7. After saving, add file and then commit
  - `git add hello.txt`
  - `git commit -m "Resolve merge conflict by combining branch contents"`

Exercise 4.

4. Git practice SmartGit (conflict resolution)

- 1. Create a new folder and initialize it with Git as
- 2. Create and commit a file





Exercise 4.

4. Git practice SmartGit (conflict resolution)

git-conflict - SmartGit 24.1.2 Evaluation until Apr 9

Repository Edit View Remote Local Branch Query Tools Window Help

Tompa_Tamas | git-conflict | +

Local Files | History | Pull | Push | Start | Integrate | Finish | Cherry-Pick | Revert | Apply Stash

My History

master

All branches

master Add hello.txt with initial content — T 18:38

initial commit — T 18:38

Commit 0e7ab628

by Tompa_Tamas, 2025-03-16 18:38

Add hello.txt with initial content

Changed files

hello.txt

3

hello.txt — Unified | Side by Side | Compact | Ignore WS

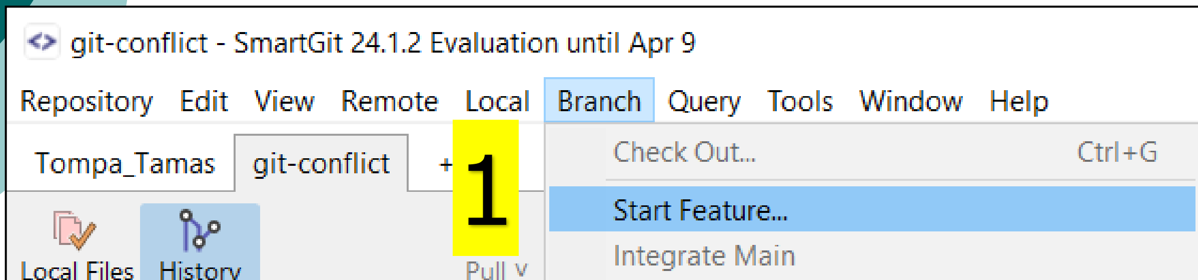
1

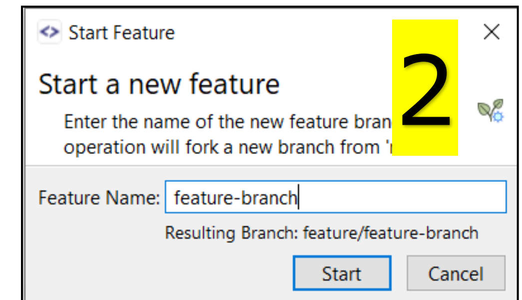
1 Hello, Git!

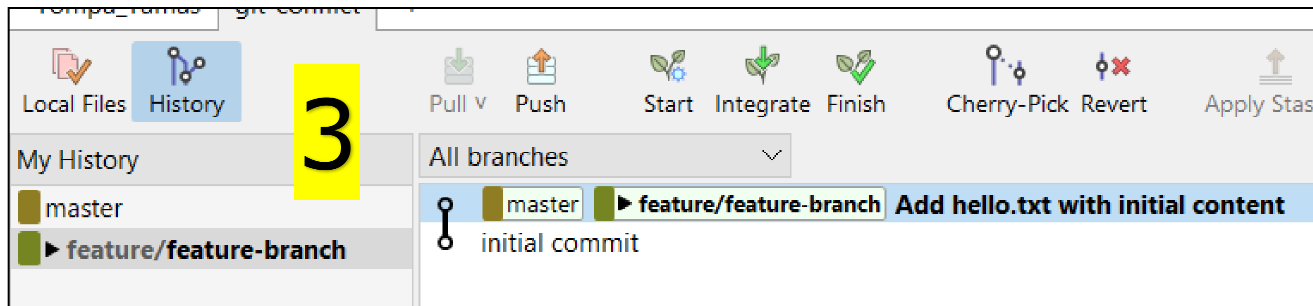
Exercise 4.

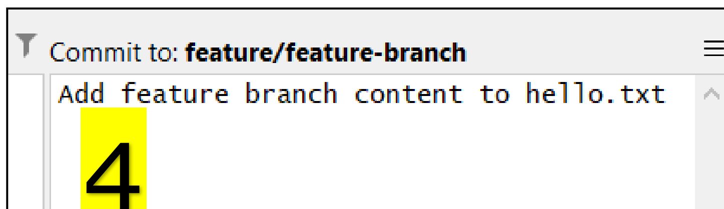
4. Git practice SmartGit (conflict resolution)

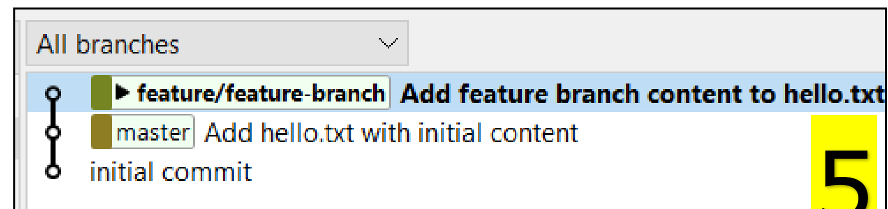
- 3. Create a new branch , modify a file, commit







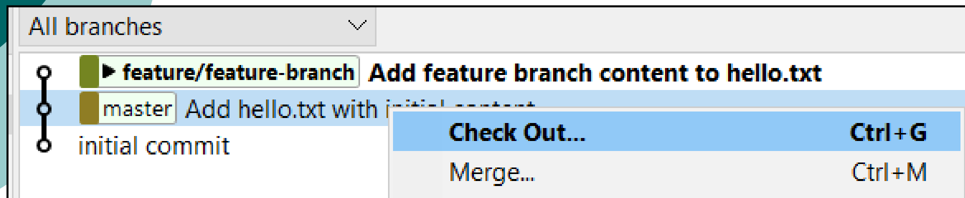




Exercise 4.

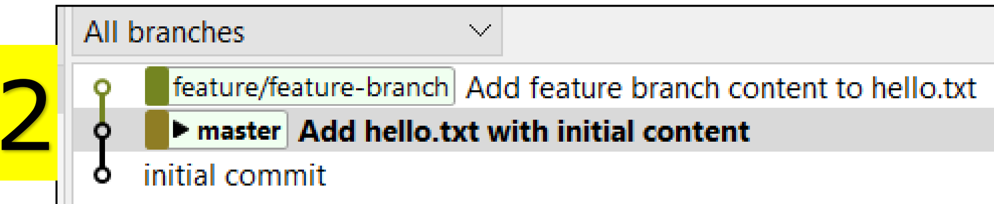
4. Git practice SmartGit (conflict resolution)

- 4. Revert to the main branch and then modify the file



1

```
C:\TT\git-conflict>git checkout master
Switched to branch 'master'
```



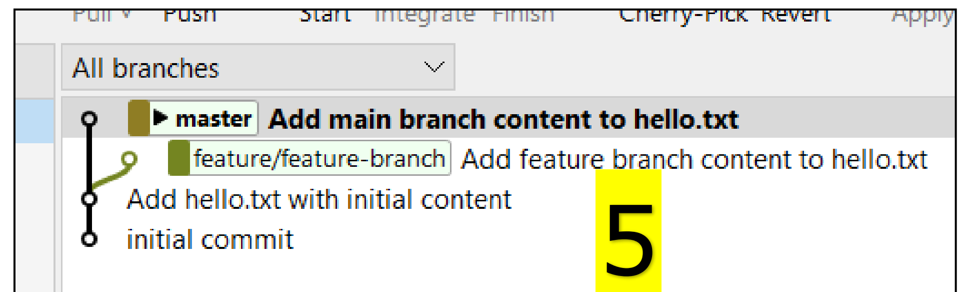
3

This is the main branch.

4

Commit to: **master**

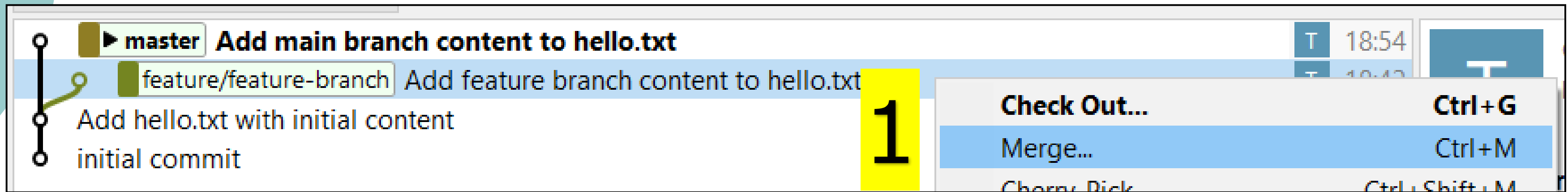
Add main branch content to hello.txt

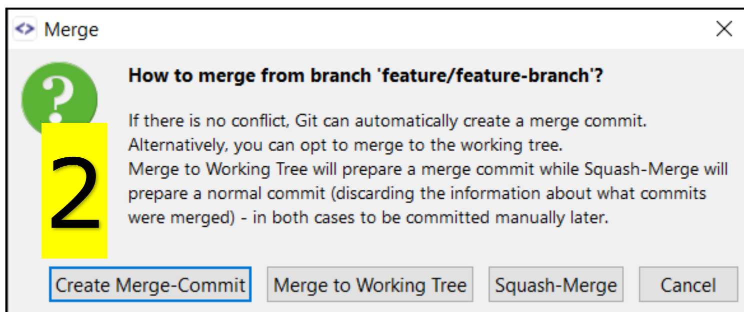


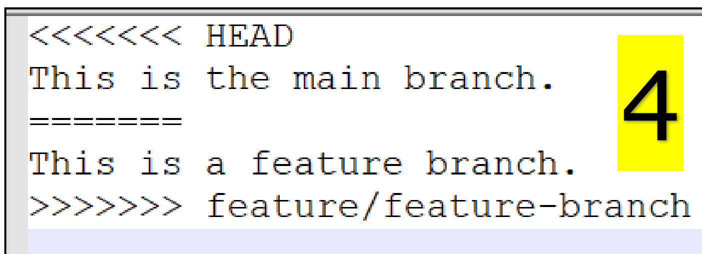
Exercise 4.

4. Git practice SmartGit (conflict resolution)

- 5. Merge resulting in conflict



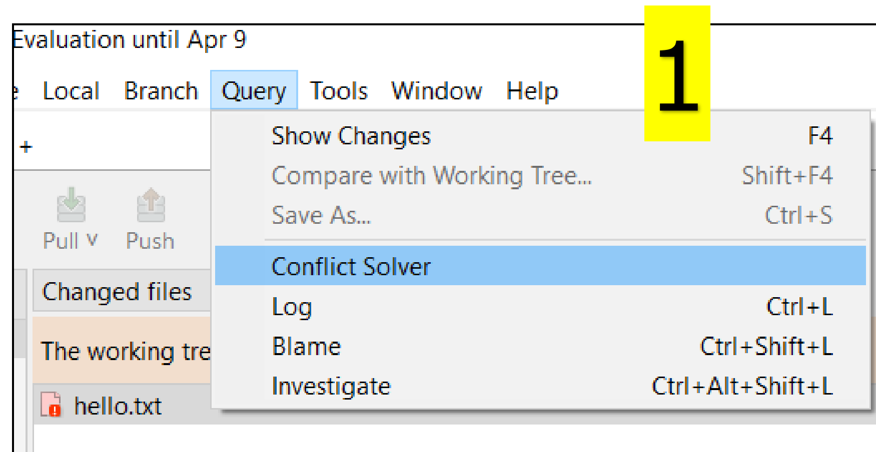


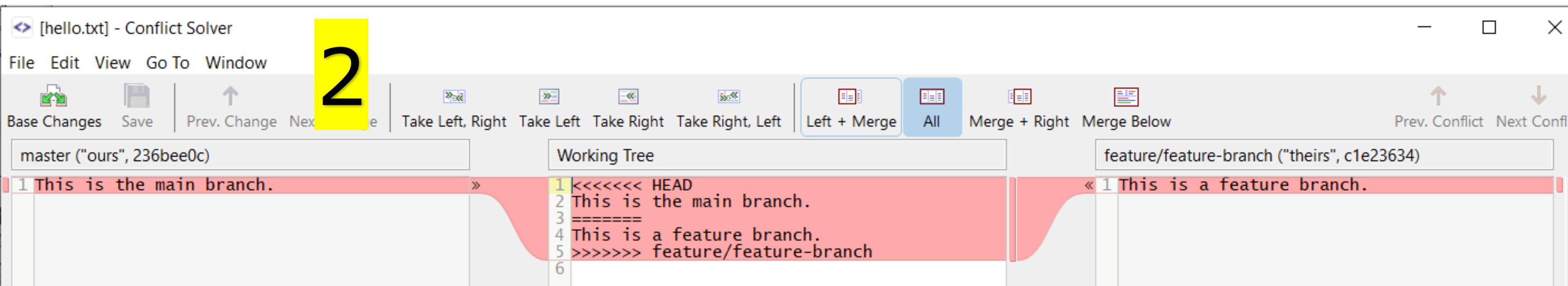


Exercise 4.

4. Git practice SmartGit (conflict resolution)

- 6. Conflict Resolution Solver menu item)

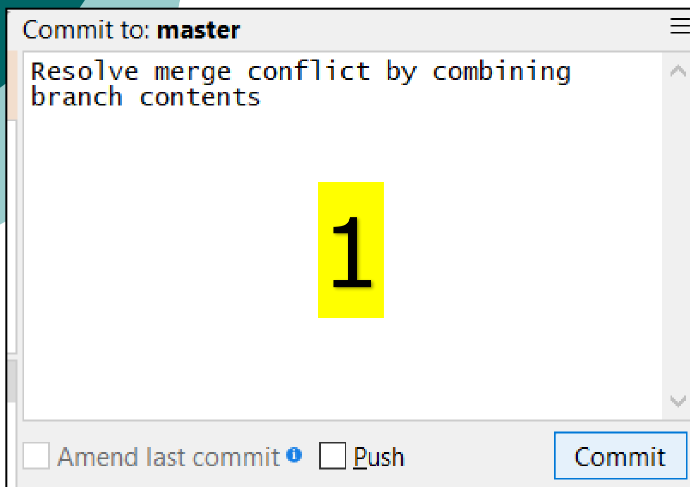


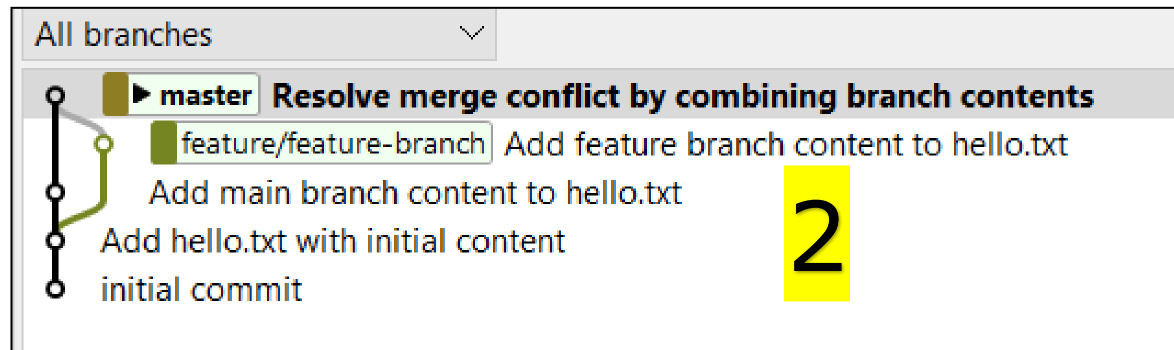


Exercise 4.

4. Git practice SmartGit (conflict resolution)

- 7. After saving, add file and then commit





- The 2 changes were merged → the conflict is resolved

3

```
1  This is the main branch.
2  This is a feature branch.
```

Exercise 4.

5. Git practice from cmd (merge - rebase)

- 1. ……
  - `mkdir conflict-demo`

5.

Exercise 5.

- **Let's write a simple calculator program in Java**
  - request input data from console (in one step, String)
    - e.g.: 3 + 5 (space delimiter)
    - we check the correctness of the input data (operand, operator)
  - knows the 4 basic operations
    - when dividing by 0, there should be an error message (exception)
    - also an error message (exception) in case of an uninterpretable operation
  - also prints the result to the console
    - E.g.: 3 + 5 = 8

  - How and where do we break down functions into methods? (how many?)
  - What classes should there be? (and how many?)
  - Let's write tests too (what kind?)

Exercise 5.

- 1. first write it as spaghetti code
  - is this OOP?
  - Is it easy to write tests for this? Is it testable at all?
- 2. then we break this down into methods
  - each method should only do one thing
- 3. then we cut this into layers
  - each layer should have a well-defined task
  - main should consist only of method calls
- 4. add a unit test

Exercise 5.

- 2. divide the spaghetti code into methods
  - Please enter the operation to be performed.
  - Return the entire line read from the console as a String
  - Split the read data (row) with Split, then return the String array
  - Split the data in the String array
    - how? double and char types → more methods
    - 1 method that returns char (operator)
    - 1 method that returns a double array (with operands)

Exercise 5.

- 2. divide the spaghetti code into methods
  - Create an object from the previously split data
    - model class introduction
    - write a method that returns this model reference with the appropriate data
  - Write a method that returns the result of the operation performed.
    - we transfer the individual operations to the service
    - write an interface → introduce a controller layer
    - introducing own Exception classes for errors
      - DividedByZeroException (division by 0)
      - NoDefinedOperationException (not a valid operation)

Exercise 5.

- 3. organize the divided methods into layers
  - each layer should have a well-defined task
  - a lower layer provides a service to a layer above it
  - MVC
  - main should consist only of method calls

- 4. add a unit test
  - unit test

thank you