

## Практичне заняття 1

### Керування версіями файлів як основа управління кодом

**Мета:** сформувати базові навички управління документами та програмним кодом.

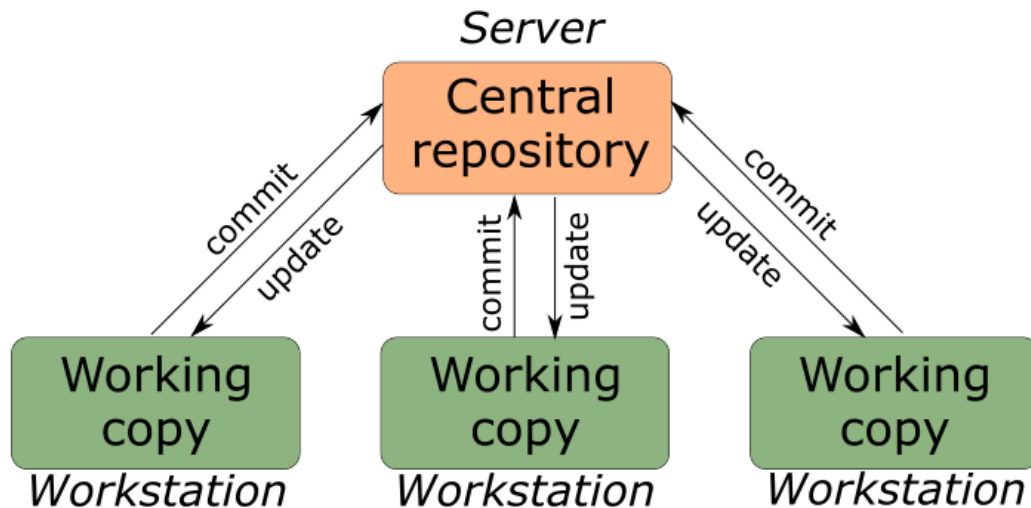
#### Система оцінювання

№	Тема	Оцінка
1.	Завдання 1.1	2
2.	Завдання 1.2	1
	Завдання 1.3	1
	Завдання 1.4	1
	Завдання 1.5	2
	Завдання 1.6	1
	<b>Всього за практичну</b>	<b>8</b>
3.	Групово розробка реферативного повідомлення (ІНДЗ)	6
	<b>Всього</b>	<b>14</b>

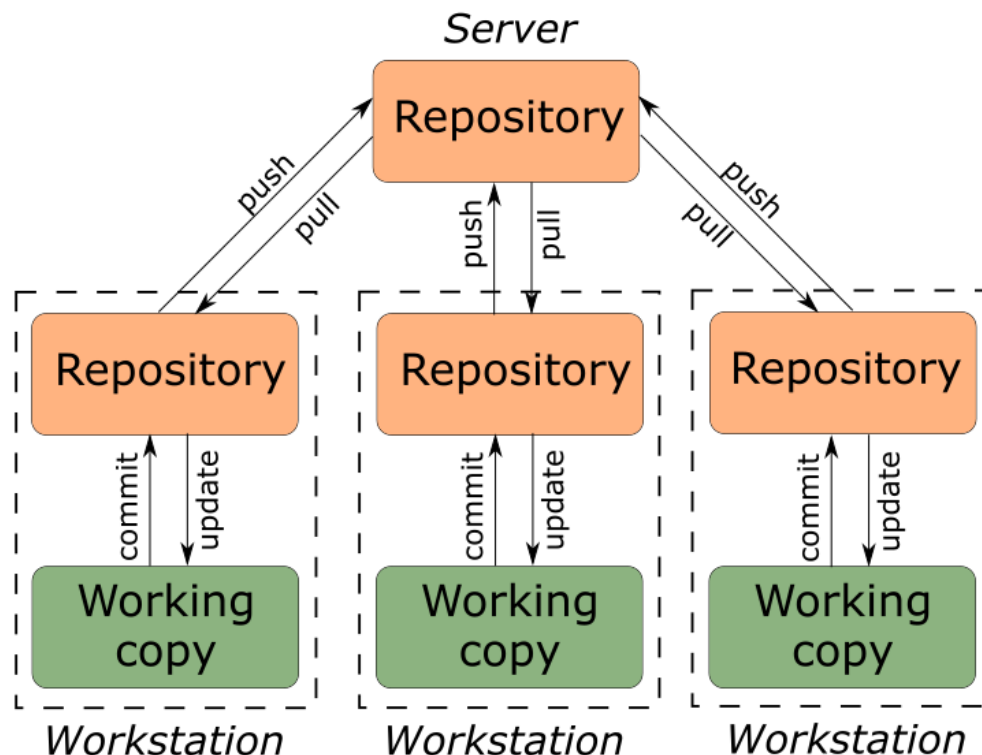
#### Знайомство з командним рядком та базовими командами Git

Контроль версій передбачає зберігання всіх версій редагованих документів та можливість повернутись до будь-якої збереженої версії в будь-який момент часу. **Система контролю версій (Version Control System, VCS)** – це програмне забезпечення, яке дозволяє відстежувати зміни в документах, при необхідності проводити їх відкат, визначати, хто і коли вніс виправлення і т. п. Залежно від способу зберігання виділяють *централізовані* та *розподілені* системи контролю версій.

**Централізовані системи контролю версій** представляють собою додатки типу клієнт-сервер. *Репозиторій* (сховище даних) проекту існує в єдиному екземплярі та зберігається на сервері. Доступ до нього здійснюється через спеціальний клієнтський додаток. До таких програмних продуктів можна віднести CVS та Subversion.

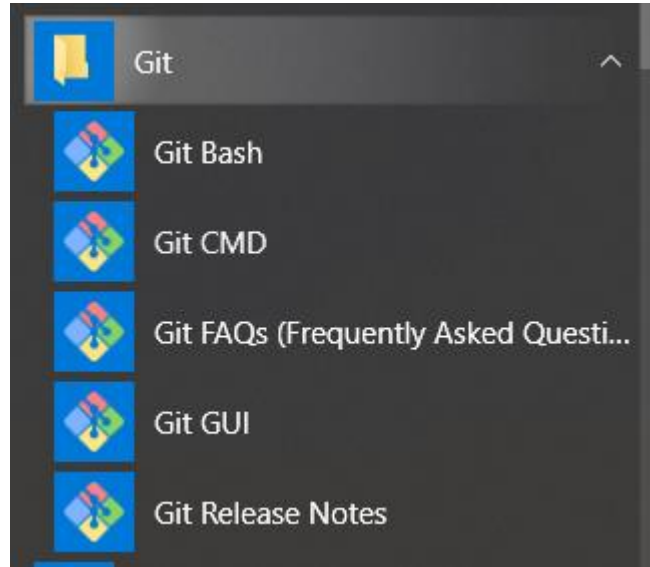


**Розподілені системи контролю версій (Distributed Version Control System, DVCS, РСКВ)** дозволяють зберігати репозиторій (його копію) для кожного розробника, що працює з даною системою. При цьому можна виділити центральний репозиторій (умовно), в який будуть відправлятися зміни з локальних і, з ним же ці локальні репозиторії будуть синхронізуватися. При роботі з такою системою, користувачі періодично синхронізують свої локальні репозиторії з центральним і працюють безпосередньо зі своєю локальною копією. Після внесення достатньої кількості змін в локальну копію вони (зміни) відправляються на сервер. При цьому сервер, найчастіше, вибирається умовно, тому що в більшості РСКВ немає такого поняття як "виділений сервер з центральним репозиторієм".



Велика перевага такого підходу полягає в автономії розробника при роботі над проектом, гнучкості загальної системи і підвищення надійності, завдяки тому, що кожен розробник має локальну копію центрального сховища. Дві найбільш відомі DVCS – це Git і Mercurial.

РСКВ Git постачається у вигляді інсталятора з сайту [git-scm.com](http://git-scm.com). Після встановлення даного програмного продукту для операційної системи Windows в меню Пуск з'явиться набір програмних інструментів, які дозволяють розпочати роботу з системою контролю версій. Основним засобом для виконання задач практичної роботи є Git Bash.



#### Етап 0. Поширені команди для командного рядка bash.

Серед засобів, які можуть інтерпретувати команди Git, поширеним є командний рядок Git Bash. У його основі лежить командний рядок UNIX-подібних операційних систем (bash, рис. 1), доповнений специфічними для Git командами. Командний рядок bash дозволяє виконувати

```
chris@ubuntu: ~  
chris@ubuntu:~$ bash --version  
GNU bash, version 4.3.46(1)-release (x86_64-pc-linux-gnu)  
Copyright (C) 2013 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.  
This is free software; you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law.  
chris@ubuntu:~$
```

Рис. 1. Вигляд командного рядка bash в операційній системі Ubuntu

Сучасні файлові системи мають дерева директорій (папок), у яких директорією є кореневий каталог (root directory, без батьківського каталогу) або субдиректорією (subdirectory, підпакою, розміщеною в іншому каталозі). Постійний рух по дереву від дочірнього до батьківського елементу завжди веде до кореневого каталогу. Деякі файлові системи мають кілька кореневих каталогів (наприклад, диски Windows C:\, D:\ та ін.), проте Unix та Unix-подібні операційні системи мають єдиний кореневий каталог, який називається `/`.

Під час роботи з файловою системою користувач завжди працює всередині деякої папки, яку називають поточною (робочою) директорією. Вивести робочу директорію можна за допомогою команди **pwd** (print working directory)

```
$ pwd
```

Вміст директорії у вигляді списку файлів та/або субдиректорій дозволяє отримати команда **ls** (list):

```
$ ls
```

Показати приховані (“dot”) файли можна за допомогою команди **ls -a**  
Показати детальну інформацію щодо файлів дозволяє команда **ls -l**  
Допускається комбінування кількох прапорців, на зразок **ls -l -a**  
Іноколи можна визначати прапорці ланцюжком: **ls -la** замість **ls -l -a**

*Якщо поточний каталог не вміщає в собі підкаталогів, створіть їх за допомогою команди **mkdir***

`$mkdir назвакаталогу`

Змінити директорію можна за допомогою команди **cd** (change directory).  
*Перегляньте вміст поточної директорії та перейдіть в одну з субдиректорій. Виведіть поточний каталог, потім – його вміст та ще раз. Якщо підкаталог існує, перейдіть у нього. Інакше – створіть за допомогою команди **mkdir**. Поверніться в батьківську директорію за допомогою команди*

```
$ cd ..
```

*Перейдіть до домашньої директорії, використовуючи команду **cd ~**.  
Перейдіть у нашілибу субдиректорію та спробуйте повернутись у домашній каталог шляхом поступового підйому по дереву каталогів: **cd ../..***

За потреби послідовного виконання кількох команд можна використовувати символ «;»:

```
$ ls; pwd
```

Іншим корисним інструментом для постановки команд у ланцюжок є **&&**: наступна команда після **&&** не буде виконуватись, якщо попередня не змогла запуснитись:

```
$ cd dir1/dir2 && pwd && ls && cd
```

*Запустіть дану команду в поточному вигляді та з реальними субдиректоріями на Вашій машині. Порівняйте роботу цих команд з відповідним записом через «;».*

З метою виведення допомоги щодо використання деякої команди спробуйте використовувати прапорець **-h** або **--help**

```
$ du --help
```

Інформаційну довідку по команді можна отримати за допомогою команди **man** (manual). Вихід з режиму довідки здійснюється при натисненні клавіші **q**.

```
$ man ls
```

Окремі команди передбачені для перегляду та редагування файлів. Спочатку розглянемо найпростіші можливості зі створення та видалення файлів та директорій. Команда **touch** спочатку була створена для редагування таймштампу файлів, проте також може застосовуватись для створення порожнього файлу. Виконати елементарне редагування файлу можна в простому текстовому редакторі **nano**.

```
$ nano назвафайлу
```

Перехід між режимами редагування тексту та командним рядком відбувається при натисненні **Ctrl+Z** (вихід з nano) та виклику команди **fg** (повернення в nano). *Створіть кілька файлів, у кожному з яких буде 10-15 рядків тексту.*

У командному рядку **bash** можна запустити й інші текстові редактори, як з графічним інтерфейсом, так і без: **nedit**, **emacs**, **vi**, **vim**, **gedit**, **Notepad++**, **Atom** та ін. Сучасні текстові редактори пропонують базові зручності на зразок пошуку та заміни, підсвітки синтаксису тощо.

Наприклад, команда **head** виводить перші кілька рядків вмісту файлу. Прапорець **-n** визначає кількість таких рядків, за умовчанням їх 10:

```
$ head -n 3 назвафайлу
```

Останні декілька рядків дозволяє вивести команда **tail**. Для встановлення кількості використовуються прапорці **-n +N**, де **N** – кількість рядків:

```
$ tail -n +4 назвафайлу
```

Команда **cat** конкатенує список файлів та направляє їх у стандартний потік виводу (зазвичай у термінал). Дана команда може застосовуватись як для одного файлу, так і для кількох. Часто вона дозволяє швидко переглянути кілька файлів (при цьому може з'явитись звинувачення [UUOC](#)).

```
$ cat назвафайлу1 назвафайлу2
```

Інший інструмент швидкого перегляду файлу – команда [less](#). Вона відкриває vim-подібне вікно, текст доступний тільки для зчитування (також існує команда **more**, проте вона має менше можливостей).

Видалення файлу можна здійснити за допомогою команди **rm**, проте будьте обережні, оскільки відновленню такі файли вже не підлягають:

```
$ rm назвафайлу && ls
```

Безпечніше додати перевірочне сповіщення "are you sure?" за допомогою прапорця **-i**. Виглядає це приблизно так:

```
$ rm -i назвафайлу
```

```
rm: remove regular empty file 'назвафайлу'? y
```

*Виведіть вміст директорії з файлами, видаліть один зі створених файлів та виведіть оновлений вміст директорії.*

Для створення та видалення каталогів застосовують команди **mkdir** та **rmdir** відповідно. Команда **rmdir** видаляє тільки порожні директорії, проте можливо видалити каталог з усім вмістом:

```
$ rm -rf назвапакки
```

*-r = recursive, -f = force. Проведіть аналогічні операції з каталогом та виведіть вміст батьківської директорії до і після видалення..*

Команда переміщення файлу суміщена з командою перейменування – **mv**. Можливо перемістити файл у нову директорію без зміни назви або встановити «новий файл»:

```
$ ls && mv шляхдофайлу новийшляхдофайлу && ls
```

*Перемістіть один з файлів на рівень вище та перейменуйте його довільним чином. Поверніться до старого каталогу та скопіюйте з нього файл у новостворений каталог (команди **mkdir** та **cp**). Аргументи копіювання подібні до аргументів **mv**.*



### Завдання 1.1.

*Виконайте завдання, текст яких виділений курсивом вище.*

## Етап 1. Конфігурація.

Інформація щодо конфігурації Git зберігається в трьох місцях, які визначають широту застосування цієї конфігурації:

- 1) Конфігурація на рівні системи – налаштування для всіх користувачів комп'ютера за умовчанням. Використовуються нечасто, оскільки кожний користувач виконує налаштування під себе. Зберігаються в папках
  - a. `/etc/gitconfig` – на Linux;
  - b. `Program Files\Git\etc\gitconfig` – на Windows;

комп'ютер > Windows 10 Compact (C:) > Program Files > Git > etc

Имя	Дата изменения	Тип	Размер
protie.d	06.07.2020 09:59	Папка с файлами	
ssh	06.07.2020 09:59	Папка с файлами	
bash.bash_logout	01.06.2020 18:55	Исходный файл В...	1 КБ
bash.bashrc	01.06.2020 18:55	Исходный файл В...	3 КБ
DIR_COLORS	01.06.2020 18:55	Файл	5 КБ
docx2txt.config	01.06.2020 18:55	Исходный файл С...	2 КБ
fstab	01.06.2020 18:55	Файл	1 КБ
gitattributes	01.06.2020 18:55	Файл	1 КБ
gitconfig	06.07.2020 10:00	Файл	1 КБ

- 2) Налаштування на рівні користувача – застосовуються до окремих користувачів. Знаходяться в файлі
  - a. `~/.gitconfig` – на Linux;
  - b. `$HOME\.gitconfig` – на Windows.

Этот компьютер > Windows 10 Compact (C:) > Пользователи > ruasson

Имя	Дата изменения	Тип	Размер
hero	10.08.2020 11:14	Папка с файлами	
source	29.06.2020 08:14	Папка с файлами	
Видео	28.06.2020 20:33	Папка с файлами	
Документы	13.07.2020 16:01	Папка с файлами	
Загрузки	03.09.2020 16:00	Папка с файлами	
Избранное	28.06.2020 20:33	Папка с файлами	
Изображения	14.07.2020 17:44	Папка с файлами	
Контакты	28.06.2020 20:33	Папка с файлами	
Музыка	28.06.2020 20:33	Папка с файлами	
Объемные объекты	28.06.2020 20:33	Папка с файлами	
Поиски	07.07.2020 13:18	Папка с файлами	
Рабочий стол	23.07.2020 09:00	Папка с файлами	
Сохраненные игры	28.06.2020 20:33	Папка с файлами	
Ссылки	28.06.2020 22:13	Папка с файлами	
.bash_history	13.08.2020 16:03	Файл "BASH_HIST...	1 КБ
.emulator_console_auth_token	14.07.2020 16:38	Файл "EMULATOR...	1 КБ
.gitconfig	10.08.2020 11:10	Исходный файл G...	1 КБ

3) Налаштування для проекту – специфічні налаштування для конкретного проекту. Знаходяться в файлі .git/config у папці проекту.

Этот компьютер > ssd (F:) > csbc-github > csbc-edu.github.io > .git

Имя	Дата изменения	Тип	Размер
hooks	10.08.2020 11:31	Папка с файлами	
info	10.08.2020 11:31	Папка с файлами	
logs	10.08.2020 11:35	Папка с файлами	
objects	28.08.2020 19:19	Папка с файлами	
refs	10.08.2020 11:35	Папка с файлами	
COMMIT_EDITMSG	02.09.2020 20:37	Файл	1 КБ
config	10.08.2020 11:35	Файл	1 КБ
description	10.08.2020 11:31	Файл	1 КБ

Можливі деякі відмінності в назвах файлів залежно від операційної системи, проте її легко знайти за вказаними розташуваннями.

Командний рядок Git пропонує команду `git config` для встановлення налаштувань. Дана команда враховує рівень конфігурації за допомогою відповідних модифікаторів:

- **git config --system** (на рівні системи);
- **git config --global** (на рівні користувача, тобто глобально для користувача);



- **git config** (на рівні проекту модифікатор не використовується).

Наприклад, за потреби задати ПІБ користувача використовується

```
MINGW64:/c/Users/puasson  
  
puasson@HOME-PC MINGW64 ~  
$ git config --global user.name "Stanislav Marchenko"
```

Для виводу значення конкретного налаштування введіть ту ж команду, проте без останнього параметра (тексту в лапках). Повний перелік налаштувань дозволяє вивести на екран команда

**git config --list**

```
puasson@HOME-PC MINGW64 ~  
$ git config user.name  
Stanislav Marchenko  
  
puasson@HOME-PC MINGW64 ~  
$ git config --list  
diff.astextplain.textconv=astextplain  
filter.lfs.clean=git-lfs clean -- %f  
filter.lfs.smudge=git-lfs smudge -- %f  
filter.lfs.process=git-lfs filter-process  
filter.lfs.required=true  
http.sslbackend=openssl  
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-bundle.crt  
core.autocrlf=true  
core.fscache=true  
core.symlinks=false  
pull.rebase=false  
credential.helper=manager  
filter.lfs.process=git-lfs filter-process  
filter.lfs.required=true  
filter.lfs.clean=git-lfs clean -- %f  
filter.lfs.smudge=git-lfs smudge -- %f  
user.name=Stanislav Marchenko  
user.email=54304912+csbc-edu@users.noreply.github.com  
core.editor=C:\Program Files\Microsoft Office\root\Office16\WINWORD.EXE
```

Подивитись наповнення файлу з конфігураціями можна і звичайними командами bash:

```

puasson@HOME-PC MINGW64 ~
$ ls -la
total 8064
drwxr-xr-x 1 puasson 197121      0 сен  3 17:15 ./
drwxr-xr-x 1 puasson 197121      0 июн 28 20:34 ../
drwxr-xr-x 1 puasson 197121      0 июл 14 17:49 .android/
drwxr-xr-x 1 puasson 197121      0 июн 29 09:13 .AndroidStudio4.0/
-rw-r--r-- 1 puasson 197121    297 авг 13 16:03 .bash_history
drwxr-xr-x 1 puasson 197121      0 июл 15 22:11 .config/
drwxr-xr-x 1 puasson 197121      0 авг 15 16:27 .dotnet/
-rw-r--r-- 1 puasson 197121    16 июл 14 16:38 .emulator_console_auth_token
-rw-r--r-- 1 puasson 197121    295 сен  3 17:15 .gitconfig
drwxr-xr-x 1 puasson 197121      0 июл 14 17:04 .gradle/
drwxr-xr-x 1 puasson 197121      0 июл  7 22:18 .idlerc/
drwxr-xr-x 1 puasson 197121      0 июл 30 11:19 .librarymanager/
drwxr-xr-x 1 puasson 197121      0 июл 30 16:41 .nuget/
drwxr-xr-x 1 puasson 197121      0 июл 26 11:39 .templateengine/
drwxr-xr-x 1 puasson 197121      0 июн 28 22:28 .vscode/
drwxr-xr-x 1 puasson 197121      0 июн 28 20:33 '3D Objects'/
drwxr-xr-x 1 puasson 197121      0 июл 14 17:38 AndroidStudioProjects/
drwxr-xr-x 1 puasson 197121      0 июн 28 20:32 AppData/

```

Виведемо повний перелік файлів за допомогою команди `ls -la`, а потім переглянемо вміст файлу `.gitconfig` за допомогою команди `cat`:

```

puasson@HOME-PC MINGW64 ~
$ cat .gitconfig
[filter "lfs"]
    process = git-lfs filter-process
    required = true
    clean = git-lfs clean -- %f
    smudge = git-lfs smudge -- %f
[user]
    name = Stanislav Marchenko
    email = 54304912+csbc-edu@users.noreply.github.com
[core]
    editor = C:\\Program Files\\Microsoft Office\\root\\Office16\\WINWORD.EXE

```



## Завдання 1.2.

Розгляньте [довідку](#) щодо конфігурації Git та встановіть наступні налаштування:

- `user.name` – своє ім'я та прізвище
- `user.email` – адресу своєї електронної пошти
- `core.editor` – шлях до зручного текстового редактору
- `commit.template` – шаблон повідомлення для коммітів
- `color.ui` – для підтримки підсвічування тексту в терміналі різними кольорами

Відобразіть у звіті скриншоти з виконанням відповідних команд та повний вміст конфігураційного файлу

Розгляньте статтю [“Особенности настройки git под windows”](#) та налаштуйте підтримку кирилических символів у консолі Git Bash.

## Етап 2. Створення репозиторію та перший коміт.

Команда `git init` (скорочено від `initialize`) дозволяє ініціалізувати проект та почати його відстеження. Спочатку потрібно вирішити, де буде розташовано проект. Для цього в консолі перейдемо у відповідну папку, створимо директорію (тут – `myrepo`) для проекту та ініціалізуємо відстеження засобами Git:

```
puasson@HOME-PC MINGW64 ~
$ pwd
/c/Users/puasson

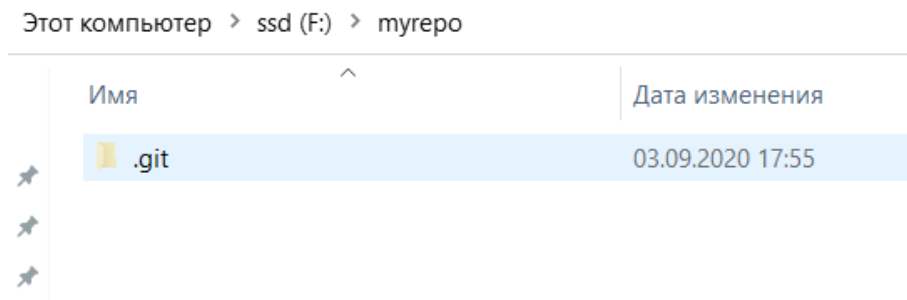
puasson@HOME-PC MINGW64 ~
$ cd F:\

puasson@HOME-PC MINGW64 /f
$ mkdir myrepo

puasson@HOME-PC MINGW64 /f
$ cd myrepo

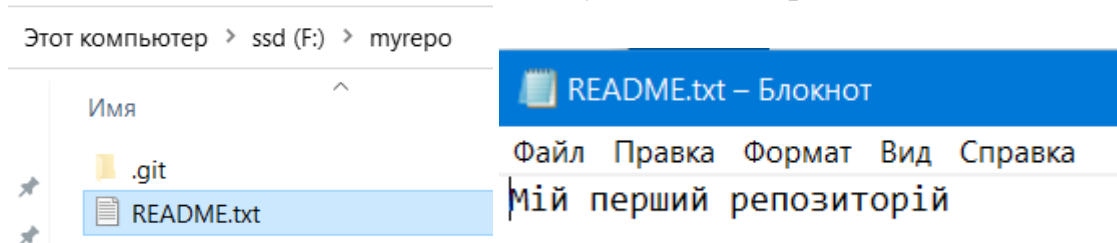
puasson@HOME-PC MINGW64 /f/myrepo
$ git init
Initialized empty Git repository in F:/myrepo/.git/
```

З Провідника ініціалізована папка виглядає так:



Саме наявність *прихованої* папки `.git` дозволяє системі контролю версій стежити за файлами в папці. Якщо виникне потреба прибрати контроль версій для папки, достатньо буде видалити `.git`. Всередині знаходяться файли, якими самостійно керує Git, тому розробник зазвичай з ними напямую не працює.

Тепер можемо внести зміни, які будуть відстежуватись системою контролю версій. Додамо в відстежувану папку текстовий файл:

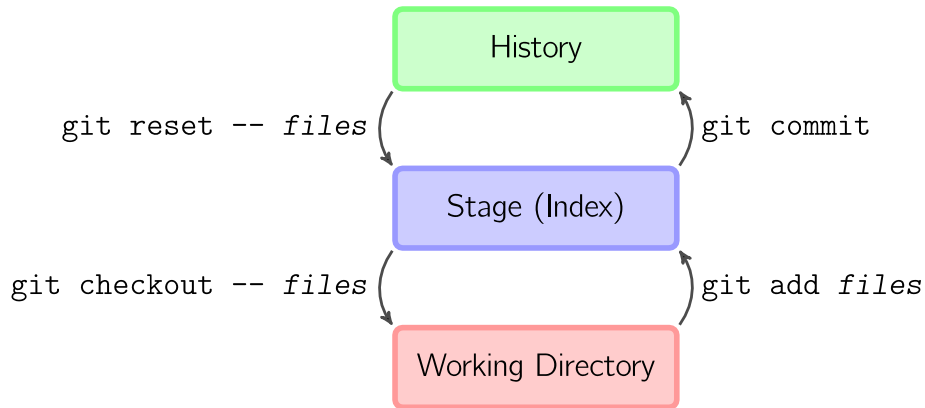


На даному етапі файл знаходиться в потрібній папці, проте ще не відстежується системою контролю версій. Кожний файл може перебувати в одній з кількох областей (area):

- у робочій директорії (working directory);

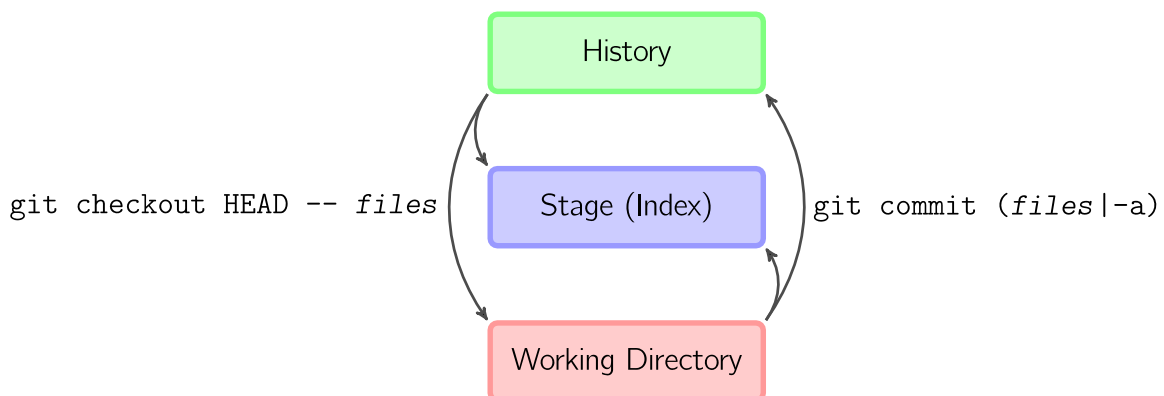
- в індексі (stage, index – файл відстежується Git, проте ще не доданий до локального репозиторію;
- в історії (history) – відстежується в локальному репозиторії.

Черговість команд на постановку для відстеження файлів чи їх отримання має такий вигляд:



- `git add файли` копіює *файли* в їх поточному стані в індекс.
- `git commit` зберігає знімок індексу в вигляді комміту.
- `git reset -- файли` відновлює файли в індексі, а саме, копіює їх з останнього комміту в індекс. Використовуйте цю команду для відміни змін, внесених командою `git add`. Також можна виконувати `git reset`, щоб відновити всі файли в індексі.
- `git checkout -- файли` копіює *файли* з індексу в робочу папку. Цю команду зручно використовувати, щоб відхилити небажані зміни в робочій директорії.

Також можна перескочити через індекс та відразу отримати файли з історії прямо в робочу папку або зробити комміт, оминаючи індекс:



- `git commit -a` аналогічний до запуску двох команд: `git add` для всіх файлів, які існували в попередньому комміті, та `git commit`.

- `git commit файли` створює новий комміт, в основі якого лежать уже існуючі файли, додаючи зміни тільки для вказаних файлів. Одночасно вказані файли будуть скопійовані в індекс.
- `git checkout HEAD -- файли` копіює файли з поточного комміту і в індекс, і в робочу папку.

Виконаємо наступні дії: додамо наш новий файл в індекс, а потім здійснимо фіксацію (комміт) змін:

```
puasson@HOME-PC MINGW64 /f/myrepo (master)
$ git add .

puasson@HOME-PC MINGW64 /f/myrepo (master)
$ git commit -m "Мій перший комміт"
[master (root-commit) 5d43a55] 1 file changed, 1 insertion(+)
create mode 100644 README.txt
```

Командний рядок має проблеми з підтримкою кирилиці, тому бажано записувати повідомлення (message) з описом комміту латинкою. Крапка в команді `git add` означає додавання поточної папки в індекс для відстеження. Ключ `-m` для команди `git commit` надає можливість включити опис здійснених в комміті змін у вигляді одного або кількох рядків тексту.

Написання повідомлення для комміту має кілька особливостей. Представлене в прикладі повідомлення погане, оскільки воно не описує дії, здійснені в комміті (додавання нового файлу). Текст самого повідомлення записується від імені комміту, а не розробника, причому в теперішньому часі: *виправляє баг ...*, *реалізує ...* тощо. Якщо потрібний перелік здійснених змін у комміті, рекомендується записувати його з маркерами `*` або `-`. Також можуть записуватись номери багів або спеціальні для організації скорочення. До прикладу, можуть додаватись позначки типу `[css, js]`, щоб уточнити мови чи технології, використані для змін. Для виправлення помилок попереду дописувати `“bugfix: ”` та ін.

Поганий комміт: `“Fix typo”` (виправити друкарську помилку).

Хороший комміт: `“Add missing > in project section of HTML”` (додати відсутній тег у розділі проектів HTML).

Визначити стан файлів допомагає команда `git status`. Якщо немає незафіксованих файлів, результат може бути подібним до такого:

```
puasson@HOME-PC MINGW64 /f/myrepo (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Припустимо, було додано файл README (тут це зроблено з консолі за допомогою команди touch). Git вкаже на появу файлу, якого раніше не було і який на даний момент не відстежується:

```
puasson@HOME-PC MINGW64 /f/myrepo (master)
$ touch README

puasson@HOME-PC MINGW64 /f/myrepo (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    README

nothing added to commit but untracked files present (use "git add" to track)
```

Додамо файл в індекс та ще раз переглянемо статус репозиторія:

```
puasson@HOME-PC MINGW64 /f/myrepo (master)
$ git add README

puasson@HOME-PC MINGW64 /f/myrepo (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   README
```

Проіндексований файл знаходиться в секції “Changes to be committed”, а його назва підсвічується зеленим кольором. Якщо ви виконаєте комміт в цей момент, то версія файлу, що існувала на момент виконання вами команди git add, буде додана в історію знімків стану.

```
puasson@HOME-PC MINGW64 /f/myrepo (master)
$ git commit -m "Додає файл README"
[master 617cb90] 1 file changed, 1 insertion(+), 0 deletions(-)
 create mode 100644 README

puasson@HOME-PC MINGW64 /f/myrepo (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Вивести повне дерево файлів, які відстежує Git, допоможе команда git ls-tree. Не задавши їй аргументів, отримаємо довідку з приводу можливих прапорців для запуску. Наприкінці потрібно задати назву вітки, тут – показник HEAD.

```
MINGW64/f/myrepo
puasson@HOME-PC MINGW64 /f/myrepo (master)
$ git ls-tree
usage: git ls-tree [<options>] <tree-ish> [<path>...]

    -d                only show trees
    -r                recurse into subtrees
    -t                show trees when recursing
    -z                terminate entries with NUL byte
    -l, --long        include object size
    --name-only        list only filenames
    --name-status      list only filenames
    --full-name        use full path names
    --full-tree        list entire tree; not just current directory (implies
--full-name)
    --abbrev[=<n>]    use <n> digits to display SHA-1s

puasson@HOME-PC MINGW64 /f/myrepo (master)
$ git ls-tree --name-only --full-tree HEAD
README
README.txt
```

Для видалення файлу з Git потрібно видалити цей файл з індексу (переліку відстежуваних файлів), а потім виконати комміт. У даному випадку допоможе команда `git rm`, яка також видаляє файл з робочого каталогу, тому надалі ви не побачите його як «невідстежуваний».

Якщо ви просто видалите файл зі свого робочого каталогу, він буде показаний в секції "Changes not staged for commit" (змінені, але не проіндексовані) виведення команди `git status`:

```
puasson@HOME-PC MINGW64 /f/myrepo (master)
$ rm README

puasson@HOME-PC MINGW64 /f/myrepo (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    deleted:    README

no changes added to commit (use "git add" and/or "git commit -a")
```

Якщо після цього виконати команду `git rm`, його видалення потрапить в індекс:

```
puasson@HOME-PC MINGW64 /f/myrepo (master)
$ git rm README
rm 'README'

puasson@HOME-PC MINGW64 /f/myrepo (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    deleted:    README
```



Після наступного комміта файл зникне і більше не буде відстежуватися. Якщо ви змінили файл і вже проіндексували його, ви повинні використовувати примусове видалення за допомогою параметра `-f`. Це зроблено для підвищення безпеки, щоб запобігти помилковому видаленню даних, які ще не були записані в знімок стану і які не можна відновити з Git.

Переміщення файлів Git розглядає як їх перейменування.

```
puasson@HOME-PC MINGW64 /f/myrepo (master)
$ mkdir subfolder2

puasson@HOME-PC MINGW64 /f/myrepo (master)
$ git add subfolder2

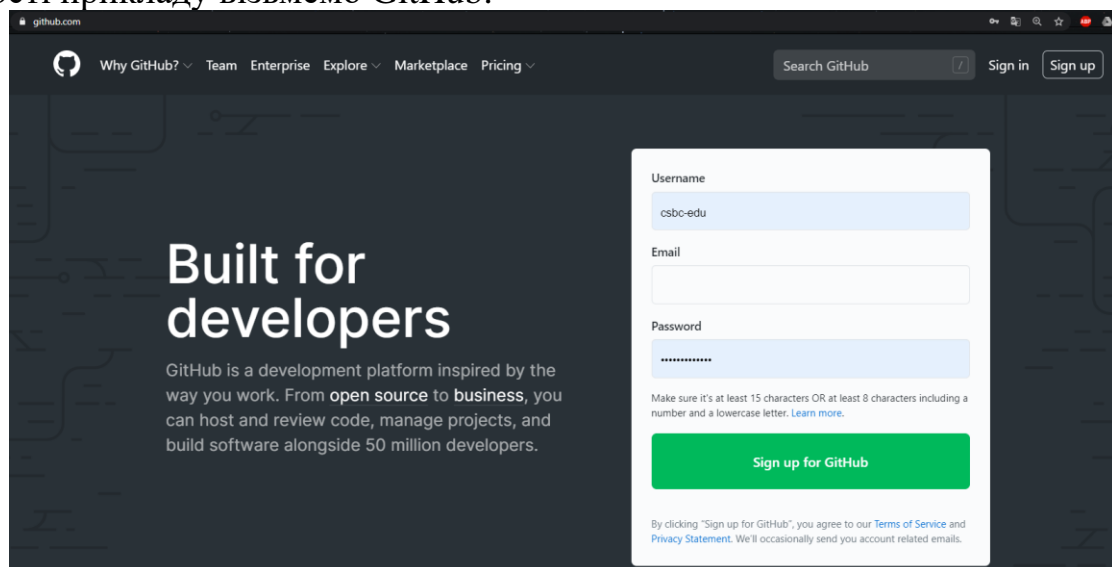
puasson@HOME-PC MINGW64 /f/myrepo (master)
$ git mv README.txt subfolder2

puasson@HOME-PC MINGW64 /f/myrepo (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:      README
        renamed:      README.txt -> subfolder2/README.txt
```

Після фіксації змін (комміту) заново виведіть перелік відстежуваних файлів.

### ***Етап 3. Відправка фіксованих змін у віддалений репозиторій.***

Сервіси хостингу репозиторіїв, на зразок GitHub або BitBucket, дозволяють формувати віддалений репозиторій, доступ до якого може отримувати ціла група розробників. З цією метою слід зареєструвати обліковий запис відповідного хостингу. Цей обліковий запис буде використовуватись протягом усього курсу. У якості прикладу візьмемо GitHub:





Натисніть кнопку Sign Up для реєстрації облікового запису, якщо досі не маєте такого ж або подібного. Після цього введіть потрібні дані та виконайте реєстрацію на сервісі.

## Create your account

Username \*

Email address \*

Password \*

Make sure it's at least 15 characters OR at least 8 characters including a number and a lowercase letter. [Learn more.](#)

Email preferences  
☐ Send me occasional product updates, announcements, and offers.

Verify your account


Пожалуйста, решите эту задачу, чтобы мы знали, что вы реальный человек

[Проверить](#)

При першому вході відразу доступне створення нового віддаленого репозиторію, куди буде потрібно відправити зафіксовані до цього зміни:

### Create a new repository


A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)


Owner \*  
 csbc-edu /

Repository name \*

Great repository names are short and memorable. Need inspiration? How about **bookish-system**?

Description (optional)

☒  **Public**  
Anyone on the internet can see this repository. You choose who can commit.

☐  **Private**  
You choose who can see and commit to this repository.





Initialize this repository with:  
Skip this step if you're importing an existing repository.

☐ **Add a README file**  
This is where you can write a long description for your project. [Learn more.](#)

☐ **Add .gitignore**  
Choose which files not to track from a list of templates. [Learn more.](#)

☐ **Choose a license**  
A license tells others what they can and can't do with your code. [Learn more.](#)

Quick setup — if you've done this kind of thing before

 Set up in Desktop or 
  HTTPS  SSH 
 


Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

---

...or create a new repository on the command line

```

echo "# test-repo" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M master
git remote add origin https://github.com/csbc-edu/test-repo.git
git push -u origin master
    
```

---

...or push an existing repository from the command line










```

git remote add origin https://github.com/csbc-edu/test-repo.git
git branch -M master
git push -u origin master
    
```

GitHub пропонує набір альтернативних команд для створення нового репозиторію чи відправки своєї локальної версії проекту в новостворений репозиторій. *Додайте свій локальний репозиторій до новоствореного віддаленого репозиторію та зробіть скриншот із заповненим репозиторієм на GitHub.*

**Етап 4. Робота з віддаленими репозиторіями.** Спробуємо налагодити роботу зі спільним репозиторієм. *Об'єднайтесь навколо одного репозиторія командами по 2-3 особи та організуйте доступ команди до нього.* На прикладі GitHub-репозиторія розгляньте процес додавання співавторів для управління репозиторієм:

csbc-edu / csbc-edu.github.io

 Code 
  Issues 
  Pull requests 
  Actions 
  Projects 
  Wiki 
  Security 
  Insights 
  Settings

Options
Manage access
Security & analysis
Branches
Webhooks
Notifications
Integrations
Deploy keys
Secrets
Actions
Moderation
Interaction limits

Who has access

PUBLIC REPOSITORY


This repository is public and visible to anyone.

[Manage](#)

DIRECT ACCESS

0 collaborators have access to this repository. Only you can contribute to this repository.

Manage access

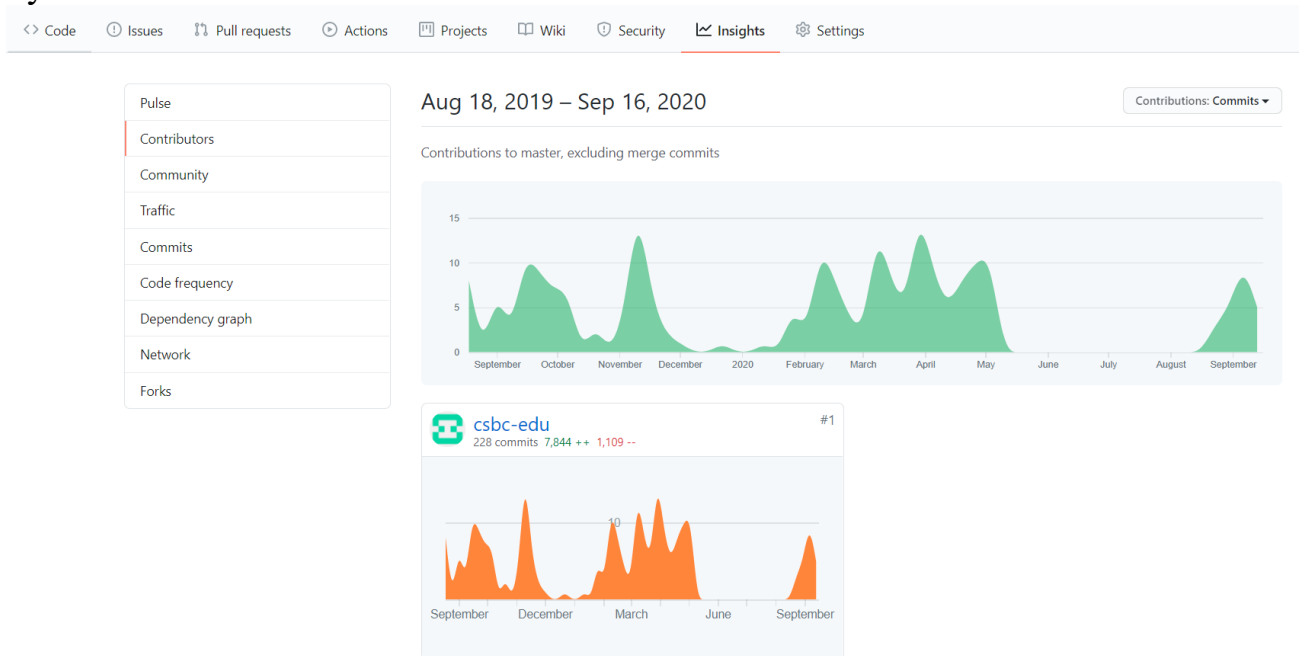


You haven't invited any collaborators yet

[Invite a collaborator](#)

Запросіть команду до спільного внесення змін у наповнення репозиторія.

Динаміку внесених змін до проекту можна побачити на вкладці Insights, пункт Collaborators



### Завдання 1.3.

*Створіть власний репозиторій та повторіть вище зазначені операції. Додайте скриншоти виконання команд у консолі до звіту.*

<https://marklodato.github.io/visual-git-guide/index-ru.html>

### **Ефективна командна робота за допомогою Git**

Коли ви робите комміт, Git зберігає його у вигляді об'єкта, який містить покажчик на знімок (snapshot) підготовлених даних. Цей об'єкт так само містить ім'я автора та email, повідомлення і покажчик на комміт або комміти, що безпосередньо передують даному (його батьків): відсутність батька для початкового комміта, один батько для звичайного комміта, і кілька батьків для результатів злиття двох і більше віток.

Припустимо, у вашій робочій директорії є три файли і ви додасте їх всі в індекс і створюєте комміт. Під час індексації обчислюється контрольна сума кожного файлу (SHA-1), потім кожен файл зберігається в репозиторій (Git називає такий файл blob – великий бінарний об'єкт), а контрольна сума потрапить в індекс:

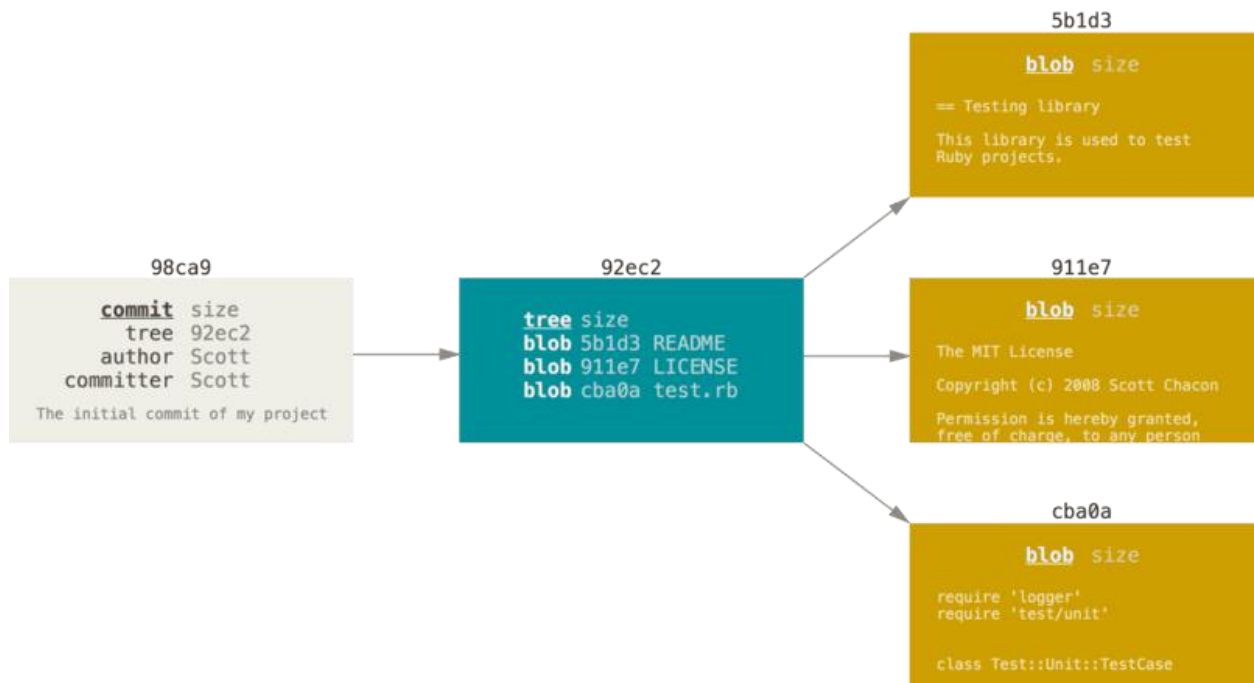
```
$ git add README test.rb LICENSE
```

```
$ git commit -m 'initial commit of my project'
```

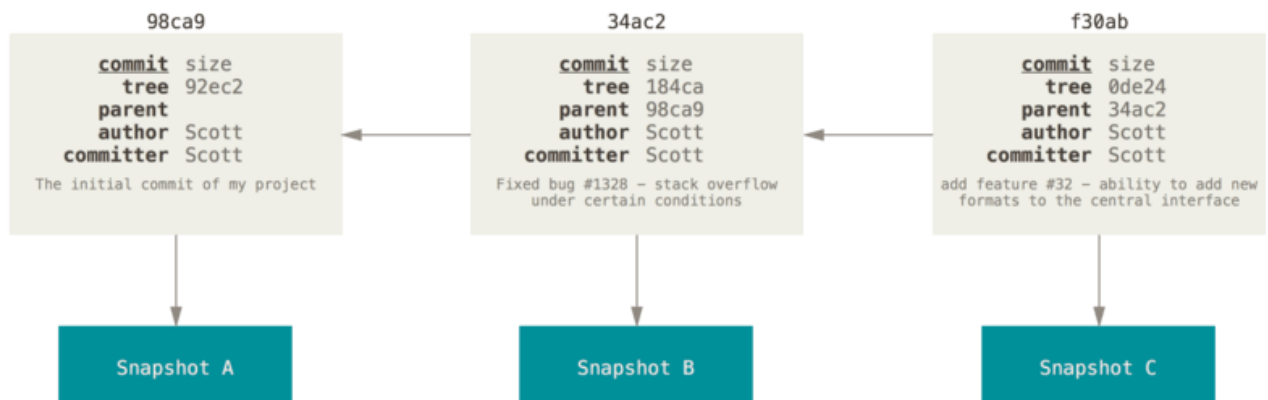
Коли ви створюєте комміт командою `git commit`, Git обчислює контрольні суми кожного підкаталогу (в нашому випадку, тільки основний каталог проекту) і зберігає його в репозиторії як об'єкт дерева каталогів. Потім Git створює об'єкт комміта з метаданими і покажчиком на основне дерево проекту для можливості відтворити цей знімок в разі потреби.

Ваш репозиторій Git тепер зберігає п'ять об'єктів:

- **три blob-об'єкта** (по одному на кожен файл),
- **об'єкт дерева** каталогів, що містить список файлів і відповідних їм blob-ів,
- **об'єкт комміта**, що містить метадані та покажчик на об'єкт дерева каталогів.

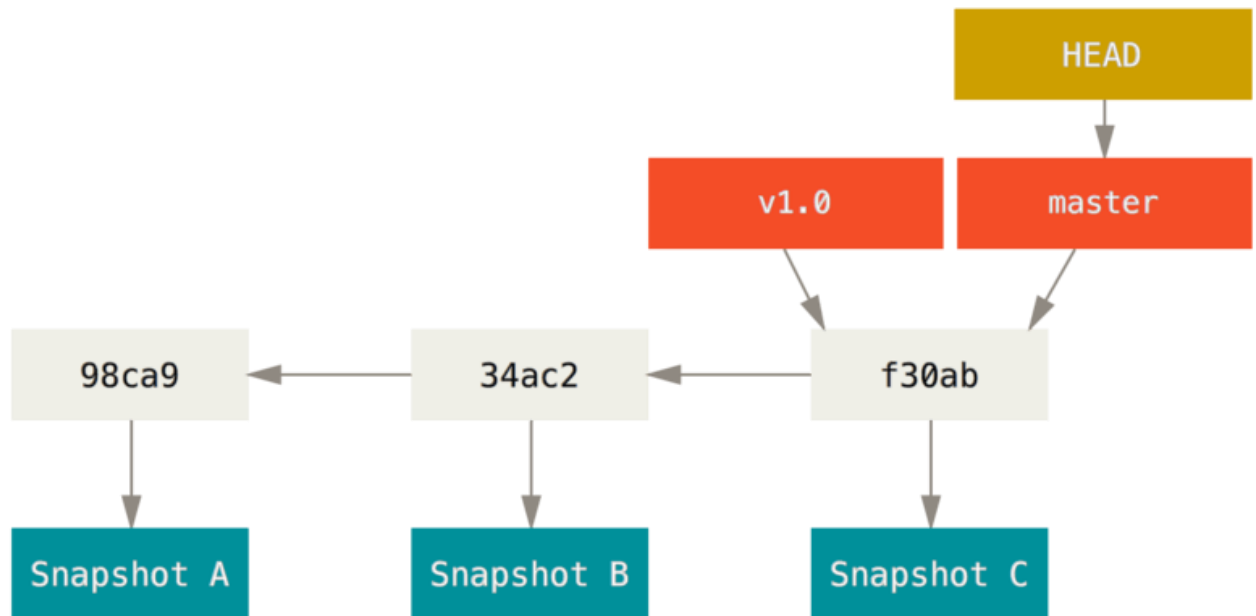


Якщо ви зробите зміни і створите ще один комміт, то він буде містити покажчик на попередній комміт.



Вітка в Git – це простий переміщуваний покажчик на один з таких коммітів. За умовчанням, назва основної вітки в Git – `master`. Як тільки ви почнете

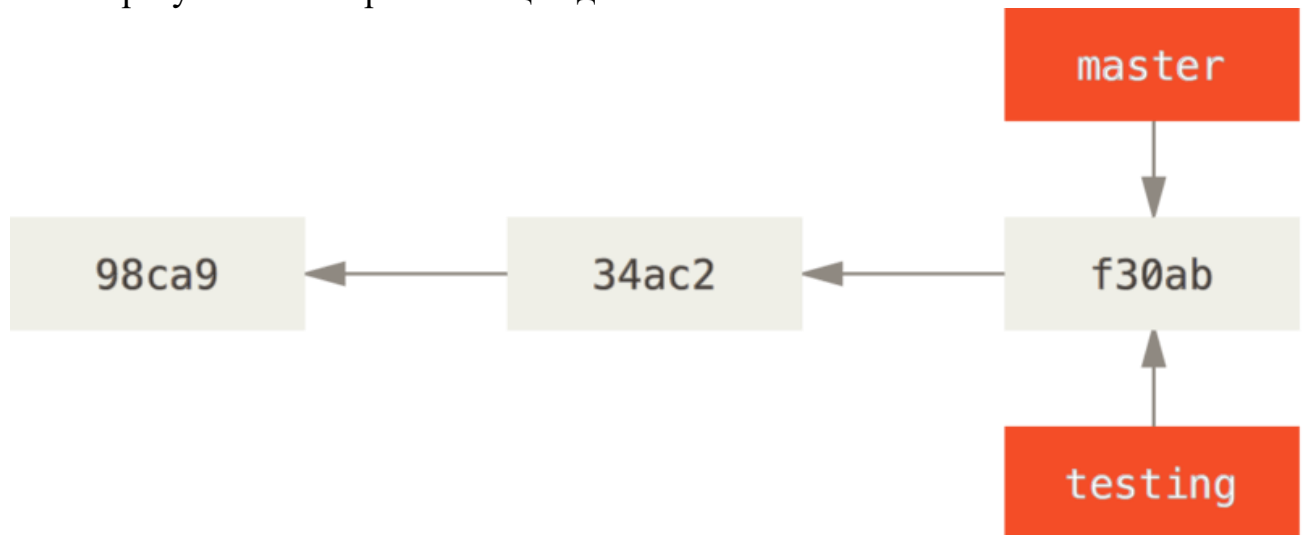
створювати комміти, вітка `master` буде завжди вказувати на останній комміт. Кожен раз при створенні коммітів показчик вітки `master` буде пересуватися на наступний комміт автоматично.



При створенні нової вітки відбувається створення лише нового показчика для подальшого переміщення. Нехай потрібно створити вітку `testing`. Ви можете зробити це за допомогою команди `git branch`:

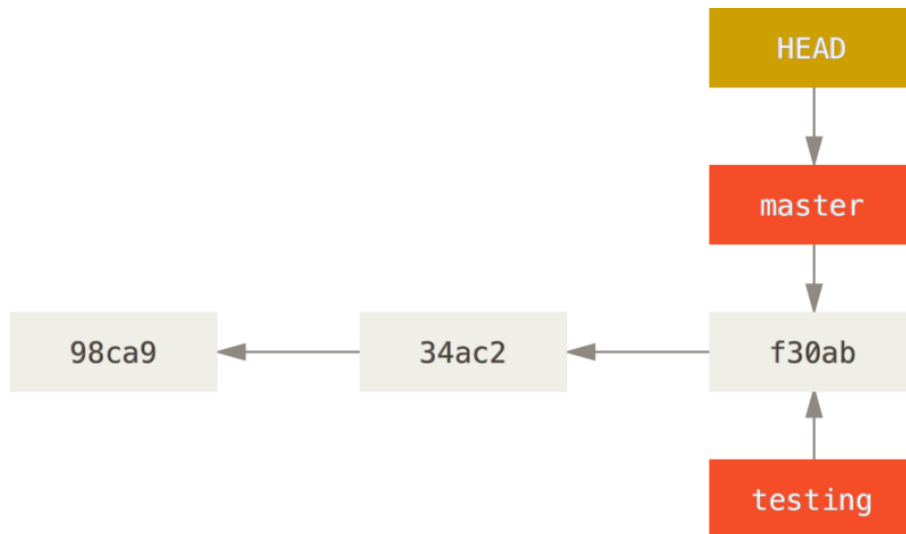
```
$ git branch testing
```

У результаті створюється ще один показчик на поточний комміт.



Як Git визначає, в який гілці ви перебуваєте? Він зберігає спеціальний показчик `HEAD` на поточну локальну вітку. У нашому випадку ми все ще

перебуваємо в гілці master. Команда `git branch` тільки створює нову вітку, але не перемикає на неї.



Ви можете легко це побачити за допомогою простої команди `git log`, яка покаже вам куди вказують покажчики віток. Ця опція називається `--decorate`.

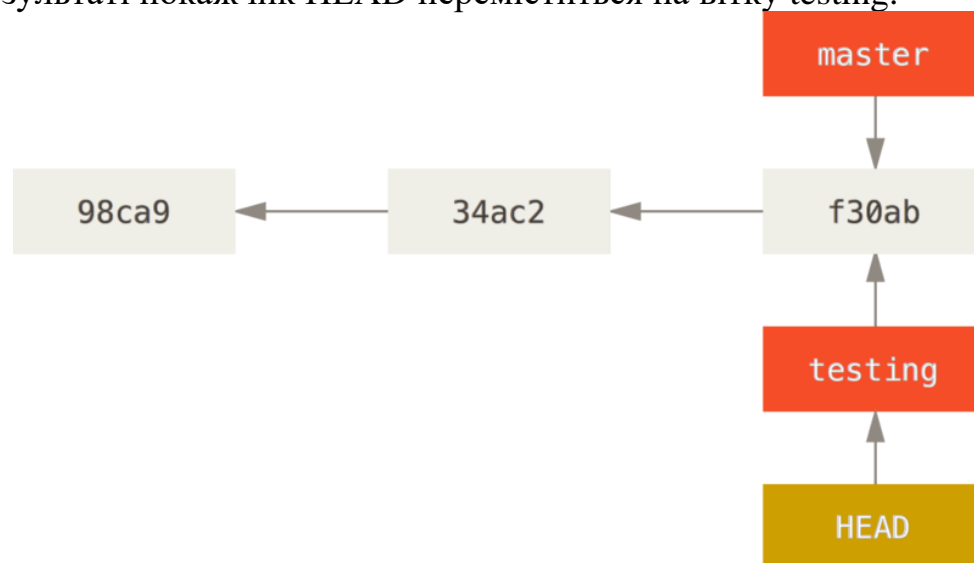
```
$ git log --oneline --decorate
```

```
f30ab (HEAD, master, testing) add feature #32 - ability to add new  
34ac2 fixed bug #1328 - stack overflow under certain conditions  
98ca9 initial commit of my project
```

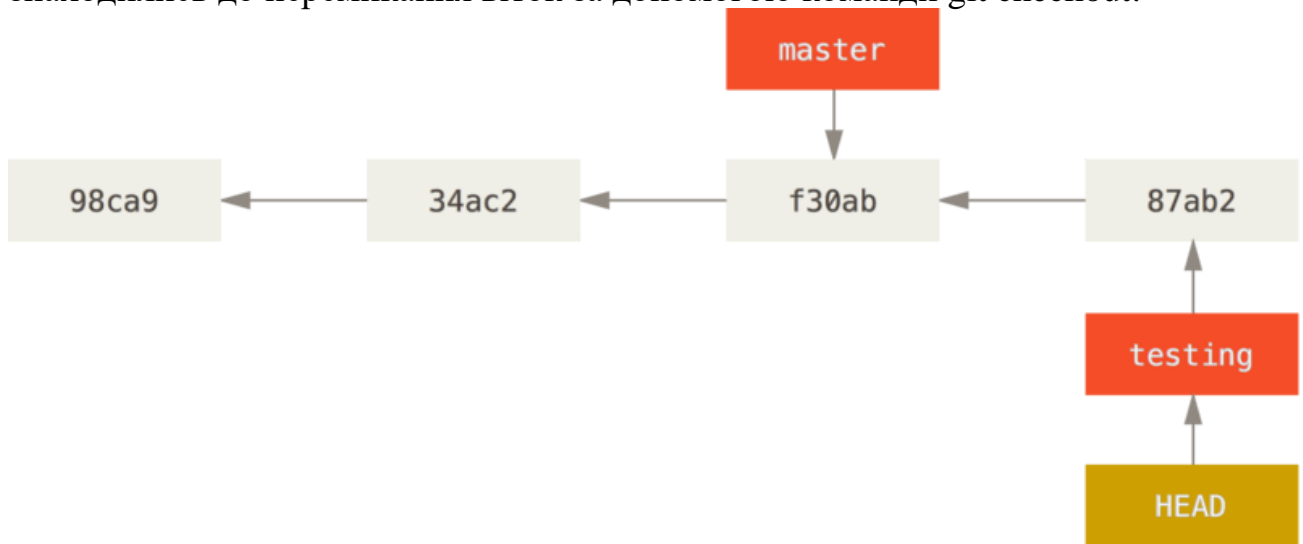
Для перемикання на існуючу вітку виконайте команду `git checkout`. Давайте перемкнемося на вітку `testing`:

```
$ git checkout testing
```

У результаті покажчик `HEAD` переміститься на вітку `testing`.



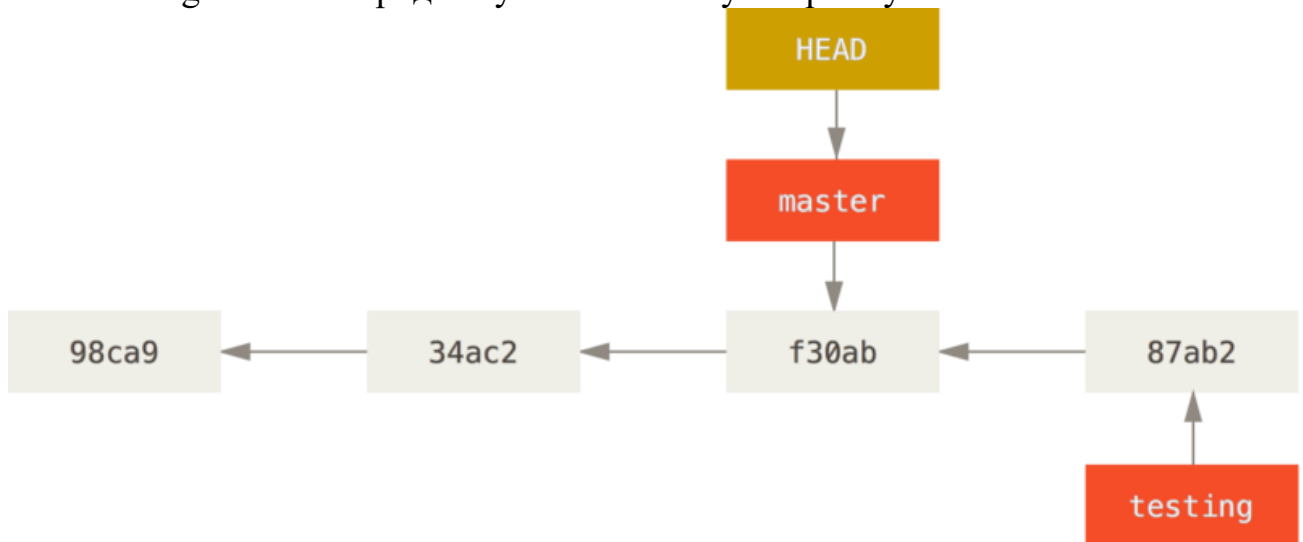
Після здійснення наступного комміту можна побачити, що покажчик на вітку `testing` зміститься вперед, а `master` вказуватиме все на той же комміт, де ви знаходились до перемикання віток за допомогою команди `git checkout`.



Спробуємо переключитись на вітку `master`:

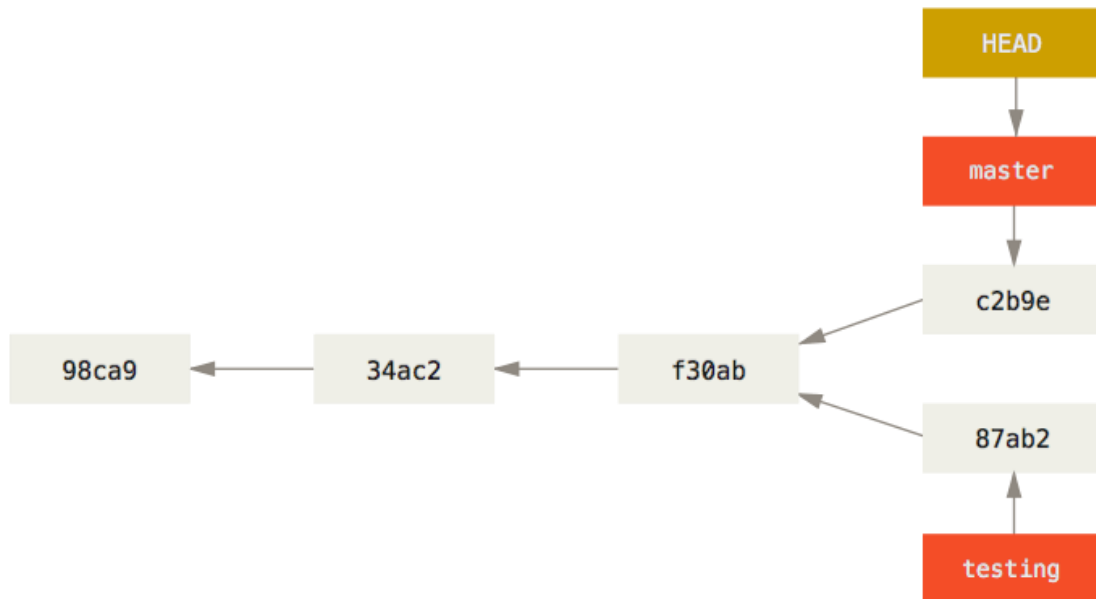
```
$ git checkout master
```

Ця команда зробила дві речі: перемістила покажчик `HEAD` назад на вітку `master` і повернула файли в робочому каталозі в той стан, на знімок якого вказує `master`. Це також означає, що всі внесені з цього моменту зміни будуть відноситись до старої версії проекту. Іншими словами, ви відкотили всі зміни вітки `testing` і можете продовжувати в іншому напрямку.



Черговий комміт утворить розгалужену історію:

```
$ vim test.rb  
$ git commit -a -m 'made other changes'
```



Ви створили вітку і переключилися на неї, попрацювали, а потім повернулися в основну вітку і попрацювали в ній. Ці зміни ізольовані один від одної: ви можете вільно перемикаватися туди і назад, а коли знадобиться – об'єднати їх. І все це робиться простими командами: `branch`, `checkout` і `commit`.

Вітка в Git – це простий файл, який містить 40 символів контрольної суми SHA-1 коммітов, на який вона вказує; тому операції з вітками є дешевими з точки зору споживання ресурсів або часу. Створення нової вітки в Git відбувається так само швидко і просто, як запис 41 байта в файл (40 знаків і символ переходу на новий рядок).

[https://git-scm.com/book/ru/v2/%D0%92%D0%B5%D1%82%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5-%D0%B2-Git-%D0%9E-%D0%B2%D0%B5%D1%82%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B8-%D0%B2-%D0%B4%D0%B2%D1%83%D1%85-%D1%81%D0%BB%D0%BE%D0%B2%D0%B0%D1%85#rdivergent\\_history](https://git-scm.com/book/ru/v2/%D0%92%D0%B5%D1%82%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5-%D0%B2-Git-%D0%9E-%D0%B2%D0%B5%D1%82%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B8-%D0%B2-%D0%B4%D0%B2%D1%83%D1%85-%D1%81%D0%BB%D0%BE%D0%B2%D0%B0%D1%85#rdivergent_history)

**Простий робочий процес на базі віток і галуження.** Нехай робота побудована так:

1. Ви працюєте над сайтом.
2. Ви створюєте вітку для нової статті, яку ви пишете.
3. Ви працюєте в цій вітці.

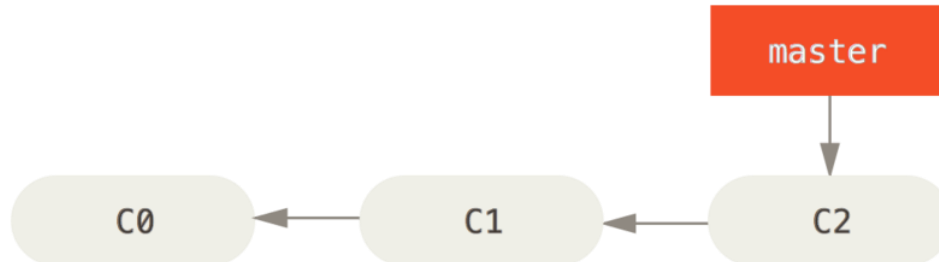
У цей момент ви отримуєте повідомлення, що виявлена критична помилка, яка потребує якнайшвидшого виправлення. Ваші дії:

1. Переключитися на основну вітку.
2. Створити вітку для додавання виправлення.



3. Після тестування злити (merge) вітку, яка містить виправлення, з основною віткою.
4. Переключитися назад в ту вітку, де ви пишете статтю і продовжити працювати.

Нехай Ви працюєте над проектом та вже маєте кілька коммітів:

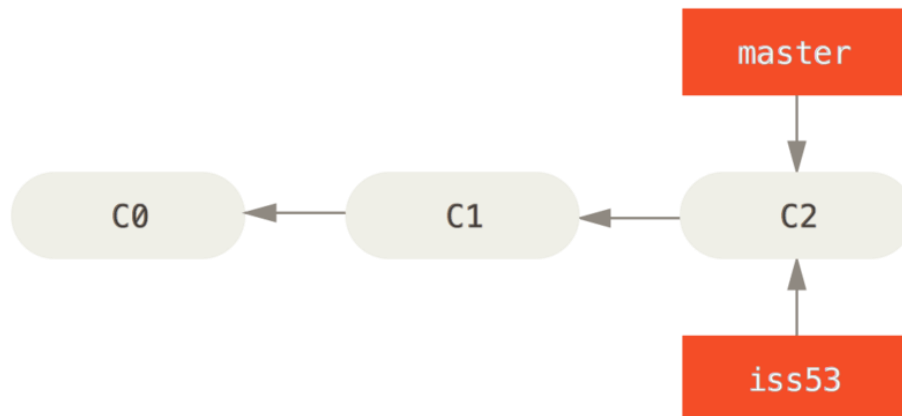


Ви вирішуєте, що тепер ви будете займатися проблемою # 53 з вашої системи відстеження помилок (баг-трекера). Щоб створити гілку і відразу переключитися на неї, можна виконати команду `git checkout` з параметром `-b`:

```
$ git checkout -b iss53  
Switched to a new branch "iss53"
```

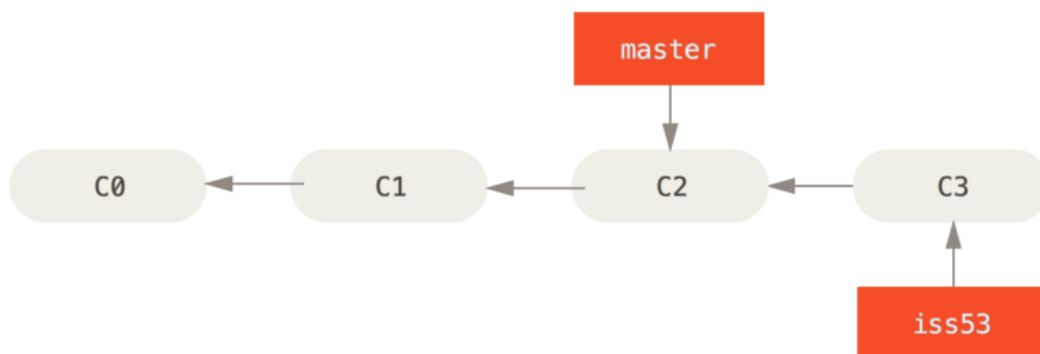
Це те ж саме, що й

```
$ git branch iss53  
$ git checkout iss53
```



Ви працюєте над своїм сайтом і робите комміти. Це призводить до того, що вітка `iss53` рухається вперед, так як ви переключилися на неї раніше (HEAD вказує на неї).

```
$ vim index.html  
$ git commit -a -m 'added a new footer [issue 53]'
```



Тут ви отримуєте повідомлення про виявлення вразливості на вашому сайті, яку потрібно негайно усунути. Завдяки Git, не потрібно розмішувати це виправлення разом з тим, що ви зробили в `iss53`. Вам навіть не доведеться докладати зусиль, щоб відкотити всі ці зміни для початку роботи над виправленням. Все, що вам потрібно – переключитися на гілку `master`.

Але перед тим як зробити це – майте на увазі, що якщо ваш робочий каталог або область підготовлених файлів містять зміни, що не потрапили в комміт і конфліктують з віткою, на яку ви хочете перейти, то Git не дозволить вам перемкнутися на вітку. Найкраще перемикаєтесь з чистого робочого стану проекту. Є способи обійти це (заховати ([stash](#)) або виправити (`amend`) комміти).

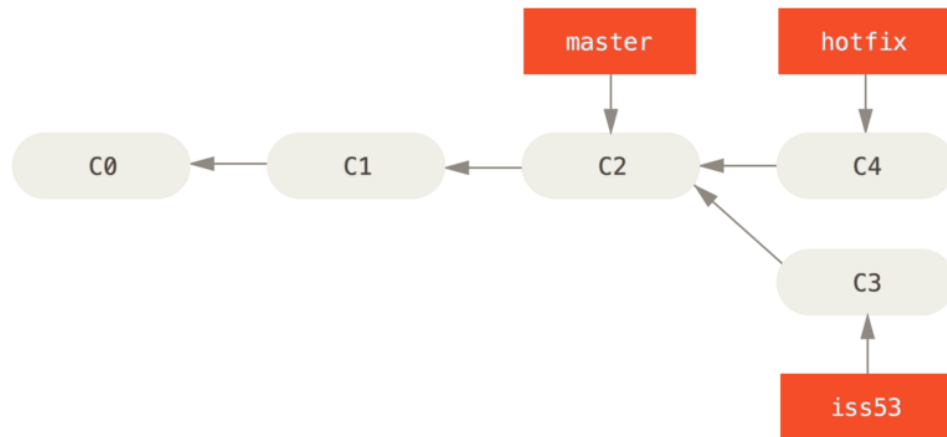
Тепер припустимо, що ви зафіксували всі свої зміни і можете переключитися на вітку `master`:

```
$ git checkout master
Switched to branch 'master'
```

З цього моменту ваш робочий каталог має точно такий же вигляд, який був перед початком роботи над проблемою # 53, і ви можете зосередитися на роботі над виправленням. Важливо запам'ятати: коли ви перемикаєте вітку, Git повертає стан робочого каталогу до того виду, який він мав у момент останнього комміту в цю вітку. Він додає, видаляє і змінює файли автоматично, щоб стан робочого каталогу відповідав тому, коли був зроблений останній комміт.

Тепер ви можете перейти до написання виправлення. Давайте створимо нову гілку для виправлення, в якій будемо працювати, поки не закінчимо виправлення.

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 1fb7853] fixed the broken email address
1 file changed, 2 insertions(+)
```

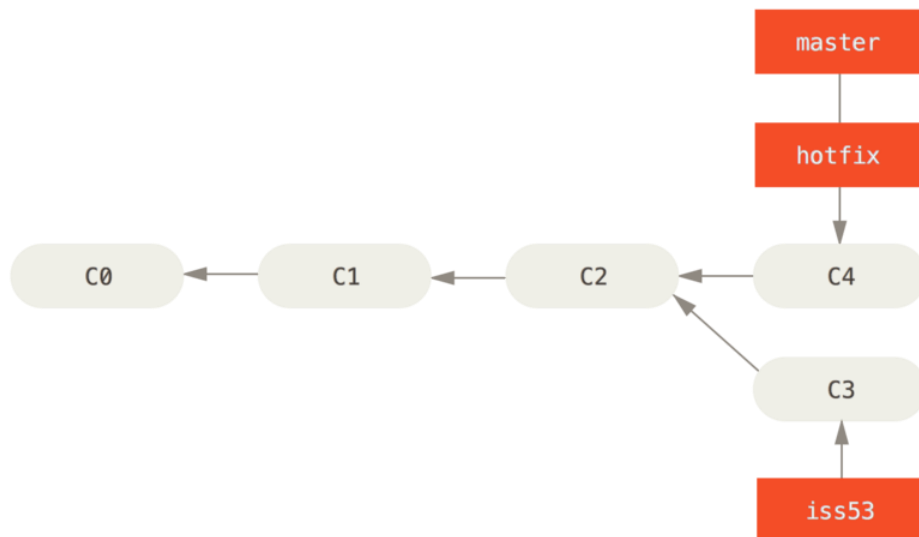


Якщо ваше виправлення вирішує проблему, слід виконати злиття віток hotfix та master для включення змін у продукт. Це робиться за допомогою команди git merge:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

Помітили фразу "fast-forward" у цьому злитті? Git просто перемістив покажчик вітки вперед, тому що комміт C4, на який вказує злита вітка hotfix, був прямим нащадком комміту C2, на якому ви перебували до цього. Іншими словами, якщо комміт зливається з тим, до якого можна дістатися рухаючись по історії прямо, Git спрощує злиття просто переносючи покажчик вітки вперед, оскільки немає розбіжностей у змінах. Це називається "fast-forward".

Тепер ваші зміни включені в комміт, на який вказує вітка master, і виправлення можна впроваджувати.

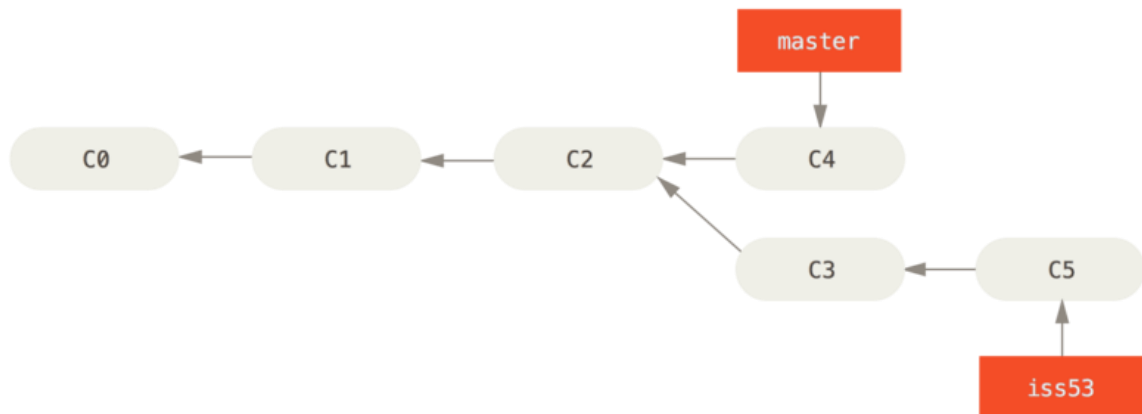


Після впровадження вашого надважливого виправлення ви готові повернутися до роботи над тим, що були змушені відкласти. Але спочатку потрібно видалити гілку hotfix, тому що вона більше не потрібна – вітка master вказує на те ж саме місце. Для видалення вітки виконайте команду git branch з параметром -d:

```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

Тепер ви можете переключитися назад на гілку iss53 і продовжити роботу над проблемою # 53:

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
1 file changed, 1 insertion(+)
```

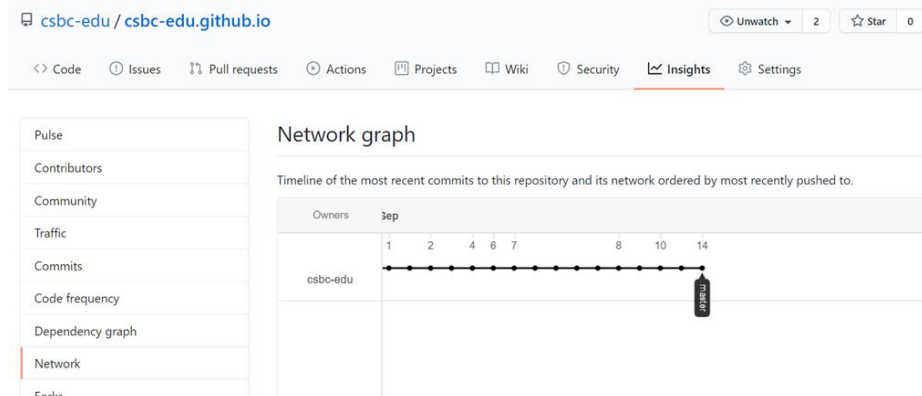


Варто звернути увагу на те, що всі зміни з вітки hotfix не включені в вашу вітку iss53. Якщо їх потрібно включити, ви можете влити вітку master у вашу вітку iss53 командою git merge master, або ж ви можете відкласти злиття цих змін до завершення роботи, і потім влити гілку iss53 в master.



#### Завдання 1.4.

*Повторіть подібний сценарій у своєму репозиторії та відобразіть у звіті візуалізацію коммітів, доступну на вкладці Insights:*

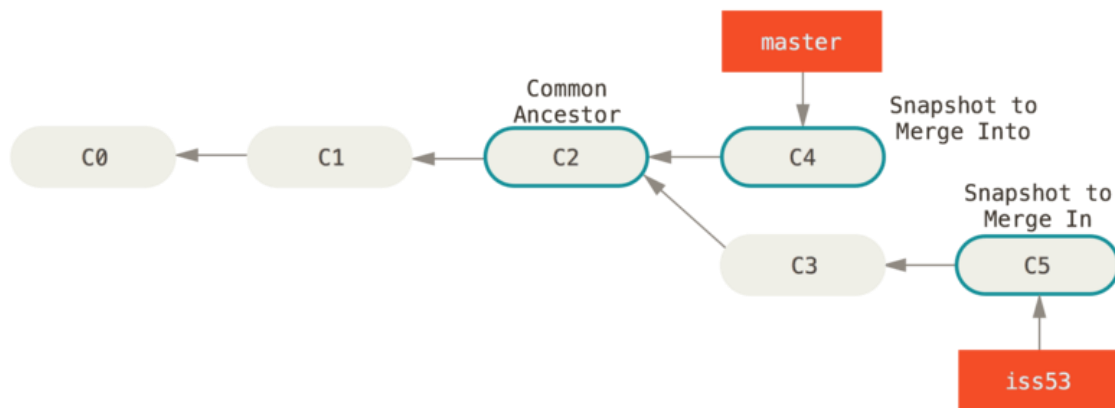


	Обов'язково зробіть скриншот після злиття віток, проте перед видалення непотрібної вітки.
--	---

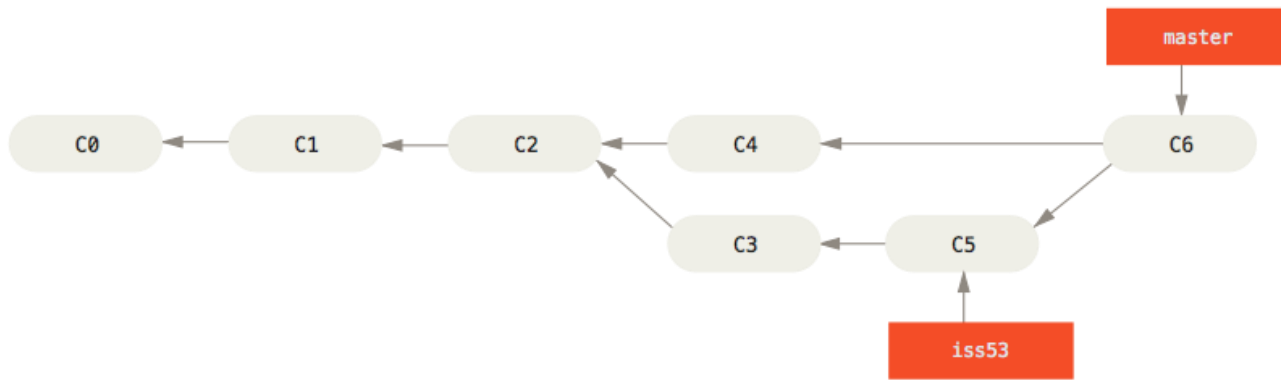
**Основи злиття.** Припустимо, ви вирішили, що робота з проблеми # 53 закінчена і її можна влити в вітку master. Для цього потрібно виконати злиття вітки iss53 точно так же, як ви робили це з віткою hotfix раніше. Все, що потрібно зробити, – перейти на вітку, в яку ви хочете включити зміни, і виконати команду `git merge`:

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |      1 +
1 file changed, 1 insertion(+)
```

Результат цієї операції відрізняється від результату злиття вітки hotfix. В даному випадку процес розробки відгалузився в більш ранній точці. Оскільки коміт, на якому ми знаходимося, не є прямим батьком вітки, з якої ми виконуємо злиття, Git доведеться трохи попрацювати. В цьому випадку Git виконує просте тристороннє злиття, використовуючи останні коміти віток, які об'єднуються, і спільного для них батьківського коміта.



Замість того, щоб просто пересунути покажчик вітки вперед, Git створює новий результуючий знімок тристороннього злиття, а потім автоматично робить коміт. Цей особливий коміт називають **комітом злиття**, так як у нього більше одного предка.



Тепер, коли зміни злиті, вітка iss53 більше не потрібна. Ви можете закрити завдання в системі відслідковування помилок і видалити вітку:

```
$ git branch -d iss53
```



#### Завдання 1.5.

*Змодельуйте поширені конфліктні ситуації та корисні практики при командній роботі з документами та відобразіть їх результати в звіті. Для цього об'єднайтесь у групи по двоє-троє осіб та виконайте нижче зазначені завдання з виділеними курсивом умовами.*

**2.1. Конфлікт: Pulling with Untracked Changes.** Відбувається управління одним файлом для двох користувачів.

Користувач 1: здійснив зміни в файлі та виконав коміт у вітку master.

Користувач 2: спочатку вніс та локально зберіг свої зміни у файл (версія без змін користувача 1), а потім вирішив здійснити пулінг з центрального репозиторію.

```

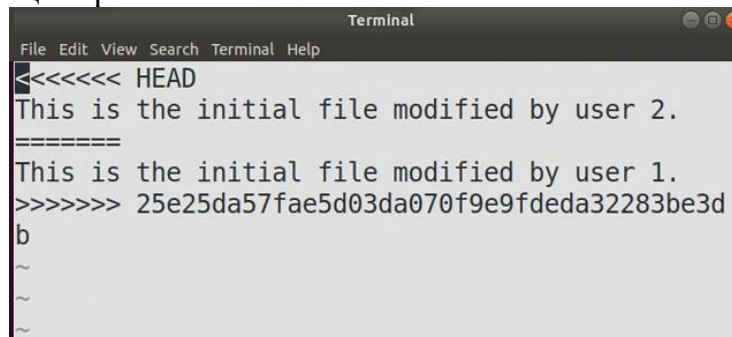
File Edit View Search Terminal Help
user2: vi demo
user2: git pull origin master
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /home/course/Desktop/repos/remote/git-demo
* branch          master      -> FETCH_HEAD
   02558b6..25e25da master      -> origin/maste
r
Updating 02558b6..25e25da
error: Your local changes to the following file
s would be overwritten by merge:
    demo
Please commit your changes or stash them before
you merge.
Aborting
user2: █
  
```

Нові користувачі вважають, що можуть здійснювати пулінг у будь-який момент. Для цього потрібно мати *або* чисту робочу папку (команда `git reset --hard`), *або* зафіксувати (коміт) свої зміни та виконати пулінг

```
user2: git commit -am "User 2 changes"
[master 1e5d508] User 2 changes
 1 file changed, 1 insertion(+), 1 deletion(-)
user2: git pull origin master
From /home/course/Desktop/repos/remote/git-demo
 * branch                master       -> FETCH_HEAD
Auto-merging demo
CONFLICT (content): Merge conflict in demo
Automatic merge failed; fix conflicts and then
commit the result.
```

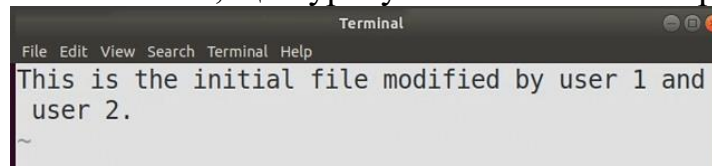
З'явиться конфлікт злиття у результаті проблем додавання локальних змін двома користувачами в один файл.

**Вирішення конфлікту.** Відкрити проблемний файл. Git додав метадані з описом конфлікту в цей файл



```
Terminal
File Edit View Search Terminal Help
<<<<<<< HEAD
This is the initial file modified by user 2.
=====
This is the initial file modified by user 1.
>>>>>>> 25e25da57fae5d03da070f9e9fdeda32283be3d
b
~
~
~
```

Виправте файл таким чином, щоб урахувати зміни обох користувачів



```
Terminal
File Edit View Search Terminal Help
This is the initial file modified by user 1 and
user 2.
~
~
```

Далі ці зміни фіксуються та пушуться

```
user2: git commit -am "Resolved merge conflict"
[master 6390d0e] Resolved merge conflict
user2: git push origin master
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (6/6), 563 bytes | 563.00
KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To /home/course/Desktop/repos/remote/git-demo/
 25e25da..6390d0e master -> master
```

**2.2. Конфлікт: *Forced Pushes*.** Користувач 2 створює новий, фіксує та пушить файл у центральний репозиторій:



```

Terminal
File Edit View Search Terminal Help
user2: touch demo2
user2: git add .
user2: git commit -m "User 2 added file"
[master 8c18e8e] User 2 added file
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 demo2
user2: git push origin master
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 257 bytes | 257.00
KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /home/course/Desktop/repos/remote/git-demo/
6390d0e..8c18e8e master -> master

```

Користувач 1 у цей час бажає внести зміни в інший, раніше створений файл (тут – demo)

```

Terminal
File Edit View Search Terminal Help
This is the initial file modified by user 1 add
some changes to the demo file.
~
~

```

та зафіксувати їх у центральному репозиторії. Git попередить, що в цей час зміни були внесені в центральний репозиторій, а в локальній версії Користувача 1 їх немає.

```

Terminal
File Edit View Search Terminal Help
user1: vi demo
user1: git commit -am "user 1 commits"
[master b0416e5] user 1 commits
1 file changed, 1 insertion(+), 1 deletion(-)
user1: git push origin master
To /home/course/Desktop/repos/remote/git-demo
! [rejected] master -> master (fetch first)
error: failed to push some refs to '/home/course/Desktop/repos/remote/git-demo'
hint: Updates were rejected because the remote
contains work that you do
hint: not have locally. This is usually caused
by another repository pushing
hint: to the same ref. You may want to first in
tegrate the remote changes
hint: (e.g., 'git pull ...') before pushing aga
in.
hint: See the 'Note about fast-forwards' in 'gi
t push --help' for details.
user1:

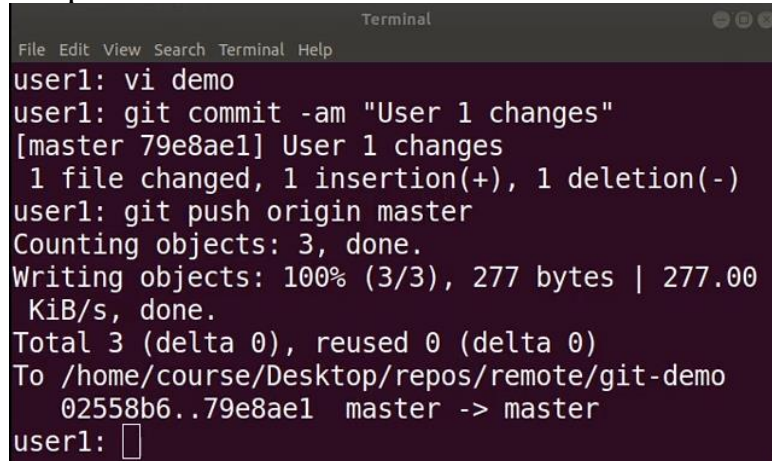
```

Впертий користувач може захотіти все-одно внести свої зміни:  
**git push origin master --force**



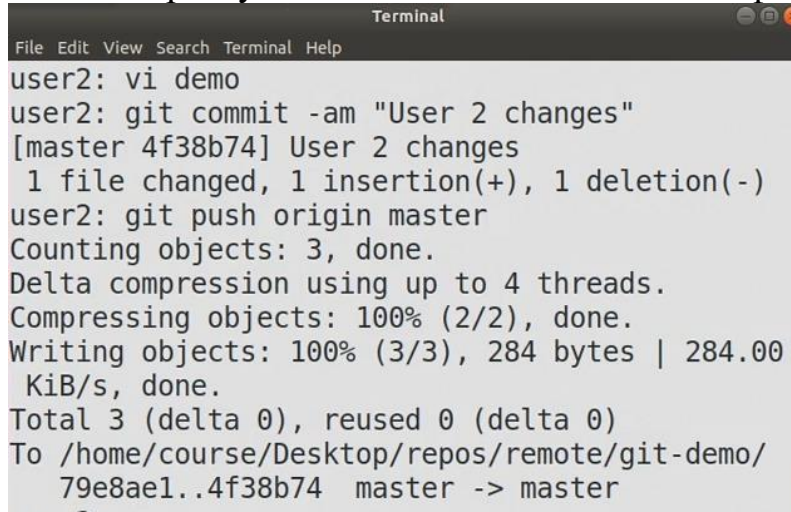
Це призведе до втрати змін Користувача 2. *Зробіть скриншот вмісту репозиторія, щоб показати відсутність відповідного файлу.*

**2.3. Корисна практика: часті коміти та синхронізація.** Нехай Користувач 1 вносить зміни в початковий файл (demo), фіксує їх та відправляє в центральний репозиторій:



```
Terminal
File Edit View Search Terminal Help
user1: vi demo
user1: git commit -am "User 1 changes"
[master 79e8ae1] User 1 changes
1 file changed, 1 insertion(+), 1 deletion(-)
user1: git push origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 277 bytes | 277.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /home/course/Desktop/repos/remote/git-demo
02558b6..79e8ae1 master -> master
user1: █
```

Відразу після цього Користувач 2 вносить свої зміни в цей файл

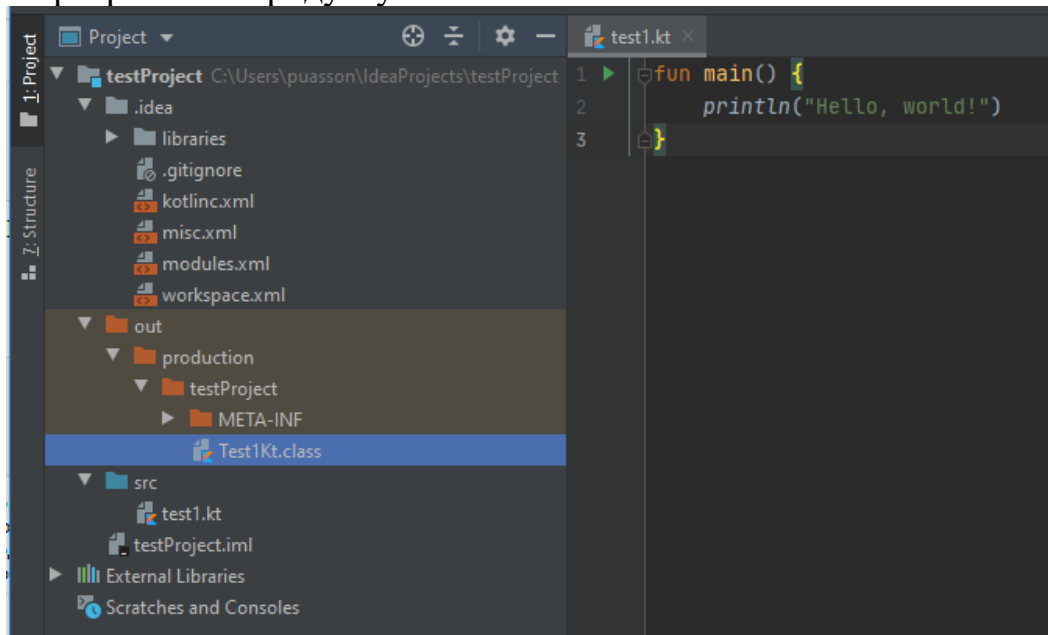


```
Terminal
File Edit View Search Terminal Help
user2: vi demo
user2: git commit -am "User 2 changes"
[master 4f38b74] User 2 changes
1 file changed, 1 insertion(+), 1 deletion(-)
user2: git push origin master
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 284 bytes | 284.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To /home/course/Desktop/repos/remote/git-demo/
79e8ae1..4f38b74 master -> master
```

Часті коміти та синхронізації дозволяють практично не переживати через можливість появи конфліктів злиття. При роботі в команді рекомендується фіксувати та синхронізувати зміни кілька разів на день.

**2.4. Корисна практика: використання *gitignore*.** У реальних проектах існує багато файлів, які потрібні локальному середовищу розробки для роботи, проте не повинні потрапити в центральний репозиторій. Одна з причин цього – різні розробники віддають перевагу різним середовищам розробки. Наприклад, розглянемо Kotlin-проект у середовищі IntelliJ IDEA. Основний код знаходиться в файлі test1.kt. Після компіляції генерується відповідний файл Test1Kt.class, який може виконуватись віртуальною машиною Java. Решта файлів, зокрема в

директорії `.idea`, містить конфігураційні налаштування проекту саме для даного середовища розробки. Якщо інший розробник працює не в IntelliJ IDEA, ці файли йому будуть просто непотрібними. Також вони будуть зайві при розгортанні готового програмного продукту.



Рекомендується підтримувати файл `.gitignore`, який включатиме перелік файлів, які не потраплятимуть з локального репозиторія в центральний при пушингу змін. Зазвичай цей файл розміщується в кореневій папці локального репозиторія. Кожний рядок `.gitignore` буде задавати шаблон, згідно з яким відповідні файли чи директорії не будуть синхронізуватись. Приклад такого файлу можете переглянути [тут](#).

*Додайте до свого локального репозиторію папку й кілька файлів та запишіть потрібні для них шаблони в `.gitignore`. Доповніть звіт скриншотом, який порівнюватиме центральний та локальний репозиторії.*

Для прикладу для новоствореного репозиторія можна виконати такі команди:

```
kb: git init  
Initialized empty Git repository in /home/course/Desktop/repos/git-ignore-demo/.git/  
kb: git add .  
kb: git status  
On branch master  
  
No commits yet  
  
Changes to be committed:  
  (use "git rm --cached <file>..." to unstage)  
  
    new file:   .gitignore  
    new file:   src/Application.java
```



## Завдання 1.6.

*Поекспериментуйте з видаленнями коммітів та відновленням попередніх станів репозиторія. Додайте скриншоти з відповідними командами та логами коммітів*

**2.5. Видалення неопублікованих коммітів.** Якщо ви ще не опублікували комміти у віддаленому репозиторії, зокрема на GitHub, існує можливість видалити їх за допомогою команди `git reset`. Це *ефективне, проте небезпечне* рішення. Ви переписуєте історію та залишаєте «видалені» комміти без посилань на них, «осиротілими». Єдиний спосіб знайти та відновити ці незв'язані комміти – це `git reflog`.

Команда `reset` має 3 різних параметри, два з яких опишемо. Використовуючи опцію `--hard`, все повертається назад до вказаного комміту, зокрема посилання на історію коммітів, проміжний індекс та ваш робочий каталог.

```
$ git reset --hard
```

Це означає, що за допомогою цієї команди ви не тільки повернетеся до попередньої фіксації, а й втратите всі робочі зміни в процесі. Щоб не втратити будь-які робочі зміни, ви можете використовувати команди `stash` і `stash pop`:

```
$ git stash
$ git reset --hard
$ git stash pop
```

Команда `stash` зберігає ваші робочі зміни (без будь-яких коммітів або змін в дереві), а потім `stash pop` повертає їх назад.

Іншим варіантом, який ви можете розглянути, є параметр `--soft`. Ця опція працює так само, як `git reset --hard`, але впливає тільки на історію коммітів, а не на ваш робочий каталог або проміжний індекс.

```
$ git reset --soft
```

**2.6. Видалення опублікованих коммітів.** Нехай ви зафіксували свій код, а потім відправили його у віддалений репозиторій. На цьому етапі *дуже не рекомендується* використовувати команди на зразок `git reset`, оскільки ви переписуєте історію.

Замість цього рекомендується використати команду `git revert`. Вона працює, відмінюючи зміни, які були внесені в указаний комміт, створюючи новий комміт та фактично не видаляючи попередні. Це ідеально для опублікованих змін, оскільки тоді реальна історія репозиторія зберігається. Приклад:

```
$ git revert
```

Припустимо, в репозиторії є текстовий файл із вмістом

```
This is my sample text
```

а ви змінюєте його на

```
This is my awesome sample text
```

Відповідна історія коммітів виглядає приблизно так:

```
$ git log --pretty=oneline
676ec97a9cb2ceb5c77904bbc61ced05b86f52 Added 'awesome' to text
735c5b43bf4b5b7107a9cc3f6614a3890e2889f6 Initial commit
```

Якщо ви вирішили, що більше не потрібно слово «awesome» в даному тексті, але не бажаєте видаляти комміт 676ec, можете використовувати `revert`, щоб скасувати цю зміну:

```
$ git revert 676ec
[master f68e546] Revert "Added 'awesome' to text"
1 file changed, 1 insertion(+), 1 deletion(-)
```

Отримавши запрошення ввести повідомлення про комміт, ви тепер можете бачити в історії коммітів, що фактично існує новий комміт:

```
$ git log --pretty=oneline
f68e546ac2ae240f22b2676b5aec499aab27f1ca Revert "Added 'awesome' to text"
676ec97a9cb2ceb5c77904bbc61ced05b86f52 Added 'awesome' to text
735c5b43bf4b5b7107a9cc3f6614a3890e2889f6 Initial commit
```

У результаті цього перший і третій комміти представляють один і той же стан проекту. Комміт був скасований, і історія не була втрачена.

Зверніть увагу, що є кілька інших способів використовувати цю команду, наприклад, якщо ви хочете повернути назад 2 комміти, ви можете використовувати:

```
git revert HEAD~2
```

Або, якщо ви хочете скасувати багато непостійних коммітів, ви вказуєте їх індивідуально:

```
git revert 676ec 735c5
```

<https://dev-gang.ru/article/git-vernutsja-k-predydusczemu-kommitu-1qctx8l7f6/>

### Групова розробка реферативного повідомлення



#### **Завдання 1.7.**

*Об'єднайтесь у групи по 2-3 особи та застосуйте отримані навички роботи з Git для створення реферату згідно з обраною тематикою.*

№	Тема	ПІБ студентів
1.	Стили кодування та написання якісного коду	
2.	Системи контролю версій	
3.	Програмна платформа Docker	
4.	Scrum-підхід до організації процесу розробки програмного забезпечення	
5.	Методи тестування програмного забезпечення	
6.	Методи оцінювання витрат на розробку програмного забезпечення	
7.	Професії в геймдеві	
8.	Обчислювальна складність алгоритмів	
9.		
10.		

*Реферат повинен включати матеріали мінімум з 3 книжок та 8 джерел загалом. Обсяг реферату повинен бути не меншим за 15 сторінок основного тексту, оформленого за стандартними вимогами: шрифт – Times New Roman, кегль – 14, інтервали – 1.5, відступ для абзацу – 1.25см та ін.*

*Продемонструйте в звіті статистику роботи над рефератом по користувачах, історію коммітів, процес збору джерел (підбір літератури, посилання на сторінки з основним матеріалом та веб-джерела) тощо.*

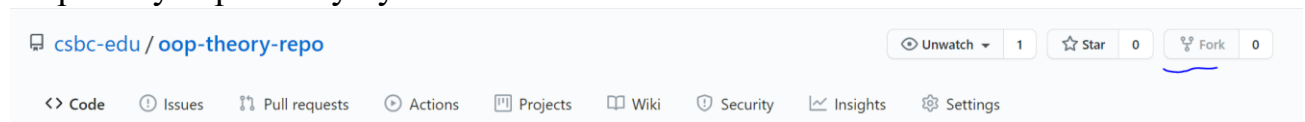
Публікація зазначеної вітки в віддаленому репозиторії разом з усіма необхідними коммітами і внутрішніми об'єктами здійснюється за допомогою команди `git push`. Ця команда створює локальну вітку в репозиторії призначення. Щоб запобігти перезапису коммітів, Git не дозволить опублікувати дані, якщо в репозиторії призначення не можна виконати прискорене злиття.

<https://www.atlassian.com/ru/git/tutorials/syncing/git-push>

**Внесення свого вкладу в проекти.** Якщо ви хочете вносити свій вклад в уже існуючі проекти, в яких у нас немає прав на внесення змін шляхом відправки (push) змін, ви можете створити своє власне відгалуження ("fork") проекту. Це означає, що GitHub створить вашу власну копію проекту, дана копія буде знаходитися в вашому просторі імен і ви зможете легко робити зміни шляхом відправки (push) змін. Таким чином, проекти не турбуються, щоб користувачі, які хотіли б виступати в ролі співавторів, мали право на внесення змін шляхом їх відправки (push). Люди просто можуть створювати свої власні розгалуження (fork), вносити туди зміни, а потім відправляти свої внесені зміни в оригінальний репозиторій проекту шляхом створення запиту на прийняття змін (Pull Request).

Запит на прийняття змін (Pull Request) відкриє нову вітку з обговоренням відправленого коду, і автор оригінального проекту, а також інші його учасники, можуть брати участь в обговоренні запропонованих змін до тих пір, поки автор проекту не буде ними задоволений, після чого автор проекту може додати запропоновані зміни в проект.

Для того, щоб створити відгалуження проекту (fork), зайдіть на сторінку проекту і натисніть кнопку "Створити відгалуження" ( "Fork"), яка розташована в правому верхньому куті.



Через кілька секунд ви будете перенаправлені на власну нову проектну сторінку, яка містить вашу копію, в якій у вас є права на запис.

GitHub розроблений з прицілом на певний робочий процес з використанням запитів на злиття. Цей робочий процес добре підходить всім: і маленьким, згуртованим навколо одного репозиторія, командам; і великим розподіленим компаніям, і групам незнайомих, які співпрацюють над проектом з сотнею копій.



<https://git-scm.com/book/ru/v2/GitHub-%D0%92%D0%BD%D0%B5%D1%81%D0%B5%D0%BD%D0%B8%D0%B5-%D1%81%D0%BE%D0%B1%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D0%B3%D0%BE-%D0%B2%D0%BA%D0%BB%D0%B0%D0%B4%D0%B0-%D0%B2-%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D1%8B>

Gitflow Workflow – це модель робочого процесу Git, яка була вперше опублікована і популяризована Вінсентом Дріссеном з компанії nvie. Gitflow Workflow передбачає вибудовування строгої моделі розгалуження з урахуванням випуску проекту. Така модель забезпечує надійну основу для управління великими проектами.

Gitflow ідеально підходить для проектів, в яких цикл релізу протікає по графіку. У цьому робочому процесі використовуються поняття і команди, які були запропоновані в рамках процесу [Feature Branch Workflow](#). Однак Gitflow привносить нові специфічні ролі для різних віток і визначає характер і частоту взаємодії між ними. Крім віток feature в рамках цього робочого процесу використовуються окремі вітки для підготовки, підтримки та реєстрації випусків. При цьому ви як і раніше можете користуватися перевагами процесу Feature Branch Workflow, такими як запити pull, ізольовані експерименти і більш ефективне командну взаємодію.

<https://www.atlassian.com/ru/git/tutorials/comparing-workflows/gitflow-workflow>

**2.5. Корисна практика: Line Endings.** Операційні системи можуть мати різні символи для закінчення рядка та переходу до наступного. Якщо на Unix-системах використовується тільки [символ нового рядка](#) (Line feed, LF), то машини на Windows підтримують також символ повернення каретки (carriage return, CR+LF = CRLF).

Git пропонує налаштувати конфігурацію в файлі core.autocrlf, спрямовану на вирішення даної проблеми. Наприклад, якщо в текстовому редакторі vim на операційній системі Ubuntu відкрити файл з первинним кодом, який редагувався на Windows-машині, отримаємо багато символів повернення каретки (у Linux-системах позначаються як ^M) в тексті файлу:

```
File Edit View Search Terminal Help
public class Sample {

    ^M
    ^M
    public static void main(String[] args) {
        ^M
        ^M
        String result = "Result:";
        ^M
        result += Sample.concat("A","B");
        ^M
        System.out.println(result);
        ^M
        String result2 = result + " Completed";

        ^M
    }
}
```

Скомпілювати такий код під Linux буде неможливо, доведеться видаляти всі такі символи. Для вирішення цієї проблеми пропонується застосувати подібні команди:

```
kb: git init
Initialized empty Git repository in /home/course/Desktop/repos/line-endings/.git/
kb: git config core.autocrlf input
kb: git add .
warning: CRLF will be replaced by LF in Sample.java.
The file will have its original line endings in your working directory.
kb: git commit -m "Changes"
[master (root-commit) 312ba10] Changes
1 file changed, 28 insertions(+)
create mode 100644 Sample.java
```

Дані зміни внесені в новий репозиторій, тому створимо нову вітку, якій можемо видалити старий проблемний файл та

```
kb: git checkout -b other
Switched to a new branch 'other'
kb: rm Sample.java
kb: git add .
kb: git commit -m "changes"
[other 38b0cba] changes
1 file changed, 28 deletions(-)
delete mode 100644 Sample.java
```

```
File Edit View Search Terminal Help
kb: git checkout master
Switched to branch 'master'
```

Налаштування значень autocrlf



autocrlf	Result	When to Use
input	Checkout LF to LF Checkout LF to LF	Cross-platform + Unix
TRUE	Commit CRLF to LF Checkout LF to CRLF	Cross-platform + Windows
FALSE	No conversions	Single platform