Programming 2: **Version Control with Git** (Lecture 3)

Comp 111

Forman Christian University

Learning Objectives

By the end of this lecture, you will be able to:

- Explain **why** version control matters

Learning Objectives

By the end of this lecture, you will be able to:

- Explain **why** version control matters
- Initialize a Git repository and make **commits**

Learning Objectives

By the end of this lecture, you will be able to:

- Explain **why** version control matters
- Initialize a Git repository and make **commits**
- Understand **branches** and when to use them

Learning Objectives

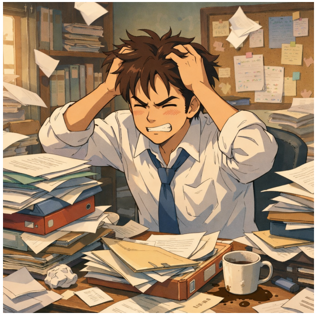
By the end of this lecture, you will be able to:

- Explain **why** version control matters
- Initialize a Git repository and make **commits**
- Understand **branches** and when to use them
- Describe how teams **collaborate** using Git

The Problem

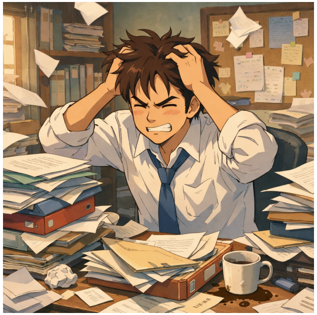
Sound Familiar?

```
essay.docx
essay_final.docx
essay_final_v2.docx
essay_FINAL_REAL.docx
essay_FINAL_REAL_thisone.docx
essay_submitted_oops.docx
```



Sound Familiar?

```
essay.docx
essay_final.docx
essay_final_v2.docx
essay_FINAL_REAL.docx
essay_FINAL_REAL_thisone.docx
essay_submitted_oops.docx
```



We've all been there!

It Gets Worse...

- What if **two people** edit the same file?

It Gets Worse...

- What if **two people** edit the same file?
- How do you **undo** a mistake from last week?

It Gets Worse...

- What if **two people** edit the same file?
- How do you **undo** a mistake from last week?
- How do companies manage code from **thousands** of developers?

It Gets Worse...

- What if **two people** edit the same file?
- How do you **undo** a mistake from last week?
- How do companies manage code from **thousands** of developers?

Solution: Version Control with Git

Real-World Connection

Every app, website, and game you use is built with Git:

- Google
- Microsoft
- Netflix
- Meta
- Apple
- Amazon
- Every startup
- Open source projects

Real-World Connection

Every app, website, and game you use is built with Git:

- Google
- Microsoft
- Netflix
- Meta

- Apple
- Amazon
- Every startup
- Open source projects

Today you learn a professional skill!

What is Git?

Git = Video Game Save System

- **Save** your progress at any point

Git = Video Game Save System

- **Save** your progress at any point
- **Go back** to any previous save

Git = Video Game Save System

- **Save** your progress at any point
- **Go back** to any previous save
- **Try risky strategies** without losing main save

Git = Video Game Save System

- **Save** your progress at any point
- **Go back** to any previous save
- **Try risky strategies** without losing main save
- **Combine progress** with friends

Key Vocabulary

Repository	A folder that Git is tracking

Key Vocabulary

Repository	A folder that Git is tracking
Commit	A snapshot / save point of your code

Key Vocabulary

Repository	A folder that Git is tracking
Commit	A snapshot / save point of your code
Branch	A parallel version to experiment safely

Key Vocabulary

Repository	A folder that Git is tracking
Commit	A snapshot / save point of your code
Branch	A parallel version to experiment safely
Merge	Combining two branches together

Your First Repository

Creating a Repository *(local)*

```
# 1. Create a project folder
mkdir my-first-repo
cd my-first-repo

# 2. Initialize Git (start tracking)
git init

# 3. Check status (your new best friend!)
git status
```

Creating a Repository *(local)*

```bash
# 1. Create a project folder
mkdir my-first-repo
cd my-first-repo

# 2. Initialize Git (start tracking)
git init

# 3. Check status (your new best friend!)
git status
```

```
Initialized empty Git repository in .../my-first-repo/.git/
```

Creating Your First File

```
# Create a file
echo "Hello, Git!" > hello.txt

# Check status - Git notices!
git status
```

Creating Your First File

```bash
# Create a file
echo "Hello, Git!" > hello.txt

# Check status - Git notices!
git status
```

```
Untracked files:
    hello.txt
```

Creating Your First File

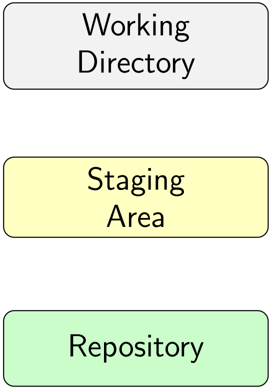
```
# Create a file
echo "Hello, Git!" > hello.txt

# Check status - Git notices!
git status
```

```
Untracked files:
    hello.txt
```

Git sees the file, but isn't saving it yet!

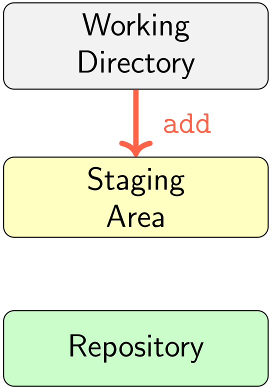
The Two-Step Save

```
#Step 1: Stage(choose what to save)
git add hello.txt

#Step 2: Commit (actually save it)
git commit -m "Add hello.txt"
```



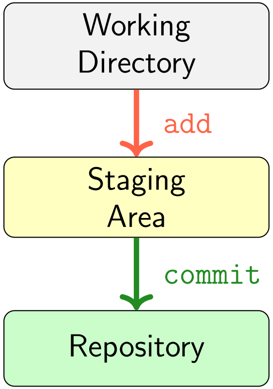
The Two-Step Save

```
#Step 1: Stage(choose what to save)
git add hello.txt

#Step 2: Commit (actually save it)
git commit -m "Add hello.txt"
```



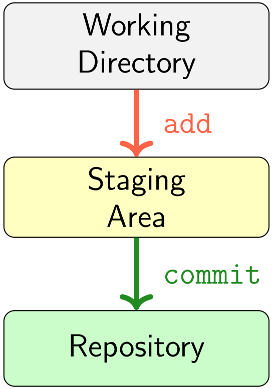
The Two-Step Save

```
#Step 1: Stage(choose what to save)
git add hello.txt

#Step 2: Commit (actually save it)
git commit -m "Add hello.txt"
```



The Two-Step Save

```
#Step 1: Stage(choose what to save)
git add hello.txt

#Step 2: Commit (actually save it)
git commit -m "Add hello.txt"
```



Analogy: Staging = shopping cart, Commit = checkout

Making More Commits

```bash
# Edit the file
echo "This is my project." >> hello.txt

# See what changed
git diff

# Stage and commit (in one step!)
git commit -am "Add project description"
```

Making More Commits

```
# Edit the file
echo "This is my project." >> hello.txt

# See what changed
git diff

# Stage and commit (in one step!)
git commit -am "Add project description"
```

Each commit = a save point you can return to!

Reading `git diff`

```
--- a/hello.txt
+++ b/hello.txt

 Hello World!
+This is my project.
```

Reading `git diff`

```
--- a/hello.txt
+++ b/hello.txt

 Hello World!
+This is my project.
```

- + **Green** = line **added**
- – **Red** = line **removed**
- No prefix = unchanged (context)

Viewing Your History

```
# See all commits
git log

# Prettier version
git log --oneline
```

Viewing Your History

```
# See all commits
git log

# Prettier version
git log --oneline
```

```
a1b2c3d Add project description
e4f5g6h Add hello.txt
```

Viewing Your History

```
# See all commits
git log

# Prettier version
git log --oneline
```

```
a1b2c3d Add project description
e4f5g6h Add hello.txt
```

Pro tip: Write good commit messages!

Bad: "Fixed stuff"   Good: "Fix login button on mobile"

Going Back to a Checkpoint

Remember: Git is like a video game save system!

```
# View your save points
git log --oneline
```

```
a1b2c3d Add project description
e4f5g6h Add hello.txt
```

Going Back to a Checkpoint

Remember: Git is like a video game save system!

```
# View your save points
git log --oneline
```

```
a1b2c3d Add project description
e4f5g6h Add hello.txt
```

Two ways to go back:

- `git restore` — Undo changes to files
- `git revert` — Create new commit that undoes old one

Method 1: `git restore`

Restore a file to how it was in a previous commit:

```
# Oops! I broke hello.txt
echo "BROKEN" > hello.txt

# Restore it to the last commit
git restore hello.txt

# Or restore from a specific commit
git restore --source=e4f5g6h hello.txt
```

Method 1: `git restore`

Restore a file to how it was in a previous commit:

```
# Oops! I broke hello.txt
echo "BROKEN" > hello.txt

# Restore it to the last commit
git restore hello.txt

# Or restore from a specific commit
git restore --source=e4f5g6h hello.txt
```

✓ **Use case:** Undo local changes before committing

Method 1: `git restore`

Restore a file to how it was in a previous commit:

```
# Oops! I broke hello.txt
echo "BROKEN" > hello.txt

# Restore it to the last commit
git restore hello.txt

# Or restore from a specific commit
git restore --source=e4f5g6h hello.txt
```

✓ **Use case:** Undo local changes before committing

✗ **Caution:** Doesn't create a new commit (history unchanged)

Method 2: `git revert`

Create a NEW commit that undoes an OLD commit:

```
# View commits
git log --oneline

# a1b2c3d Add project description
# e4f5g6h Add hello.txt

# Undo "Add project description"
git revert a1b2c3d
```

Method 2: `git revert`

Create a NEW commit that undoes an OLD commit:

```
# View commits
git log --oneline

# a1b2c3d Add project description
# e4f5g6h Add hello.txt

# Undo "Add project description"
git revert a1b2c3d
```

✓ **Use case:** Undo a commit that was already pushed/shared

Method 2: `git revert`

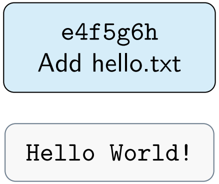
Create a NEW commit that undoes an OLD commit:

```
# View commits
git log --oneline

# a1b2c3d Add project description
# e4f5g6h Add hello.txt

# Undo "Add project description"
git revert a1b2c3d
```

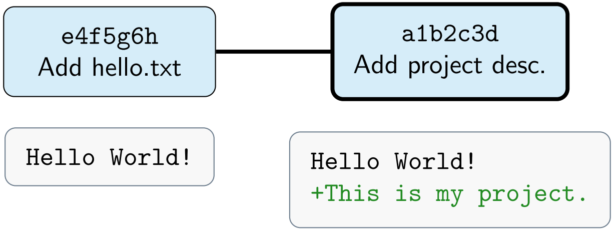
✓ **Use case:** Undo a commit that was already pushed/shared

Safe: Creates new commit, keeps full history

How `git revert` Works



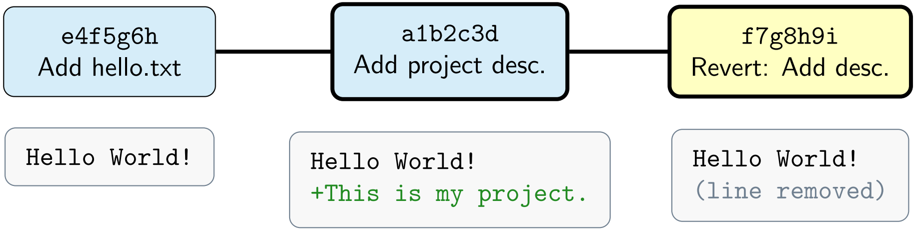
How `git revert` Works



```
Hello World!
```

```
Hello World!
+This is my project.
```

How `git revert` Works



All 3 commits stay in history — nothing is erased!

Restore vs Revert

`git restore`

Undo local changes

(before commit)

No new commit

History unchanged

`git revert`

Undo old commit

(after commit)

Creates new commit

History preserved

Restore vs Revert

`git restore`

Undo local changes

(before commit)

No new commit

History unchanged

`git revert`

Undo old commit

(after commit)

Creates new commit

History preserved

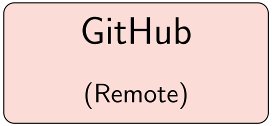
Rule: Use `revert` for shared/pushed commits!

Working with GitHub

The GitHub-First Workflow

Most teams start projects on GitHub:

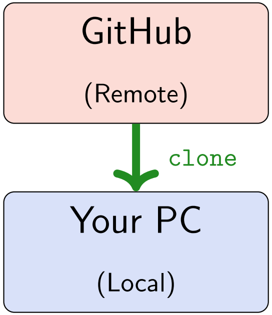
1. Create repo on GitHub *(remote)*



The GitHub-First Workflow

Most teams start projects on GitHub:

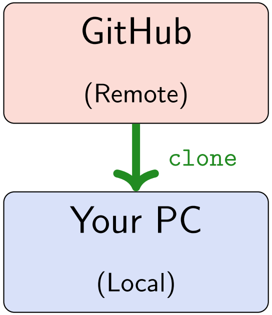
1. Create repo on GitHub *(remote)*
2. Clone to your computer *(local)*



The GitHub-First Workflow

Most teams start projects on GitHub:

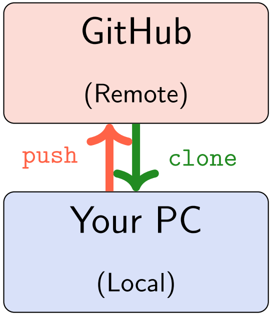
1. Create repo on GitHub *(remote)*
2. Clone to your computer *(local)*
3. Make changes & commit



The GitHub-First Workflow

Most teams start projects on GitHub:

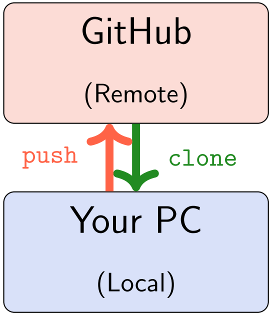
1. Create repo on GitHub *(remote)*
2. Clone to your computer *(local)*
3. Make changes & commit
4. Push back to GitHub



The GitHub-First Workflow

Most teams start projects on GitHub:

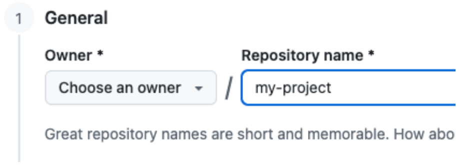
1. Create repo on GitHub *(remote)*
2. Clone to your computer *(local)*
3. Make changes & commit
4. Push back to GitHub



This is how real teams work!

Step 1: Create Repo on GitHub

1. Go to `github.com`
2. Click "**New repository**"
3. Name it `my-project`
4. Add a README (optional)
5. Click "**Create**"



1 General

Owner *

Choose an owner

Repository name *

my-project

Great repository names are short and memorable. How abo

Step 2: Clone to Your Computer

```
# Copy the repo URL from GitHub
# It looks like:
# https://github.com/username/my-project.git

# Clone it to your computer
git clone https://github.com/username/my-project.git

# Enter the folder
cd my-project
```

Step 2: Clone to Your Computer

```
# Copy the repo URL from GitHub
# It looks like:
# https://github.com/username/my-project.git

# Clone it to your computer
git clone https://github.com/username/my-project.git

# Enter the folder
cd my-project
```

```
Cloning into 'my-project'...
done.
```

Step 2: Clone to Your Computer

```
# Copy the repo URL from GitHub
# It looks like:
# https://github.com/username/my-project.git

# Clone it to your computer
git clone https://github.com/username/my-project.git

# Enter the folder
cd my-project
```

```
Cloning into 'my-project'...
done.
```

Now you have a complete copy on your computer!

Step 3: Make Changes & Commit

```
# Create a new file
echo "My awesome project" > project.txt

# The familiar workflow:
git add project.txt
git commit -m "Add project description"

# Check your commits
git log --oneline
```

Step 3: Make Changes & Commit

```
# Create a new file
echo "My awesome project" > project.txt

# The familiar workflow:
git add project.txt
git commit -m "Add project description"

# Check your commits
git log --oneline
```

Same commands as before!
But now the repo came from GitHub.

Step 4: Push to GitHub

```
# Send your commits to GitHub
git push origin main

# Or simply:
git push
```

Step 4: Push to GitHub

```
# Send your commits to GitHub
git push origin main

# Or simply:
git push
```

```
Enumerating objects:  3, done.
Writing objects:  100% (3/3), 256 bytes | 256.00 KiB/s, done.
To https://github.com/username/my-project.git
```

Step 4: Push to GitHub

```
# Send your commits to GitHub
git push origin main

# Or simply:
git push
```

```
Enumerating objects: 3, done.
Writing objects: 100% (3/3), 256 bytes | 256.00 KiB/s, done.
To https://github.com/username/my-project.git
```

Your changes are now on GitHub for the world to see!

Understanding `origin`

```
# See where your repo came from
git remote -v
```

Understanding `origin`

```
# See where your repo came from
git remote -v
```

```
origin https://github.com/username/my-project.git (fetch)
origin https://github.com/username/my-project.git (push)
```

Understanding origin

```
# See where your repo came from
git remote -v
```

```
origin https://github.com/username/my-project.git (fetch)
origin https://github.com/username/my-project.git (push)
```

origin = the GitHub server

Understanding `origin`

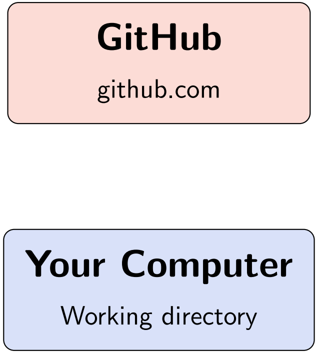
```
# See where your repo came from
git remote -v
```

```
origin https://github.com/username/my-project.git (fetch)
origin https://github.com/username/my-project.git (push)
```

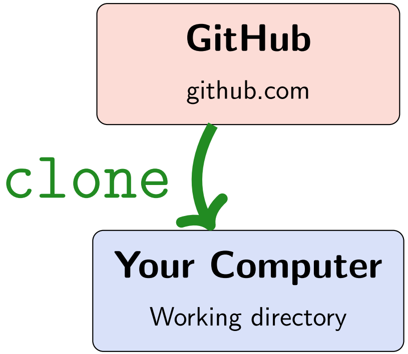
origin = the GitHub server

When you clone, Git automatically sets up `origin`

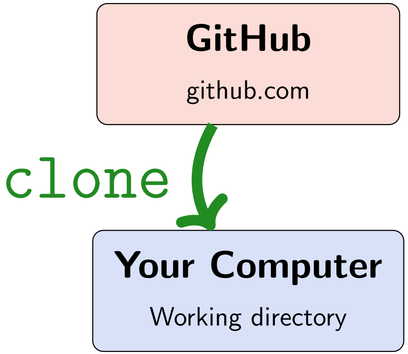
The Complete Cycle



The Complete Cycle



The Complete Cycle

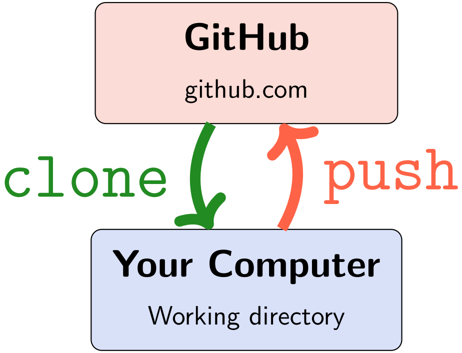


Work locally:

`add`

`commit`

The Complete Cycle

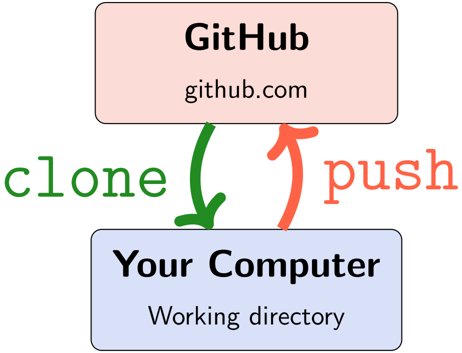


Work locally:

`add`

`commit`

The Complete Cycle



Work locally:

`add`

`commit`

clone once → work locally → **push** often

Why This Workflow?

- Your code is **backed up** on GitHub

Why This Workflow?

- Your code is **backed up** on GitHub
- Teammates can see and contribute

Why This Workflow?

- Your code is **backed up** on GitHub
- Teammates can see and contribute
- Work from multiple computers

Why This Workflow?

- Your code is **backed up** on GitHub
- Teammates can see and contribute
- Work from multiple computers
- Portfolio of your work for job interviews!

Why This Workflow?

- Your code is **backed up** on GitHub
- Teammates can see and contribute
- Work from multiple computers
- Portfolio of your work for job interviews!

Professional developers do this hundreds of times per day

Branches

Branches = The Multiverse

Imagine you could:

- Create a **parallel universe**

Branches = The Multiverse

Imagine you could:

- Create a **parallel universe**
- Try something **risky**

Branches = The Multiverse

Imagine you could:

- Create a **parallel universe**
- Try something **risky**
- If it works: **merge** it back

Branches = The Multiverse

Imagine you could:

- Create a **parallel universe**
- Try something **risky**
- If it works: **merge** it back
- If it fails: **delete** that universe

Branches = The Multiverse

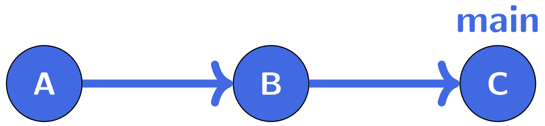
Imagine you could:

- Create a **parallel universe**
- Try something **risky**
- If it works: **merge** it back
- If it fails: **delete** that universe

No harm to your main timeline!

Branch Visualization

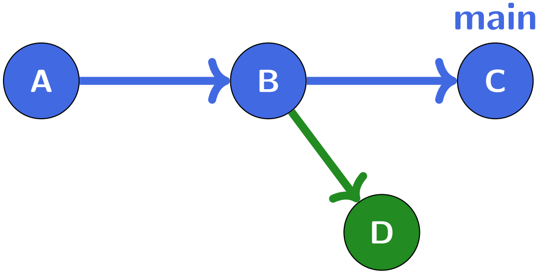
Each circle = a commit (save point)



- A → B → C: Commits on **main** branch

Branch Visualization

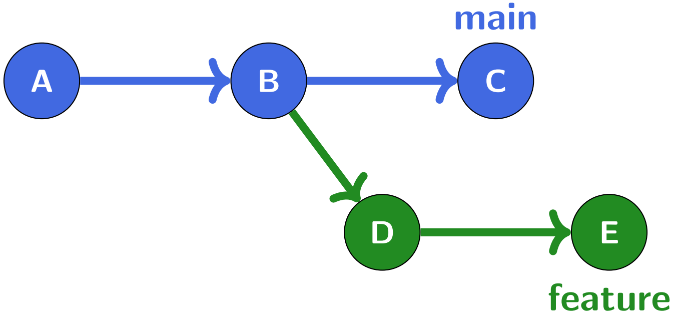
Each circle = a commit (save point)



- A → B → C: Commits on **main** branch

Branch Visualization

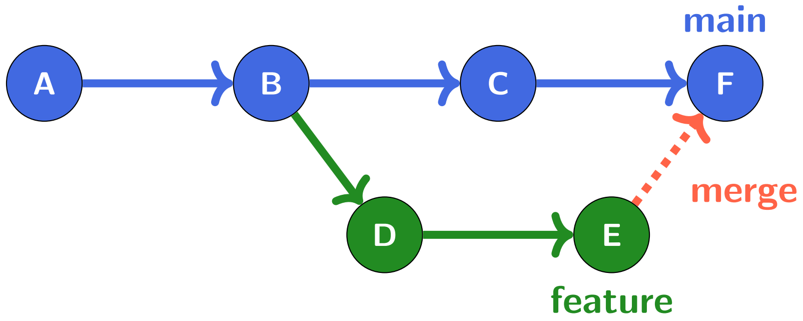
Each circle = a commit (save point)



- A → B → C: Commits on **main** branch
- D → E: Commits on **feature** branch (created from B)

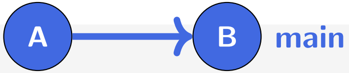
Branch Visualization

Each circle = a commit (save point)



- A → B → C: Commits on **main** branch
- D → E: Commits on **feature** branch (created from B)
- F: Merge commit — combines both branches

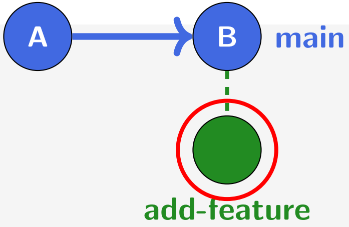
Creating a Branch



```
# See current branch
git branch

# Create and switch to new branch
git checkout -b add-feature

# Or the newer syntax:
git switch -c add-feature
```

Creating a Branch

```
# See current branch
git branch

# Create and switch to new branch
git checkout -b add-feature

# Or the newer syntax:
git switch -c add-feature
```

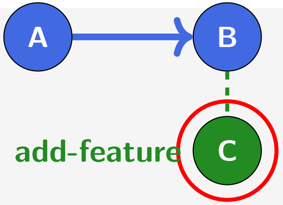


Both branches point to the same commit initially

```
Switched to a new branch 'add-feature'
```

Working on a Branch

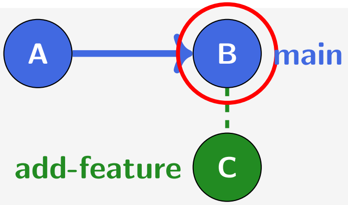
```
# Make changes on the branch
echo "Cool new feature!" >> hello.txt
git add hello.txt
git commit -m "Add cool feature"

# Switch back to main
git checkout main

# hello.txt doesn't have our
#  feature yet!
cat hello.txt
```



Commit C: "Add cool feature"
New commit on feature branch

Working on a Branch

```
# Make changes on the branch
echo "Cool new feature!" >> hello.txt
git add hello.txt
git commit -m "Add cool feature"

# Switch back to main
git checkout main

# hello.txt doesn't have our
#  feature yet!
cat hello.txt
```



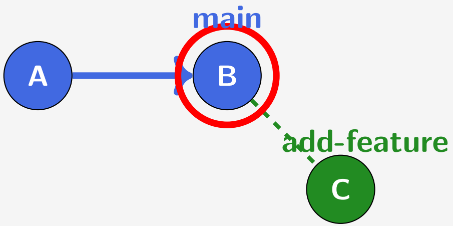
Commit C: "Add cool feature"
New commit on feature branch

Switched to main
Feature not visible here

Changes are isolated to the branch!

Merging a Branch

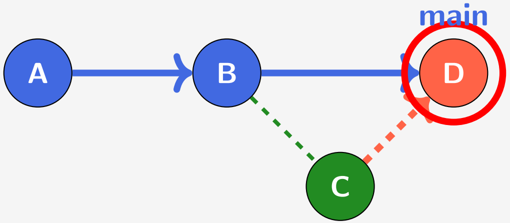
```
# Make sure you're on main
git checkout main

# Merge the feature branch
git merge add-feature

# Now main has the feature!
cat hello.txt
```



On main branch
Ready to merge

Merging a Branch

```
# Make sure you're on main
git checkout main

# Merge the feature branch
git merge add-feature

# Now main has the feature!
cat hello.txt
```



Merge complete!

Commit D combines both branches

```
Hello, Git!
This is my project.
Cool new feature!
```

When to Use Branches?

- ✓ Adding a new feature

When to Use Branches?

- ✓ Adding a new feature
- ✓ Fixing a bug

When to Use Branches?

- ✓ Adding a new feature
- ✓ Fixing a bug
- ✓ Experimenting with something risky

When to Use Branches?

- ✓ Adding a new feature
- ✓ Fixing a bug
- ✓ Experimenting with something risky
- ✓ Each team member works on their own branch

When to Use Branches?

- ✓ Adding a new feature
- ✓ Fixing a bug
- ✓ Experimenting with something risky
- ✓ Each team member works on their own branch

Rule of thumb: Never experiment directly on main!

Collaboration

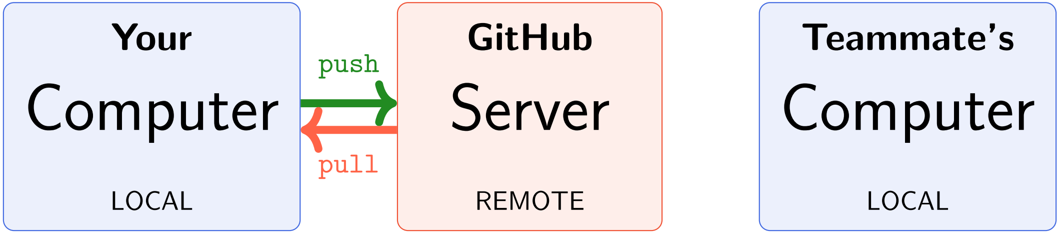
The Big Picture

Your
Computer
LOCAL

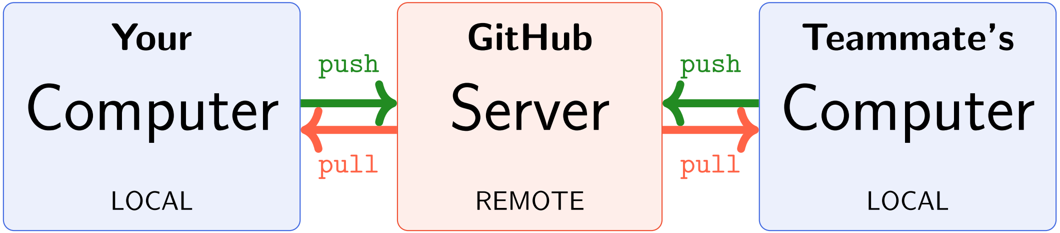
GitHub
Server
REMOTE

Teammate's
Computer
LOCAL

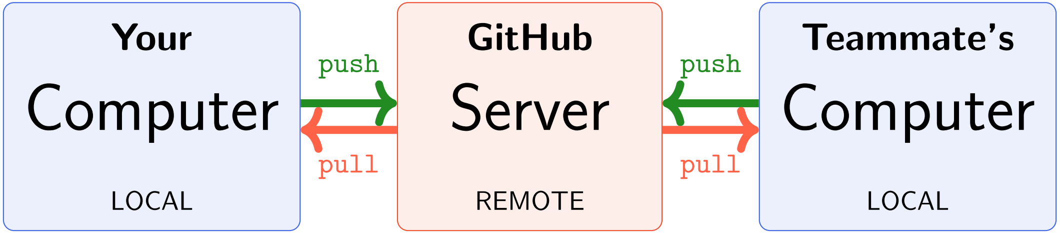
The Big Picture



The Big Picture



The Big Picture



- `git push` — Send your commits to the server
- `git pull` — Get your teammates' commits
- `git clone` — Download a project for the first time

Real-World Scale

- Linux kernel: **25,000+** contributors

Real-World Scale

- Linux kernel: **25,000+** contributors
- You can see the code history of:
  - VS Code
  - React
  - Python itself!

Real-World Scale

- Linux kernel: **25,000+** contributors
- You can see the code history of:
  - VS Code
  - React
  - Python itself!
- `github.com/microsoft/vscode`

Real-World Scale

- Linux kernel: **25,000+** contributors
- You can see the code history of:
  - VS Code
  - React
  - Python itself!
- `github.com/microsoft/vscode`

Almost every dev job interview asks about Git!

You Try!

Mini-Challenge

Complete these steps on your own:

1. Create a new repo on GitHub called `git-practice`
2. Clone it to your computer
3. Create `about_me.txt` with your name
4. Add, commit, and push to GitHub
5. Create a branch called `add-hobbies`
6. Add your hobbies to the file
7. Commit on that branch
8. Merge back to main and push again to github

Raise your hand if you need help!

Common Questions

- **Q: Git vs GitHub?**
- `Git` = tool on your computer. `GitHub` = website to share repos.
  Like video files vs YouTube.
- **Q: Why add then commit? Why two steps?**
- Lets you choose exactly what to save.
  Like selecting which photos go in an album.
- **Q: What if I mess up?**
- `Git` is designed to undo mistakes! Very hard to permanently lose work.

Summary

What We Learned

- Git solves the "**final_FINAL_v3**" problem
- `init`, `add`, `commit` — the basic workflow
- **Branches** let you experiment safely
- Teams use `push`/`pull` to collaborate

```
git init → git add . → git commit -m "message"
```

Next Lecture

Testing & Debugging

- Writing code that checks itself
- Finding bugs like a detective 🔍
- Unit testing fundamentals

Before Next Class:

- Install Git: `git-scm.com`
- Optional: Create a free GitHub account

Questions?