VSC

💡 1. Что такое система контроля версий (СКВ)?

Система контроля версий (Version Control System, VCS) — это инструмент, который **отслеживает изменения в файлах** (чаще всего — в исходном коде программы) и позволяет:

- сохранять историю всех изменений;
- возвращаться к предыдущим версиям;
- работать над проектом нескольким людям одновременно;
- видеть, кто и когда внёс изменения.

🧩 Примеры систем контроля версий:

- **Git** (наиболее популярная),
- **Subversion (SVN)**,
- **Mercurial**.

🎯 2. Основная цель использования Git

Git — это распределённая система контроля версий.
Её **главная цель** — обеспечить **удобную, надёжную и безопасную работу с историей изменений** в коде, особенно при **совместной разработке**.

Ключевые преимущества Git:

- Полная история проекта хранится у каждого участника (можно работать офлайн).
- Простое слияние изменений от разных разработчиков.
- Возможность создания **веток (branches)** для экспериментов и новых функций.
- Контроль версий, кто и что изменил.

⚙️ 3. Как создать новый репозиторий в Git

Есть два основных способа:

✅ Способ 1. Создать репозиторий в существующей папке проекта

```
cd путь/к/папке
git init
```

➡️ Git создаст скрытую папку `.git` — туда будет сохраняться история изменений.

✅ Способ 2. Клонировать уже существующий репозиторий

```
git clone https://github.com/username/repository.git
```

➡️ Скопирует репозиторий с сервера (например, GitHub) на ваш компьютер.

🔄 После создания репозитория обычно делают:

```
git add .            # Добавить файлы в индекс
git commit -m "Первый коммит"   # Сохранить изменения
git branch -M main  # Переименовать ветку master в main (опционально)
git remote add origin https://github.com/username/repo.git  # Привязать
удалённый репозиторий
git push -u origin main   # Отправить изменения на сервер
```

🧩 1. Состояния файлов в Git

В Git каждый файл в проекте может находиться **в одном из четырёх состояний**:

Состояние	Что значит	Команда для перехода
Untracked	Файл **не отслеживается** Git (новый, не добавлен в индекс).	`git add <файл>`
Tracked / Unmodified	Файл **отслеживается** и **не изменён** с момента последнего коммита.	—
Modified	Файл **изменён**, но **изменения не добавлены** в индекс.	`git add <файл>`
Staged	Изменения **добавлены в индекс** (готовы к коммиту).	`git commit -m "Сообщение"`

📊 Визуально это можно представить так:

```
Untracked → (git add) → Staged → (git commit) → Tracked/Unmodified
              ↑                                    ↓
           (изменили файл) ←—— Modified ——→
```

🌿 2. Как переключиться на другую ветку и создать новую

🔄 Переключение на существующую ветку:

```
git checkout имя_ветки
```

Например:

```
git checkout main
```

🌱 Создание новой ветки:

```
git branch имя_новой_ветки
```

Затем перейти в неё:

```
git checkout имя_новой_ветки
```

⚡ Или одной командой (современный вариант):

```
git switch -c имя_новой_ветки
```

Пример:

```
git switch -c feature/login-page
```

(создаст и сразу переключит на ветку `feature/login-page`)

🔗 3. Как объединить две ветки с помощью merge

Предположим:

- ты находишься в ветке `main`,
- хочешь объединить в неё изменения из ветки `feature`.

Команда:

```
git merge feature
```

Git попытается объединить изменения:

- Если **нет конфликтов** → создаётся новый коммит слияния.
- Если **есть конфликты**, Git сообщит об этом, и нужно вручную решить их в файлах, затем:
- `git add <файлы_с_конфликтами>`
- `git commit`

🔁 Пример процесса:

```
git switch main           # Переключаемся на главную ветку
git merge feature          # Сливаем ветку feature в main
```

Хочешь, я покажу наглядную схему, как ветки и коммиты выглядят до и после `merge` (в виде ASCII-графа или картинки)?

⚔️ 1. Какие бывают конфликты в Git

Конфликт в Git возникает, когда Git **не может автоматически объединить изменения из двух веток**.

Это происходит, если **в одном и том же месте файла** были сделаны **разные изменения**.

🔸 Примеры конфликтов:

1. **Один и тот же участок кода изменён по-разному**:

```
<<<<<<< HEAD
color = "blue"
=======
color = "red"
>>>>>>> feature
```

   Здесь ветка `main` изменила цвет на `blue`, а `feature` — на `red`.

7. **Один удалил файл, другой изменил его**
   Git не знает — удалить файл или сохранить изменения.
8. **Изменены одинаковые строки в одном файле** двумя участниками.

🧭 2. Процесс решения конфликтов при слиянии веток

Пошагово 👇

① Запускаем слияние

```
git merge feature
```

Если есть конфликты — Git покажет сообщение вроде:

```
Auto-merging main.py
CONFLICT (content): Merge conflict in main.py
```

---

② Открываем файл с конфликтом

Git вставит специальные разделители:

```
<<<<<<< HEAD
print("Hello from main")
=======
print("Hello from feature")
>>>>>>> feature
```

---

③ Ручное решение

Редактируем вручную, чтобы выбрать правильный вариант:

```
print("Hello from both versions!")
```

---

④ Отмечаем, что конфликт решён

```
git add main.py
```

---

⑤ Завершаем слияние

```
git commit
```

💡 Git создаст **merge commit** — специальный коммит, объединяющий историю обеих веток.

🧱 3. Как работает система хранения Git (снимки, а не дельты)

Git **не хранит изменения построчно**, как старые системы (например, SVN).
Он работает по принципу **снимков (snapshots)**.

📸 Представь:
Каждый коммит — **снимок (snapshot)** всего проекта **в текущий момент времени**.

- Если файл не изменился — Git **не копирует его заново**, а **создаёт ссылку на прежнюю версию**.
- Это делает Git **очень быстрым** и **надёжным**, ведь можно мгновенно вернуться к любой версии.

📦 Структура хранения:

- Каждый объект в Git (файл, коммит, ветка) хранится под **хешем (SHA-1)**.
- Это даёт защиту от подделки и гарантирует целостность данных.

🧠 Итог:
Git — не “записывает, что изменилось”, а “сохраняет, как выглядит всё сейчас”.

🔙 4. Как отменить коммит, уже отправленный в удалённый репозиторий, не удаляя историю

Если коммит уже ушёл на GitHub или GitLab, **удалять его нельзя** — это может повредить историю других разработчиков.
Но можно **отменить его действие** новым коммитом.

✅ Используй:

```
git revert <хеш_коммита>
```

- Эта команда **создаёт новый коммит**, который **отменяет изменения** из указанного.
- История сохраняется, ничего не ломается.

Пример:

```
git log --oneline      # Посмотреть хеши коммитов
git revert a1b2c3d4    # Отменить конкретный коммит
git push origin main   # Отправить изменения
```

💡 В отличие от `git reset`, команда `git revert` безопасна для командной работы.

📥 1. Команда для клонирования репозитория

Клонирование (копирование) удалённого репозитория на компьютер:

```
git clone <url>
```

- Пример:

```
git clone https://github.com/user/project.git
```

➡️ Эта команда:

- создаёт папку с проектом;
- скачивает все файлы;
- загружает историю коммитов;
- настраивает удалённый репозиторий под именем `origin`.

🕓 2. Команда для просмотра истории коммитов

Чтобы увидеть историю изменений:

```
git log
```

- Показывает:

- автора коммита,
- дату,
- сообщение,
- хеш коммита.

📋 Пример вывода:

```
commit a1b2c3d4
Author: Alex <alex@example.com>
Date:   Fri Nov 10 10:15 2025
```

```
    Добавил страницу авторизации
```

💡 Удобные варианты:

```
git log --oneline        # Короткий формат (1 строка на коммит)
git log --graph --oneline --decorate --all  # Ветвления в виде дерева
```

🌿 3. Команда для создания новой ветки

Создать новую ветку:

```
git branch <имя_ветки>
```

🔹 Пример:

```
git branch feature/login-page
```

➡️ Эта команда только **создаёт ветку**, но **не переключает** на неё.

Чтобы сразу создать **и перейти** в новую ветку:

```
git checkout -b <имя_ветки>
```

или современный вариант:

```
git switch -c <имя_ветки>
```

🔗 1. Команда для слияния изменений из одной ветки в другую

Чтобы **объединить изменения** из другой ветки, используется команда:

```
git merge <имя_ветки>
```

📌 Пример:

```
git switch main         # Переходим на ветку, куда хотим слить изменения
git merge feature        # Сливаем ветку feature в main
```

🧩 Если ветки не конфликтуют — создаётся *merge commit*.
Если есть конфликты — Git попросит их вручную разрешить (см. предыдущий ответ).

🔙 2. Как отменить изменения в файле и вернуть его к последнему коммиту

Если ты изменил файл, но **ещё не добавил его в индекс** (`git add`), можно вернуть его к последней сохранённой версии:

```
git restore <имя_файла>
```

📘 Пример:

```
git restore index.html
```

➡️ Это отменит все несохранённые изменения и вернёт файл к состоянию последнего коммита.

💡 Альтернативный старый вариант (всё ещё часто используется):

```
git checkout -- <имя_файла>
```

⚠️ Осторожно: команда безвозвратно удаляет несохранённые правки.

📝 3. Команда для изменения последнего коммита (только сообщения)

Если ты уже сделал коммит, но понял, что **нужно поменять сообщение**, без изменения содержимого:

```
git commit --amend -m "Новое сообщение коммита"
```

📘 Пример:

```
git commit --amend -m "Исправлено сообщение последнего коммита"
```

➡️ Эта команда:

- не меняет файлы,
- просто обновляет сообщение у последнего коммита.

⚠️ Если коммит **уже отправлен на удалённый репозиторий**, **не рекомендуется** использовать `--amend`, чтобы не ломать историю другим участникам.

🧩 1. Команда для перебазирования текущей ветки на указанную

Перебазирование — это способ "переписать" историю коммитов, чтобы ветка выглядела так, будто она создана поверх другой.

Команда:

```
git rebase <имя_ветки>
```

📘 Пример:

```
git switch feature
git rebase main
```

➡️ Что происходит:

- Git “перемещает” коммиты из ветки `feature` так, чтобы они шли **после** последних коммитов ветки `main`.
- История становится **линейной** и более аккуратной (без merge-коммитов).

⚠️ Осторожно: не рекомендуется использовать `rebase` для веток, которые **уже были отправлены** на сервер — это может запутать историю.

💾 2. Как временно сохранить изменения без коммита

Когда нужно быстро переключиться на другую ветку, но изменения ещё не готовы для коммита — используй **stash**.

Команда:

```
git stash
```

➡️ Она:

- сохраняет текущие изменения во временное хранилище;
- очищает рабочую директорию (возвращает её к последнему коммиту).

📦 Основные команды:

Команда	Что делает
`git stash`	Сохраняет текущие изменения
`git stash list`	Показывает список сохранённых наборов
`git stash apply`	Применяет последний stash, но оставляет его в списке
`git stash pop`	Применяет последний stash **и удаляет его** из списка

📘 Пример:

```
git stash
git switch main
git pull
git switch feature
git stash pop
```

🌿 3. Как просмотреть список всех веток (локальных и удалённых)

Чтобы увидеть **все ветки**, включая **удалённые**, и последние коммиты в них:

```
git branch -a -v
```

📋 Расшифровка:

- `-a` — показать **все ветки** (локальные и удалённые),
- `-v` — показать **последний коммит** в каждой ветке.

📘 Пример вывода:

```
* main                 5f2b3a7 Добавлен README
  feature/login        a8c9d02 Добавлен шаблон авторизации
  remotes/origin/main  5f2b3a7 Добавлен README
  remotes/origin/dev   3c2a1f9 Исправлены стили
```

🔹 Только локальные ветки:

```
git branch
```

🔹 Только удалённые ветки:

```
git branch -r
```
