Version Control

Without Version Control

- folder chaos
  - filenames are the only version history you have
- fear of breaking things – or, time lost because things broke
  - you hesitate to try a risky change because there's no way to go back to the current version once you begin
  - attempting to add a new feature or fix a problem broke the code, and the changes were too complex to find them all now
- no memory of why
  - coming back to a project weeks later, you don't remember what all you changed, when changes were made, or why certain things are the way they are

Benefits of Version Control

- for the solo developer – history and backups
  - an annotated history of changes – timestamped and searchable
  - easy undo and safe experimentation
  - pushing to a remote server provides a backup
- for teams – also supports collaboration and automation
  - parallel work without collisions and review before merges
    - team members can work on their own branches simultaneously without overwriting each other's changes
    - pull/merge requests let teammates discuss and approve changes before they are integrated into the main codebase
  - built-in accountability
    - every commit is attributed to its author – easy to see who changed what, and why
  - a foundation for automation
    - commits and branches can trigger automated testing and deployment pipelines
- for large and ongoing projects – manage multiple versions

Key Concepts

- a *snapshot* is the saved state of the entire project at a particular point in time
  - not just a single file!
- a *history* is an ordered sequence of snapshots
  - a chronological timeline, with each snapshot pointing back to the one before it
- a *branch* is an independent line of development
  - lets you test new features, fix bugs, or safely experiment without disturbing the main, stable version of the project
- *revert* undoes changes since a previous snapshot
  - anywhere in the history – not limited to just the immediately prior snapshot

Solo Developer Basic Workflow

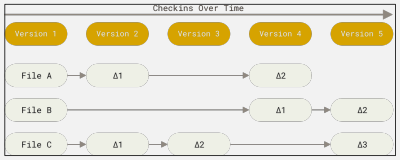
- edit
  - make a focused change – fix a bug, add a feature, tweak a file
- checkpoint
  - once the change is complete and working correctly, save a snapshot along with a short note on what was changed and why
- repeat
  - keeping editing and checkpointing, one clear step at a time
- branch and merge
  - for anything risky, create a new branch before editing, then merge back into the main trunk after checkpointing

Git (and GitHub)

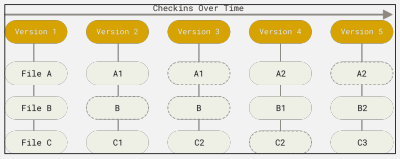
- Git is a distributed version control system
  - originally developed by Linus Torvalds (creator of Linux)
  - by far the most commonly used system for software development
- in a distributed system developers maintain their own local copies of the entire repository
  - typically changes are synchronized via a repository hosted on a central server rather than directly peer-to-peer
- GitHub is a platform hosting Git repositories
  - commonly used for open source software development projects
  - can also host private projects

The Git Mental Model

- snapshots, not differences
  - most version control systems (e.g. CVS, Subversion) store a base set of files along with the series of changes made to each file over time



  - Git stores a series of snapshots
    - for efficiency, Git doesn't store redundant identical copies if a file hasn't changed since the last snapshot

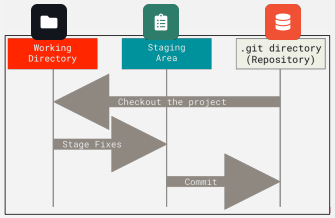


The Git Mental Model

- nearly every operation is local
  - your local repository is a full repository
  - only actions that synchronize with a remote repository aren't fully local
- you have full access to history etc even offline
- you don't automatically see changes others have made to their copy of the repository

The Git Mental Model

- the three areas
  - the *working directory* contains the files you see and edit directly
  - the *staging area* holds a list of the changes to go into the next snapshot
  - the *repository* stored the saved snapshots – it holds the permanent committed history of the project
- basic workflow
  - *modify* files in your working directory
  - *stage* the changes that are to be part of the next commit, adding them to the staging area
  - *commit* the staged changes, saving the snapshot to the repository



https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F

Directory Layout

```
~/cs329                          ← parent folder for all cs329 coursework
  +-- dev/                       ← parent folder for all your Git repositories
  |  +-- lab1/                   ← each project gets its own independent Git repository
  |       + .git/                ← the repository itself
  |       + .gitignore           ← identifies files to keep out of the repository
  |       + src/                 ← Java source code (.java files)
  |       + bin/                 ← Java build output (.class files) – never committed
  + workspace/                   ← Eclipse workspace (bookkeeping only, no files)
     + .metadata/                ← Eclipse's internal state – never committed
```

- Git and Eclipse are kept separate
  - `dev` contains Git repositories
  - `workspace` contains the Eclipse workspace (metadata and potentially project files)
- without Git, project files are contained in separate subdirectories within the Eclipse workspace
- with Git, project files are contained in the working area which is part of Git's space
  - project files don't have to be physically contained within the workspace directory in order for the project to be part of the workspace

Best Practices

- commit early and often – keep commits small
  - save checkpoints frequently rather than bundling a huge work session into a single commit
- atomic commits and separation of concerns
  - changes committed together are forever bundled together
  - focus each commit on a single logical task or issue
  - don't mix different types of changes (e.g. bug fixes, refactoring, formatting updates, new features) in a single commit
- don't commit broken code
- use clear messages
  - explain both what was changed and why
  - be informative but summarize – avoid generic "fixed stuff" but also don't line item each individual edit that is part of one change
- use feature branches for new features and experimental work
  - allows you to maintain small and atomic commits without mixing larger incomplete pieces into the main trunk