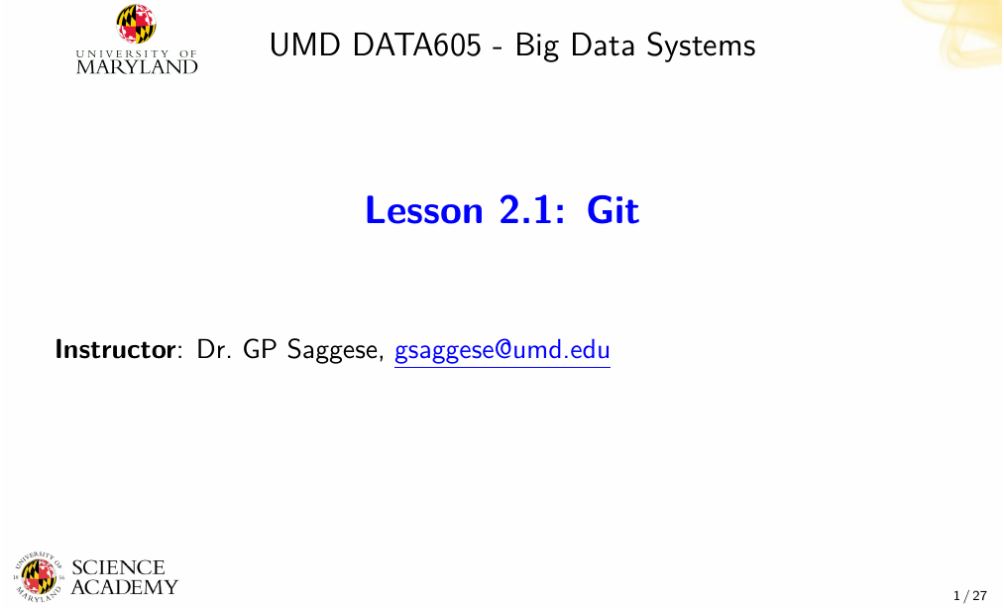
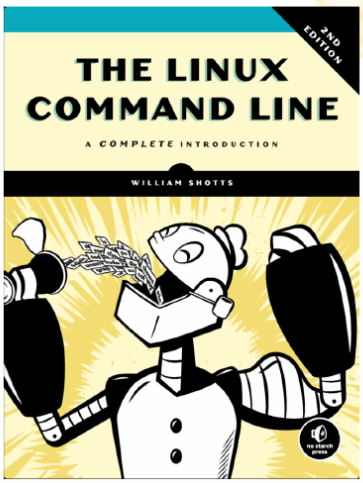
Lesson 2.1: Git


UNIVERSITY OF MARYLAND
UMD DATA605 - Big Data Systems
Lesson 2.1: Git
Instructor: Dr. GP Saggese, gsaggese@umd.edu
SCIENCE ACADEMY
1 / 27

2 / 27: Bash / Linux: Resources

Bash / Linux: Resources

- **How Linux works**
  - Processes
  - File ownership and permissions
  - Virtual memory
  - How to administer a Linux box as root
- **Easy**
  - Command-Line for Beginners
  - E.g., `find`, `xargs`, `chmod`, `chown`, symbolic, and hard links
- **Mastery**
  - The Linux Command Line



SCIENCE ACADEMY

2 / 27

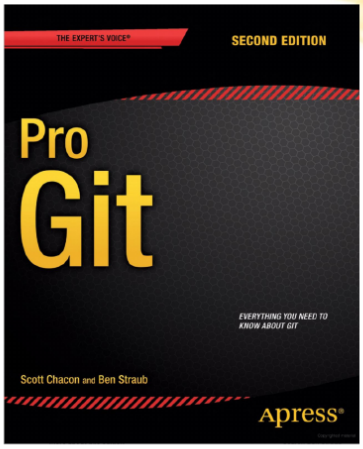
- **How Linux works**
  - *Processes*: In Linux, a process is an instance of a running program. Understanding processes is crucial because they are the basic units of execution in Linux. You can manage processes using commands like `ps`, `top`, and `kill`.
  - *File ownership and permissions*: Every file and directory in Linux has an owner and a set of permissions that determine who can read, write, or execute it. This is important for security and managing access to files.
  - *Virtual memory*: Linux uses virtual memory to extend the physical memory of the system. It allows the system to use disk space as additional RAM, which is essential for running large applications or multiple programs simultaneously.
  - *How to administer a Linux box as root*: The root user has full control over the system. Administering as root involves tasks like installing software, managing users, and configuring system settings. It's important to be cautious when operating as root to avoid accidental system damage.
- **Easy**
  - *Command-Line for Beginners*: This resource is a great starting point for those new to Linux. It covers basic commands and concepts, helping users become comfortable with the command line.
  - *E.g., `find`, `xargs`, `chmod`, `chown`, symbolic, and hard links*: These are fundamental commands and concepts in Linux. `find` helps locate files, `xargs` is used to build and execute command lines, `chmod` and `chown` are for changing file permissions and ownership, and symbolic/hard links are ways to reference files.
- **Mastery**
  - *The Linux Command Line*: This resource is for those who want to deepen their under-

standing of Linux. It covers advanced topics and provides a comprehensive guide to mastering the command line, making it ideal for users who want to become proficient in Linux.

3 / 27: Git Resources

Git Resources

- Concepts in the slides
- Tutorial: Tutorial Git
- We will use Git during the project
- Mastery: Pro Git (free)
- Web resources:
  - https://githowto.com
  - dangitgit.com (without swearing)
  - Oh Sh*t, Git!?! (with swearing)
- Playgrounds
  - https://learngitbranching.js.org



SCIENCE ACADEMY

3 / 27

- **Concepts in the slides**: This bullet point suggests that the slide presentation includes key concepts related to Git, a version control system. Understanding these concepts is crucial for managing code and collaborating with others in software development projects.
- **Tutorial**: The link provided (Tutorial Git) is a resource for a Git tutorial. This tutorial is likely designed to help beginners get started with Git, covering basic commands and workflows.
- **We will use Git during the project**: This indicates that Git will be an essential tool for the upcoming project. Students should familiarize themselves with Git to effectively manage their code and collaborate with team members.
- **Mastery**: The link to Pro Git offers a free, comprehensive guide to mastering Git. This resource is ideal for those who want to deepen their understanding and become proficient in using Git.
- **Web resources**:
  - githowto.com provides step-by-step instructions for learning Git.
  - dangitgit.com offers solutions to common Git problems without using profanity.
  - Oh Sh*t, Git!?! humorously addresses Git issues with a bit of swearing, making it a fun and relatable resource.
- **Playgrounds**: learngitbranching.js.org is an interactive platform where users can practice Git commands and branching strategies in a visual and engaging way. This is particularly useful for hands-on learning and experimentation.

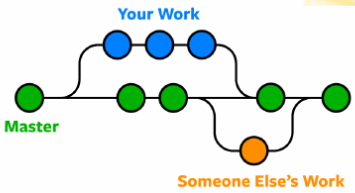
The images on the right side of the slide likely provide visual aids or examples related to the Git

concepts discussed, although they are not visible in this text format.

4 / 27: Git Branching

Git Branching

- **Branching**
  - Diverge from main development line
- **Why branch?**
  - Work without affecting main code
  - Avoid changes in main branch
  - Merge code downstream for updates
  - Merge code upstream after completion
- **Git branching is lightweight**
  - Instantaneous
  - Branch is a pointer to a commit
  - Git stores data as snapshots, not file differences
- **Git workflows branch and merge often**
  - Multiple times a day
  - Surprising for users of distributed VCS
    - E.g., branch before lunch
  - Branches are cheap
    - Use them to isolate and organize work



SCIENCE ACADEMY

4 / 27

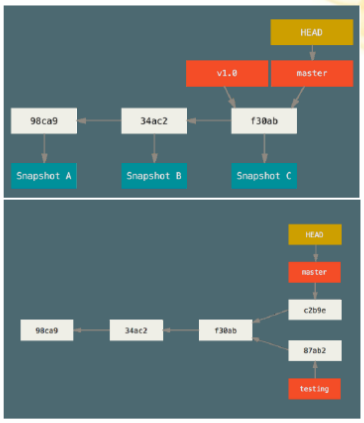
- **Branching**
  - Branching in Git allows developers to create a separate line of development. This means you can work on new features or bug fixes without interfering with the main codebase. It's like creating a parallel universe where you can experiment freely.
- **Why branch?**
  - Branching lets you work independently without affecting the main code. This is crucial for maintaining stability in the main branch, especially in collaborative projects.
  - By keeping changes isolated, you avoid introducing errors into the main branch. This is particularly important in large projects where multiple developers are working simultaneously.
  - Once your work is ready, you can merge your changes downstream to update your branch with the latest code from the main branch. This ensures your work is up-to-date.
  - After completing your work, you can merge your changes upstream into the main branch, integrating your new features or fixes.
- **Git branching is lightweight**
  - Creating a branch in Git is quick and doesn't require much space. It's essentially a pointer to a specific commit, making it efficient.
  - Unlike some other systems, Git stores data as snapshots of the entire project, not just the differences between files. This makes branching and merging fast and reliable.
- **Git workflows branch and merge often**
  - In Git, it's common to branch and merge multiple times a day. This might be surprising for those used to other version control systems, but it's a testament to Git's efficiency.
  - You can create a branch for even small tasks, like before taking a lunch break, to keep your work organized and isolated.

– Since branches are inexpensive to create, they are a great tool for managing different tasks and experiments without cluttering the main codebase.

5 / 27: Git Branching

Git Branching

- `master` (or `main`) is a normal branch
  - Pointer to the last commit
  - Moves forward with each commit
- `HEAD`
  - Pointer to the local branch
  - E.g., `master`, `testing`
  - `git checkout <BRANCH>` moves across branches
- `git branch testing`
  - Create a new pointer `testing`
  - Points to the current commit
  - Pointer is movable
- Divergent history
  - Work progresses in two "split" branches

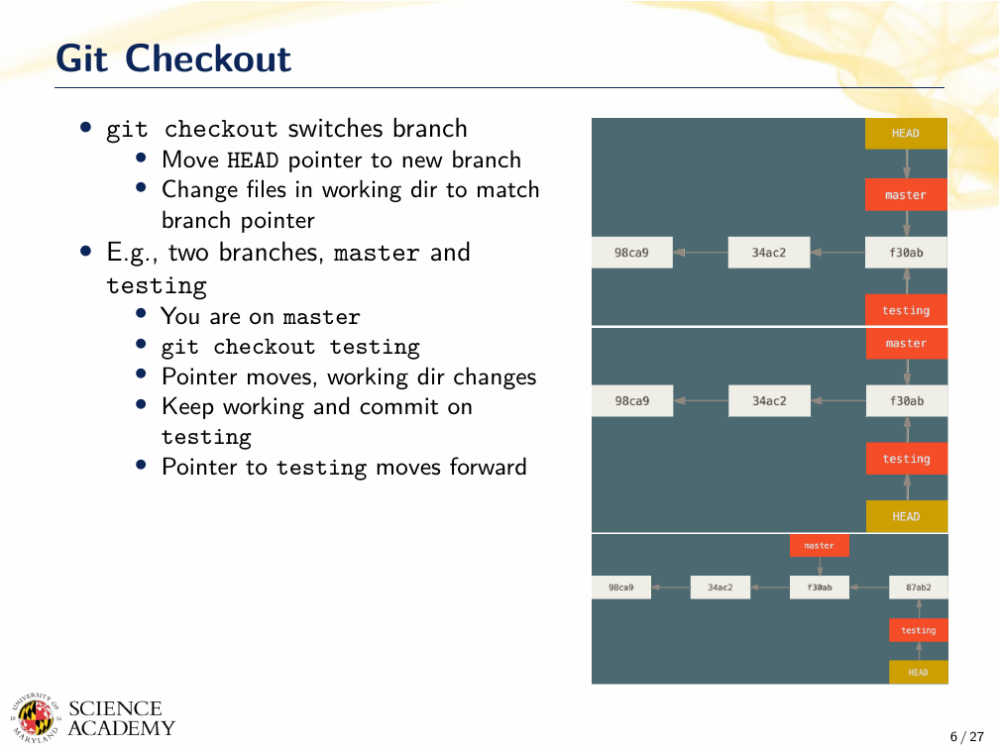


SCIENCE ACADEMY

5 / 27

- **`master` (or `main`) is a normal branch**
  - In Git, the `master` or `main` branch is the default branch where the main line of development occurs. It's essentially a pointer that keeps track of the last commit in the sequence. As you make new commits, this pointer moves forward to include the latest changes, ensuring that the branch always reflects the most recent state of your project.
- **`HEAD`**
  - `HEAD` is a special pointer in Git that represents your current working location in the repository. It usually points to the latest commit on the branch you are working on, such as `master` or `testing`. When you use the command `git checkout <BRANCH>`, you are telling Git to move `HEAD` to point to a different branch, effectively switching your working context to that branch.
- **`git branch testing`**
  - This command creates a new branch named `testing`. It acts as a new pointer that starts at the current commit where `HEAD` is located. This new branch is independent and can be moved forward with new commits, allowing you to experiment or develop features without affecting the `master` branch.
- **Divergent history**
  - When you have multiple branches, such as `master` and `testing`, they can develop independently. This means that each branch can have its own sequence of commits, leading to a "split" or divergent history. This is useful for working on different features or versions of a project simultaneously without interference.

6 / 27: Git Checkout



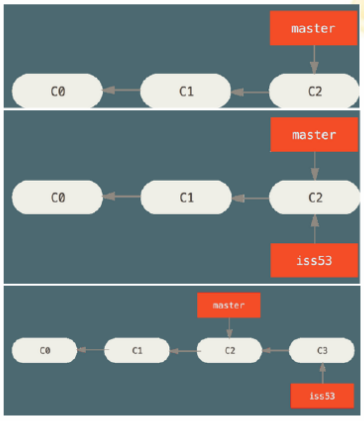
- **`git checkout` switches branch**
  - The `git checkout` command is used to switch between different branches in a Git repository. This is a fundamental operation in Git that allows you to move between different lines of development.
  - When you use `git checkout` to switch branches, it moves the *`HEAD` pointer* to the branch you want to work on. The `HEAD` is a reference to the current branch you are working on.
  - It also changes the files in your working directory to match the state of the branch you have switched to. This means that the files you see and work with will be the ones from the branch you have checked out.
- **Example with two branches, `master` and `testing`**
  - Imagine you have two branches in your repository: `master` and `testing`. You start on the `master` branch.
  - By executing `git checkout testing`, you switch from the `master` branch to the `testing` branch.
  - This action moves the *pointer* to the `testing` branch, and the files in your working directory are updated to reflect the state of the `testing` branch.
  - You can continue to work on the `testing` branch, making changes and committing them. Each commit will move the pointer of the `testing` branch forward, capturing the new state of your work.
- **Images**
  - The images likely illustrate the process of switching branches and how the `HEAD` pointer and working directory change as a result. These visuals can help you understand the concept of branch switching and how it affects your work in Git.

7 / 27: Git Branching and Merging

Git Branching and Merging

- Tutorials
  - Work on main
  - Hot fix
- Start from a project with some commits
- Branch to work on a new feature "Issue 53"

```
> git checkout -b iss53
work ... work ... work
> git commit
```



SCIENCE ACADEMY

7 / 27

- **Git Branching and Merging**: This slide is about managing different versions of a project using Git, a popular version control system. Branching and merging are key concepts in Git that help developers work on different features or fixes simultaneously without interfering with the main project.
- **Tutorials**: The slide provides links to tutorials that guide you through working on the main branch and applying hot fixes. These tutorials are useful for understanding how to manage your code effectively and ensure that changes are integrated smoothly.
- **Start from a project with some commits**: Before branching, it's important to have a project with existing commits. This means you have a history of changes that have been made to the project, which is essential for tracking progress and understanding the context of your work.
- **Branch to work on a new feature "Issue 53"**: The slide demonstrates how to create a new branch to work on a specific feature or issue. By using the command `git checkout -b iss53`, you create and switch to a new branch named "iss53". This allows you to work on the feature independently, making changes and committing them without affecting the main branch. This is crucial for maintaining a clean and organized workflow, especially in collaborative environments.
- **Images**: The images likely illustrate the process of branching and merging visually, helping to reinforce the concepts discussed. Visual aids can be very helpful in understanding how branches diverge and merge back into the main project.

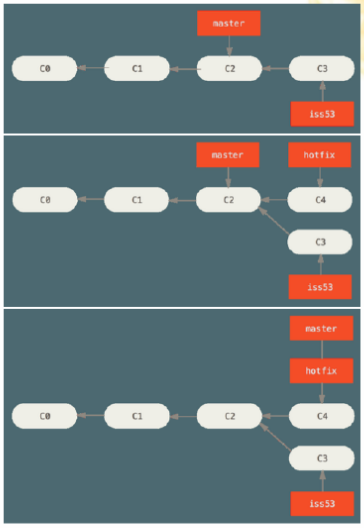
8 / 27: Git Branching and Merging

Git Branching and Merging

- **Need a hotfix** to `master`

```
> git checkout master
> git checkout -b hotfix
fix ... fix ... fix
> git commit -am "Hot fix"
> git checkout master
> git merge hotfix
```

- **Fast forward**
  - Now there is a divergent history between `master` and `iss53`



SCIENCE ACADEMY

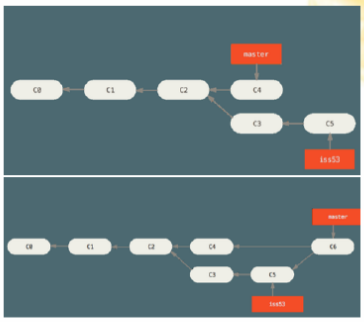
8 / 27

- **Need a hotfix** to `master`
  - Sometimes, you might encounter a situation where an urgent fix is needed in the main branch of your project, often called `master`. This is known as a *hotfix*.
  - The process begins by switching to the `master` branch using the command `git checkout master`.
  - Next, you create a new branch specifically for the hotfix with `git checkout -b hotfix`. This allows you to work on the fix without affecting the main branch until it's ready.
  - After making the necessary changes, you commit them with a message, for example, `git commit -am "Hot fix"`.
  - Finally, you merge the hotfix back into the `master` branch using `git merge hotfix`. This incorporates the changes into the main branch, resolving the issue.
- **Fast forward**
  - In Git, a *fast forward* occurs when you merge branches and there is no divergent history between them. However, in this case, there is a divergent history between `master` and another branch named `iss53`.
  - This means that changes have been made in both branches that need to be reconciled. Understanding how to handle these situations is crucial for maintaining a clean and functional project history.

9 / 27: Git Branching and Merging

Git Branching and Merging

- Keep working on `iss53`
  `> git checkout iss53`
  `work ... work ... work`
  - The branch keeps diverging
- At some point you are done with `iss53`
  - You want to merge your work back to `master`
  - Go to the target branch
    `> git checkout master`
    `> git merge iss53`



- Git can't fast forward
- Git creates a new snapshot with the 3-way "merge commit" (i.e., a commit with more than one parent)
- Delete the branch

  `> git branch -d iss53`

SCIENCE ACADEMY

9 / 27

- **Keep working on `iss53`**
  - This step involves switching to the branch named `iss53` using the command `git checkout iss53`. Once on this branch, you continue making changes and committing them. This is a common practice when working on a specific feature or issue, allowing you to isolate your work from the main codebase.
  - **The branch keeps diverging**: As you work on `iss53`, the branch may diverge from the `master` branch if other changes are made to `master` in the meantime. This means the two branches have different histories.
- **At some point you are done with `iss53`**
  - Once your work on `iss53` is complete, you need to integrate these changes back into the main branch, typically `master`. This is done by first switching to the `master` branch using `git checkout master`.
  - **Go to the target branch**: After switching to `master`, you merge the changes from `iss53` into `master` using `git merge iss53`.
- **Git can't fast forward**
  - If Git cannot perform a fast-forward merge, it means that the `master` branch has progressed since `iss53` was created. In such cases, Git creates a new commit called a "merge commit" to combine the histories of both branches.
- **Git creates a new snapshot with the 3-way "merge commit"**
  - A merge commit is a special type of commit that has more than one parent. It represents the point where two branches have been combined, preserving the history of both branches.
- **Delete the branch**
  - After successfully merging `iss53` into `master`, the branch `iss53` is no longer needed. You

can delete it using `git branch -d iss53` to keep your branch list clean and organized. This step is important for maintaining a tidy repository and avoiding clutter from unused branches.

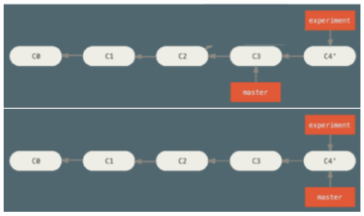
10 / 27: Fast Forward Merge

Fast Forward Merge

- **Fast forward merge**
  - Merge a commit `X` with a commit `Y` that can be reached by following the history of commit `X`
- There is not divergent history to merge
  - Git simply moves the branch pointer forward from X to Y
- **Mental model**: a branch is just a pointer that says where the tip of the branch is
- E.g., `C4'` is reachable from `C3`

```
> git checkout master
> git merge experiment
```

- Git moves the pointer of master to `C4'`

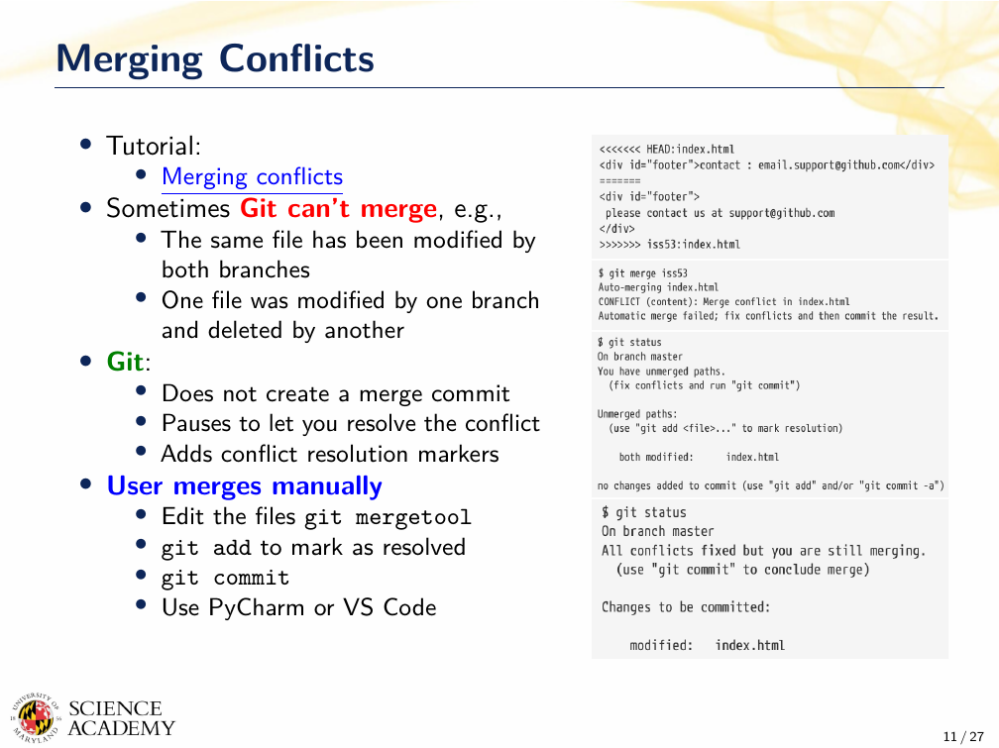


SCIENCE ACADEMY

10 / 27

- **Fast forward merge**
  - When we talk about a *fast forward merge*, we're referring to a situation where you want to merge two commits, `X` and `Y`. In this case, commit `Y` is directly reachable from commit `X` by simply following the history. This means there are no other changes or branches that have diverged between these two commits.
- **There is not divergent history to merge**
  - In a fast forward merge, Git doesn't have to do any complex merging work. Instead, it just updates the branch pointer to the latest commit. So, if you were on commit `X` and you want to merge with commit `Y`, Git will just move the branch pointer from `X` to `Y`.
- **Mental model**: a branch is just a pointer that says where the tip of the branch is
  - Think of a branch in Git as a simple pointer. It points to the latest commit in that branch. For example, if you have a branch with commits `C1`, `C2`, `C3`, and `C4'`, and you want to merge `C3` with `C4'`, Git will just move the branch pointer from `C3` to `C4'`.
- **Example command**
  - The command `git checkout master` followed by `git merge experiment` is an example of how you might perform a fast forward merge. Here, `master` is the branch you're on, and `experiment` is the branch you want to merge into `master`. If `experiment` is ahead of `master` with no divergent changes, Git will simply move the `master` pointer forward to the latest commit on `experiment`.
- **Visual aids**
  - The images provided likely illustrate the concept of a fast forward merge by showing how the branch pointer moves from one commit to another without any branching or merging conflicts. This visual representation helps in understanding how straightforward a fast forward merge is compared to other types of merges.

11 / 27: Merging Conflicts



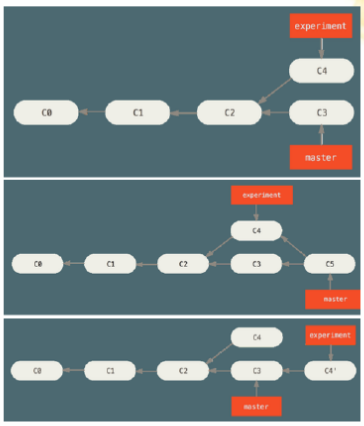
- **Merging Conflicts**: This slide is about handling situations where Git, a version control system, encounters conflicts while trying to merge changes from different branches. Merging conflicts occur when Git cannot automatically combine changes because the same file has been altered in different ways in separate branches, or when one branch modifies a file that another branch deletes.
- **Git's Behavior**:
  - When Git encounters a conflict, it does not automatically create a merge commit. Instead, it pauses the process to allow the user to manually resolve the conflict.
  - Git adds conflict resolution markers in the files to highlight the conflicting sections. These markers help users identify the parts of the code that need attention.
- **User's Role in Conflict Resolution**:
  - Users must manually edit the files to resolve conflicts. This can be done using tools like `git mergetool`, which assists in comparing and editing the conflicting files.
  - After resolving the conflicts, users need to use `git add` to mark the conflicts as resolved, followed by `git commit` to finalize the merge.
  - Integrated development environments (IDEs) like PyCharm or VS Code can be used to simplify the process of resolving merge conflicts, providing a more visual and user-friendly interface.

The images on the right likely illustrate examples of merge conflicts and how they appear in different tools, helping to visualize the process described.

12 / 27: Git Rebasing

Git Rebasing

- In Git there are **two ways of merging divergent history**
  - E.g., `master` and `experiment` have a common ancestor `C2`
- **Merge**
  - Go to the target branch
    `> git checkout master`
    `> git merge experiment`
  - Create a new snapshot `C5` and commit
- **Rebase**
  - Go to the branch to rebase
    `> git checkout experiment`
    `> git rebase master`
  - Rebase algo:
    - Get all the changes committed in the branch (`C4`) where we are on (`experiment`) since the common ancestor (`C2`)
    - Sync to the branch that we are rebasing onto (`master` at `C3`)
    - Apply the changes `C4`
    - Only current branch is affected
    - Finally fast forward `experiment`



SCIENCE ACADEMY

12 / 27

- **Git Rebasing**
  - In Git, there are **two ways of merging divergent history**. This means when you have two branches that have developed separately from a common starting point, you can bring them back together using either merging or rebasing. For example, if you have a `master` branch and an `experiment` branch that both originated from a common commit `C2`, you can use these methods to combine their histories.
- **Merge**
  - To merge, you first **go to the target branch** where you want to combine the changes. In this case, you switch to the `master` branch using the command:

```
> git checkout master
> git merge experiment
```

  - This process creates a new commit, `C5`, which represents the combined history of both branches. This new commit is a snapshot that includes all changes from both branches.
- **Rebase**
  - Rebasing involves **going to the branch you want to rebase**. Here, you switch to the `experiment` branch:

```
> git checkout experiment
> git rebase master
```

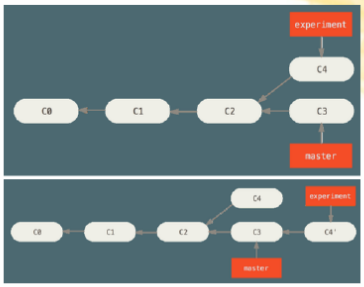
  - The rebase algorithm works by taking all the changes made in the `experiment` branch since the common ancestor `C2` (in this case, changes in `C4`), and then applying them on top of the `master` branch at `C3`. This effectively moves the base of the `experiment` branch to the tip of the `master` branch.
  - Only the current branch (`experiment`) is affected by this operation, and it results in a

cleaner project history by avoiding unnecessary merge commits.
- Finally, the `experiment` branch is fast-forwarded, meaning it now appears as if it was developed directly from the latest commit on `master`.

13 / 27: Uses of Rebase

Uses of Rebase

- **Rebasing makes for a cleaner history**
  - The history looks like all the work happened in series
  - Although in reality it happened in parallel to the development in master
- **Rebasing to contribute to a project**
  - Developer
    - You are contributing to a project that you don't maintain
    - You work on your branch
    - When you are ready to integrate your work, rebase your work onto `origin/master`
  - The maintainer
    - Does not have to do any integration work
    - Does just a fast forward or a clean apply (no conflicts)



SCIENCE ACADEMY

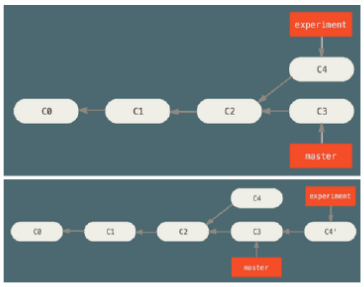
13 / 27

- **Rebasing makes for a cleaner history**
  - When you rebase, it rearranges the commit history so that it appears as if all the work was done one after the other, in a straight line. This is different from what actually happens, where multiple developers might be working on different features at the same time. By making the history linear, it becomes easier to follow and understand the sequence of changes.
  - *In reality*, development often happens in parallel, with different branches being worked on simultaneously. Rebasing helps to present this parallel work in a more organized and sequential manner.
- **Rebasing to contribute to a project**
  - *Developer*: If you're contributing to a project that you don't own or maintain, you typically work on your own branch. Once your work is ready to be added to the main project, you rebase your branch onto the latest version of the main branch (often called `origin/master`). This ensures your changes are up-to-date with the latest project developments.
  - *The maintainer*: By rebasing before you submit your changes, you make it easier for the project maintainer. They can simply fast-forward the main branch to include your changes or apply them cleanly without having to resolve conflicts. This streamlines the integration process and reduces the workload for the maintainer.

14 / 27: Golden Rule of Rebasing

Golden Rule of Rebasing

- **Remember**: rebasing means abandoning existing commits and creating new ones that are similar but different
- **Problem**
  - You push commits to a remote
  - Others pull commits and base work on them
  - You rewrite commits with `git rebase`
  - You push again with `git push --force`
  - Collaborators must re-merge work
- **Solution**
  - Strict: *"Do not ever rebase commits outside your repository"*
  - Loose: *"Rebase your branch if only you use it, even if pushed to a server"*



SCIENCE ACADEMY

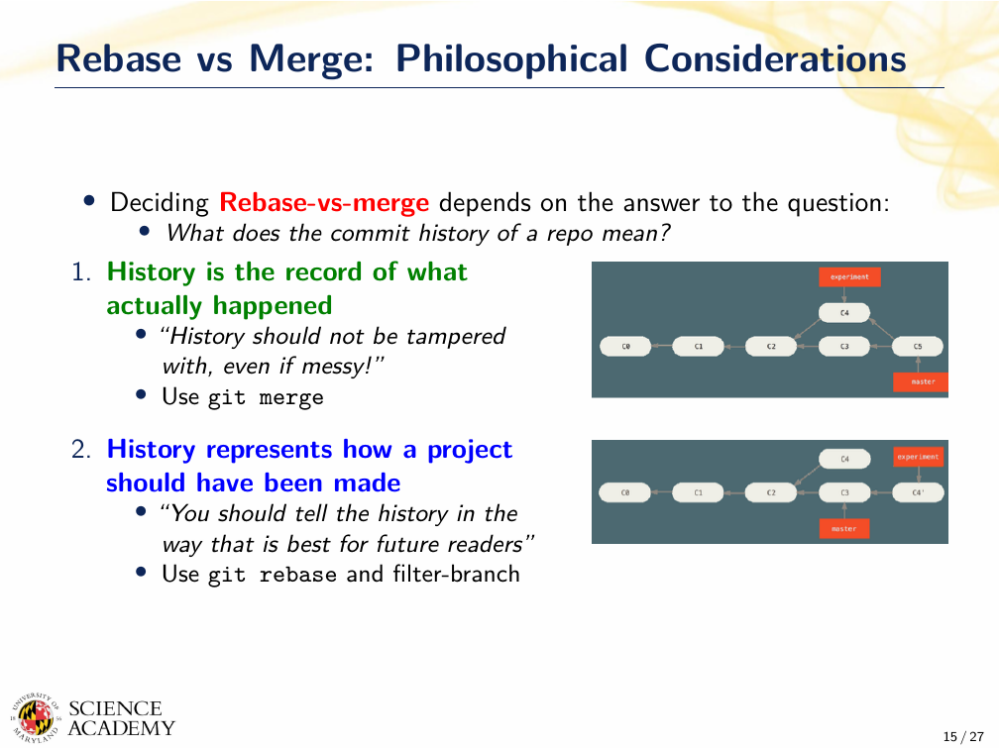
14 / 27

- **Golden Rule of Rebasing**
  - **Remember**: When you rebase, you are essentially taking existing commits and creating new ones that are similar but not identical. This means the history of your project changes, which can lead to complications if not handled carefully.
- **Problem**
  - **You push commits to a remote**: This means you've shared your work with others, and they might start using it as a base for their own work.
  - **Others pull commits and base work on them**: Your collaborators download your changes and start building their own work on top of it.
  - **You rewrite commits with `git rebase`**: Rebasing changes the commit history, which can make the history look cleaner but also changes the commit IDs.
  - **You push again with `git push --force`**: This action forces the new commit history onto the remote, overwriting the old history.
  - **Collaborators must re-merge work**: Because the commit history has changed, your collaborators will have to adjust their work to fit the new history, which can be time-consuming and error-prone.
- **Solution**
  - **Strict**: *"Do not ever rebase commits outside your repository"*: This means you should avoid rebasing commits that have already been shared with others to prevent disrupting their work.
  - **Loose**: *"Rebase your branch if only you use it, even if pushed to a server"*: If you are the only one working on a branch, it's generally safe to rebase, even if the branch is on a remote server, because no one else is affected by the changes.

The images likely illustrate the difference between a clean commit history with rebasing and a

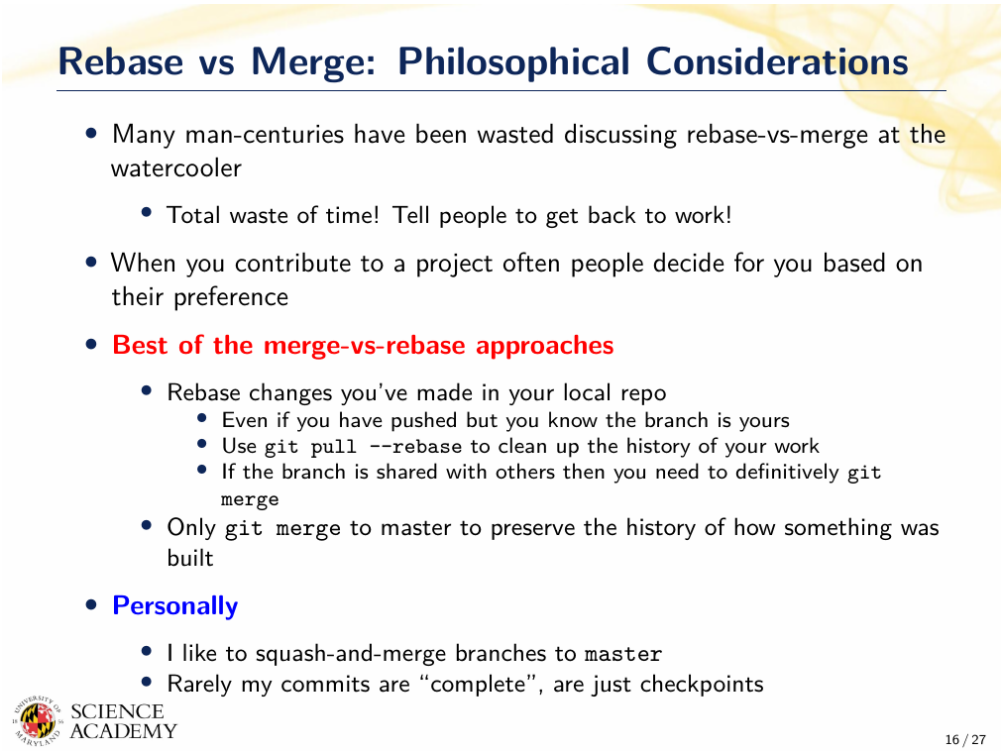
more complex one without it, emphasizing the importance of understanding when and how to use rebasing effectively.

15 / 27: Rebase vs Merge: Philosophical Considerations



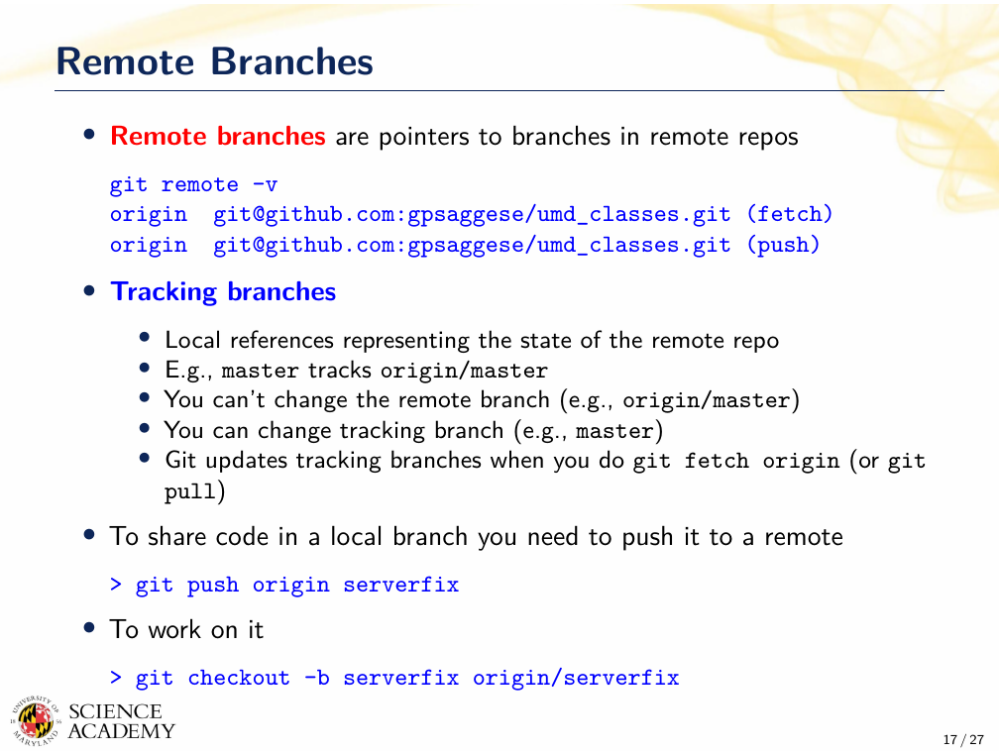
- **Rebase vs Merge: Philosophical Considerations**
  - When working with Git, a version control system, you often face the decision of whether to use **rebase** or **merge**. This decision is not just technical but also philosophical, as it relates to how you perceive the commit history of a repository.
  - The key question to consider is: *What does the commit history of a repo mean?*
- **History is the record of what actually happened**
  - This perspective values the authenticity of the commit history. It suggests that the history should reflect the actual sequence of events, even if it appears messy or complex.
  - If you believe in preserving the true sequence of events, you should use `git merge`. This approach keeps all the original commits intact and shows how the project evolved over time, including all branches and merges.
- **History represents how a project should have been made**
  - This viewpoint emphasizes clarity and readability for future developers. It suggests that the commit history should be organized in a way that makes it easy to understand the project's development process.
  - If you prefer a clean and linear history, you should use `git rebase` and possibly `filter-branch`. This approach allows you to rewrite the commit history to present a more streamlined and coherent narrative, which can be beneficial for new team members or when reviewing the project's evolution.
- **Conclusion**
  - The choice between rebase and merge is not just about technical differences but also about how you want to present the history of your project. Consider your team's needs and the importance of clarity versus authenticity when making this decision.

16 / 27: Rebase vs Merge: Philosophical Considerations



- **Rebase vs Merge: Philosophical Considerations**
  - The debate between using *rebase* or *merge* in version control systems like Git is a common topic among developers. It's often discussed at length, sometimes humorously referred to as a waste of time, as it can distract from actual work. The key takeaway is that while these discussions can be engaging, they shouldn't overshadow productivity.
- **Project Contribution Decisions**
  - When contributing to a project, the choice between rebase and merge is often made by the project maintainers based on their preferences. This means that as a contributor, you might not always have the freedom to choose your preferred method.
- **Best of the merge-vs-rebase approaches**
  - **Rebase**: This is useful for cleaning up your local commit history. If you're working on a branch that is solely yours, you can rebase even after pushing. Using `git pull --rebase` helps keep your work history tidy.
  - **Merge**: When working on a shared branch, merging is safer to avoid conflicts. Merging into the master branch is recommended to maintain a clear history of how the project evolved.
- **Personally**
  - The speaker prefers to use a *squash-and-merge* approach when integrating branches into the master. This method combines all changes into a single commit, which is useful when individual commits are more like checkpoints rather than complete features. This approach helps keep the master branch clean and organized.

17 / 27: Remote Branches



- **Remote branches** are essentially bookmarks that point to the branches in a remote repository. They help you keep track of the state of branches in a repository that is hosted somewhere else, like on GitHub. When you run the command `git remote -v`, it shows you the URLs of the remote repositories you have set up, both for fetching and pushing changes. This is crucial for collaboration, as it allows multiple people to work on the same project from different locations.
- **Tracking branches** are local branches that are set up to track the state of a branch in a remote repository. For example, your local `master` branch might track `origin/master`, which is the master branch on the remote repository. While you can't directly change a remote branch like `origin/master`, you can make changes to your local tracking branch, such as `master`. When you run `git fetch origin` or `git pull`, Git updates your tracking branches to reflect the latest state of the remote branches.
- To share your work from a local branch with others, you need to push it to a remote repository. For instance, if you have a local branch named `serverfix`, you can share it by executing `git push origin serverfix`. This command sends your local changes to the remote repository, making them accessible to others.
- If you want to start working on a branch that exists on the remote repository but not locally, you can create a new local branch that tracks the remote branch. For example, using `git checkout -b serverfix origin/serverfix` creates a new local branch named `serverfix` that tracks the `serverfix` branch on the remote repository. This allows you to work on the branch locally and later push your changes back to the remote.

18 / 27: Git Workflows

Git Workflows

- **Git workflows** = ways of working and collaborating using Git
- **Long-running branches** = branches at different level of stabilities, that are always open
  - `master` is always ready to be released
  - `develop` branch to develop in
  - topic / feature branches
  - When branches are "stable enough" they are merged up

SCIENCE ACADEMY

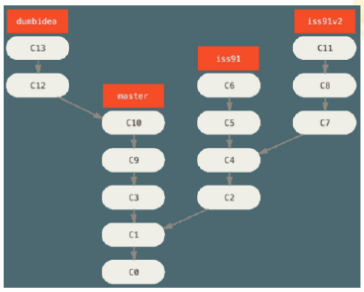
18 / 27

- **Git workflows**: These are structured methods for using Git, a version control system, to manage and collaborate on projects. Git workflows help teams organize their work, manage changes, and ensure that everyone is on the same page. They provide a framework for how code is developed, reviewed, and integrated into the main project.
- **Long-running branches**: These are branches in a Git repository that remain open for extended periods. They serve different purposes and have varying levels of stability:
  - **`master` branch**: This is the main branch that is always ready for release. It contains the most stable version of the project.
  - **`develop` branch**: This branch is used for ongoing development. It acts as an integration branch for features and fixes before they are considered stable enough for the `master`.
  - **Topic/feature branches**: These are short-lived branches created for specific features or fixes. Developers work on these branches independently and merge them into the `develop` branch once they are stable.
  - The process of merging branches ensures that only stable and tested code is integrated into the main branches, maintaining the project's overall stability.

19 / 27: Git Workflows

Git Workflows

- **Topic branches** = short-lived branches for a single feature
  - E.g., `hotfix`, `wip-XYZ`
  - Easy to review
  - Silo-ed from the rest
  - This is typical of Git since other VCS support for branches is not good enough
  - E.g.,
    - You start `iss91`, then you cancel some stuff, and go to `iss91v2`
    - Somebody starts `dumbidea` branch and merge to `master` (!)
    - You squash-and-merge your `iss91v2`



SCIENCE ACADEMY

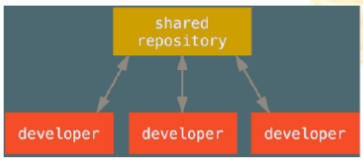
19 / 27

- **Topic branches** are a key concept in Git workflows. They are *short-lived branches* created specifically for working on a single feature or fix. This allows developers to work independently on different tasks without interfering with the main codebase.
  - For example, you might create a branch named `hotfix` to quickly address a bug or `wip-XYZ` for a work-in-progress feature. These branches are temporary and usually deleted after their changes are merged into the main branch.
  - **Easy to review**: Because topic branches are focused on a single feature or fix, they contain fewer changes, making it easier for others to review the code.
  - **Silo-ed from the rest**: These branches are isolated from the main codebase, reducing the risk of introducing errors into the main branch until the feature is complete and tested.
  - Git excels at handling branches compared to other version control systems (VCS), which may not support branching as effectively.
  - For instance, you might start working on a branch `iss91`, realize some changes are needed, and then create a new branch `iss91v2` to continue your work. Meanwhile, someone else might create a `dumbidea` branch and mistakenly merge it into `master`, highlighting the importance of careful branch management. Finally, you might use a *squash-and-merge* strategy to combine your `iss91v2` changes into the main branch, keeping the commit history clean.

20 / 27: Centralized Workflow

Centralized Workflow

- **Centralized workflow in centralized VCS**
  - Developers:
    - Check out the code from the central repo on their computer
    - Modify the code locally
    - Push it back to the central hub (assuming no conflicts with latest copy, otherwise they need to merge)
- **Centralized workflow in Git**
  - Developers:
    - Have push (i.e., write) access to the central repo
    - Need to fetch and then merge
    - Cannot push code that will overwrite each other code (only fast-forward changes)

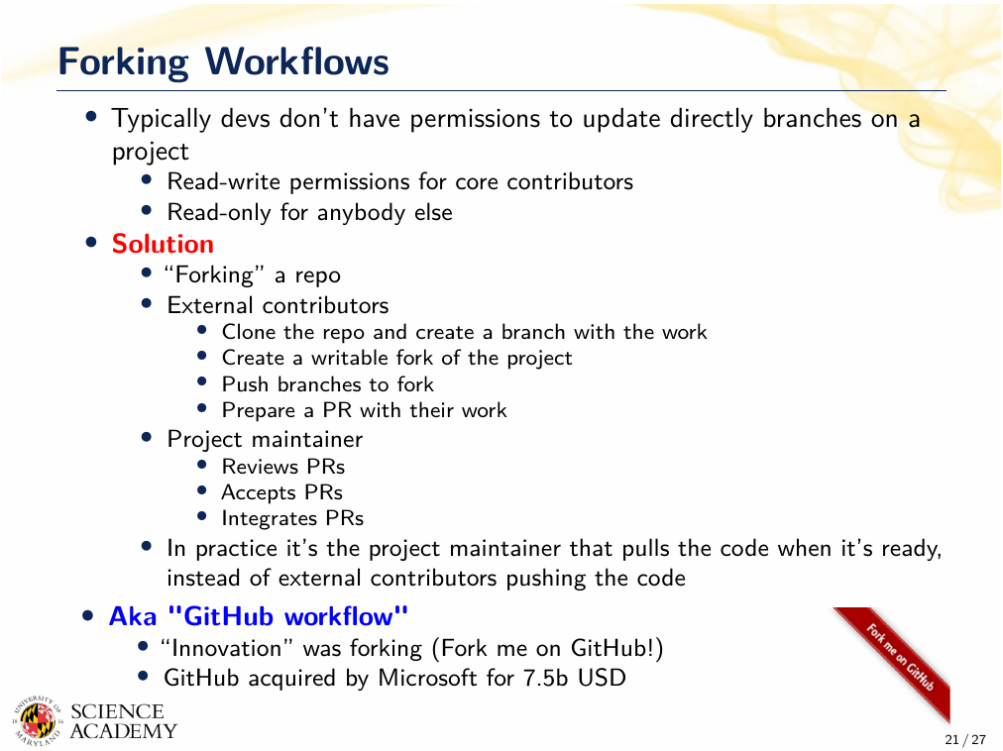


SCIENCE ACADEMY

20 / 27

- **Centralized workflow in centralized VCS**
  - *Developers*:
    - **Check out the code from the central repo on their computer**: In a centralized version control system (VCS), there is a single central repository where all the code is stored. Developers need to check out or download the code from this central location to their local machines to work on it.
    - **Modify the code locally**: Once the code is on their local machine, developers can make changes and improvements as needed.
    - **Push it back to the central hub**: After making changes, developers push their modified code back to the central repository. If someone else has made changes to the same part of the code, developers must resolve these conflicts before pushing.
- **Centralized workflow in Git**
  - *Developers*:
    - **Have push (i.e., write) access to the central repo**: In Git, developers need permission to push changes to the central repository. This ensures that only authorized users can update the main codebase.
    - **Need to fetch and then merge**: Before pushing changes, developers must fetch the latest version of the code from the central repository and merge it with their local changes. This helps prevent conflicts and ensures that the codebase is up-to-date.
    - **Cannot push code that will overwrite each other code (only fast-forward changes)**: Git prevents developers from pushing changes that would overwrite others' work. Only changes that can be fast-forwarded, meaning they don't conflict with the current state of the repository, can be pushed directly. This helps maintain the integrity of the codebase.

21 / 27: Forking Workflows



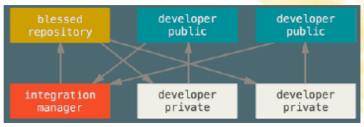
- **Forking Workflows**
  - In many open-source projects, developers usually don't have the permission to directly update the main branches of a project. This is to maintain the integrity and stability of the codebase.
    - *Core contributors* have read-write permissions, meaning they can make changes directly to the project.
    - Everyone else has read-only access, which means they can view the code but cannot make changes directly.
- **Solution**
  - The concept of "forking" a repository allows external contributors to work on a project without having direct write access.
    - External contributors can clone the repository and create their own branch to work on.
    - They create a writable fork, which is essentially a personal copy of the project where they can make changes.
    - Once their work is ready, they push their changes to their fork and prepare a Pull Request (PR) to propose their changes to the original project.
  - The project maintainer plays a crucial role in this workflow.
    - They review the PRs submitted by external contributors.
    - If the changes are satisfactory, they accept and integrate the PRs into the main project.
    - Typically, the maintainer pulls the code into the main project, rather than the contributor pushing it directly.
- **Aka "GitHub workflow"**

- This workflow is often referred to as the "GitHub workflow" because GitHub popularized the concept of forking with their platform.
- The ability to fork repositories and collaborate in this way was a significant innovation, leading to the phrase "Fork me on GitHub!"
- GitHub's success and influence in the software development community were highlighted by its acquisition by Microsoft for $7.5 billion USD. This underscores the importance and value of collaborative coding platforms.

22 / 27: Integration-Manager Workflow

Integration-Manager Workflow

- This is the classical model for open-source development
  - E.g., Linux, GitHub (forking) workflow



1. **One repo is the official project**
   - Only the project maintainer pushes to the public repo
   - E.g., `causify-ai/csfy`
2. **Each contributor**
   - Has read access to everyone else's public repo
   - Forks the project into a private copy
     - Write access to their own public repo
     - E.g., `gpsaggese/csfy`
   - Makes changes
   - Pushes changes to his own public copy
   - Sends email to maintainer asking to pull changes (pull request)
3. **The maintainer**
   - Adds contributor repo as a remote
   - Merges the changes into a local branch
   - Tests changes locally
   - Pushes branch to the official repo

SCIENCE ACADEMY

22 / 27

- **Integration-Manager Workflow**
  This workflow is a traditional model used in open-source development. It's commonly seen in projects like Linux and on platforms like GitHub, where the concept of forking is prevalent. This model helps manage contributions from multiple developers efficiently.

- **One repo is the official project**
  In this workflow, there is a single official repository for the project. Only the project maintainer has the authority to push changes directly to this public repository. For example, in a project named `causify-ai/csfy`, the maintainer would be the only one updating the official repo.

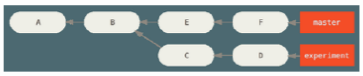
- **Each contributor**
  Contributors have read access to all public repositories, allowing them to see what others are working on. They fork the official project, creating their own private copy where they have write access. For instance, a contributor might fork the project to `gpsaggese/csfy`. They make changes in their forked copy and push these changes to their public repo. Once satisfied, they send a pull request to the maintainer, asking for their changes to be incorporated into the official project.

- **The maintainer**
  The maintainer plays a crucial role in this workflow. They add the contributor's repository as a remote to their local setup. After merging the proposed changes into a local branch, they test these changes to ensure they work correctly. Once verified, the maintainer pushes the updated branch to the official repository, integrating the contributor's work into the project. This process ensures that all changes are reviewed and tested before becoming part of the official codebase.

23 / 27: Git log

Git log

- `git log` reports info about commits
- refs are references to:
  - `HEAD` (commit you are working on, next commit)
  - `origin/master` (remote branch)
  - `experiment` (local branch)
  - `d921970` (commit)
- `^` after a reference resolves to the parent of that commit
  - `HEAD^` = commit before `HEAD`, i.e., last commit
  - `^2` means `^^`
  - A merge commit has multiple parents



SCIENCE ACADEMY

23 / 27

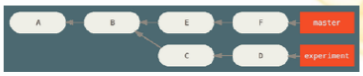
- **Git log**
  - The `git log` command is a tool used in Git to display a history of commits. This is useful for tracking changes and understanding the evolution of a project over time. Each commit in the log contains information such as the author, date, and a message describing the changes made.
- **Refs (References)**
  - *Refs* are pointers to specific commits or branches in a Git repository. They help you navigate through different points in the project's history.
  - **HEAD**: This is a special reference that points to the current commit you are working on. It represents the latest state of your working directory and is where your next commit will be based.
  - **origin/master**: This refers to the remote branch named `master`. It is the version of the branch stored on a remote server, often used to collaborate with others.
  - **experiment**: This is an example of a local branch. Local branches are used to work on different features or fixes without affecting the main codebase.
  - **d921970**: This is an example of a specific commit identified by its unique hash. Each commit in Git has a unique identifier, which allows you to reference it directly.
- **Caret (^) Notation**
  - The caret symbol (`^`) is used to navigate through commit history. It helps you identify parent commits.
  - `HEAD^` refers to the commit immediately before the current `HEAD`. This is useful for looking at the previous state of the project.
  - `^2` is shorthand for `^^`, which means the second parent of a commit. This is particularly relevant for merge commits, which have multiple parent commits due to the merging of

branches.

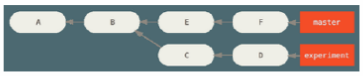
24 / 27: Dot notation

Dot notation

- **Double-dot notation**
  - `1..2` = commits that are reachable from 2 but not from 1
  - Like a "difference"
  - `git log master..experiment` → `D,C`
  - `git log experiment..master`→ `F,E`



- **Triple-dot notation**
  - `1...2` = commits that are reachable from either branch but not from both
  - Like "union excluding intersection"
  - `git log master...experiment` → `F,E,D,C`

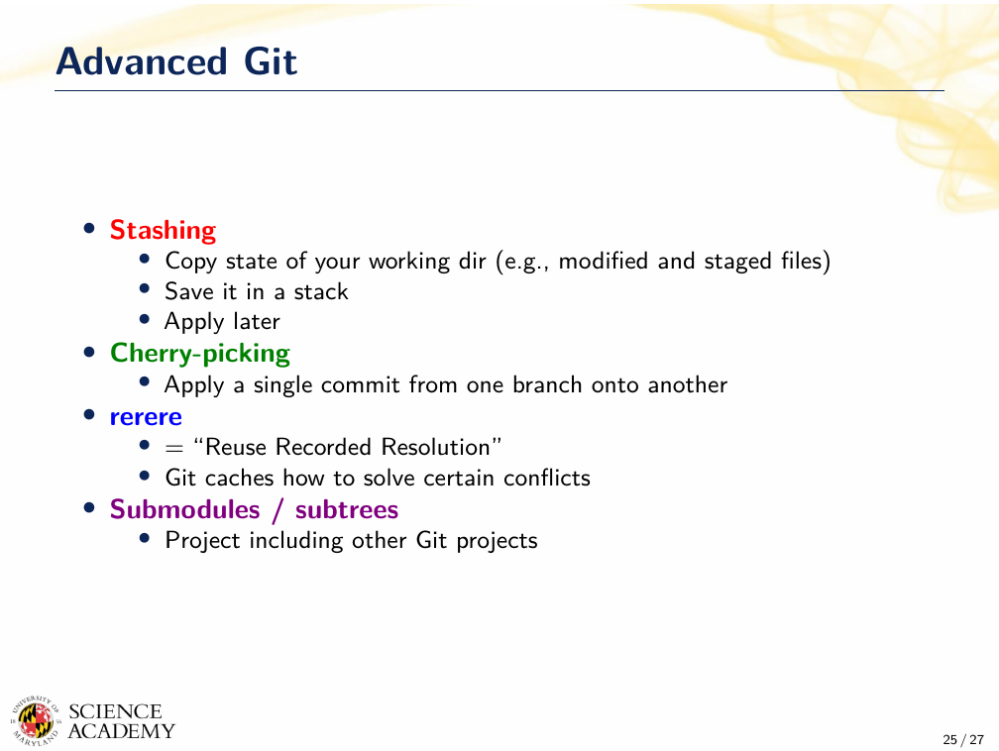


SCIENCE ACADEMY

24 / 27

- **Double-dot notation**
  - The double-dot notation is a way to compare two branches or commits in Git. When you see something like `1..2`, it means you're looking at the commits that are reachable from the second commit (or branch) but not from the first. Think of it like finding the *difference* between two sets.
  - For example, if you run `git log master..experiment`, Git will show you the commits that are in the `experiment` branch but not in the `master` branch. In this case, it would show commits `D` and `C`.
  - Conversely, `git log experiment..master` will show you the commits that are in `master` but not in `experiment`, which are `F` and `E`.
- **Triple-dot notation**
  - The triple-dot notation is slightly different. When you see `1...2`, it refers to the commits that are reachable from either of the two branches but not from both. It's like taking the *union* of the two sets and then excluding the *intersection*.
  - For instance, `git log master...experiment` will list all the commits that are unique to either `master` or `experiment`, but not those that are common to both. In this example, it would show commits `F`, `E`, `D`, and `C`.

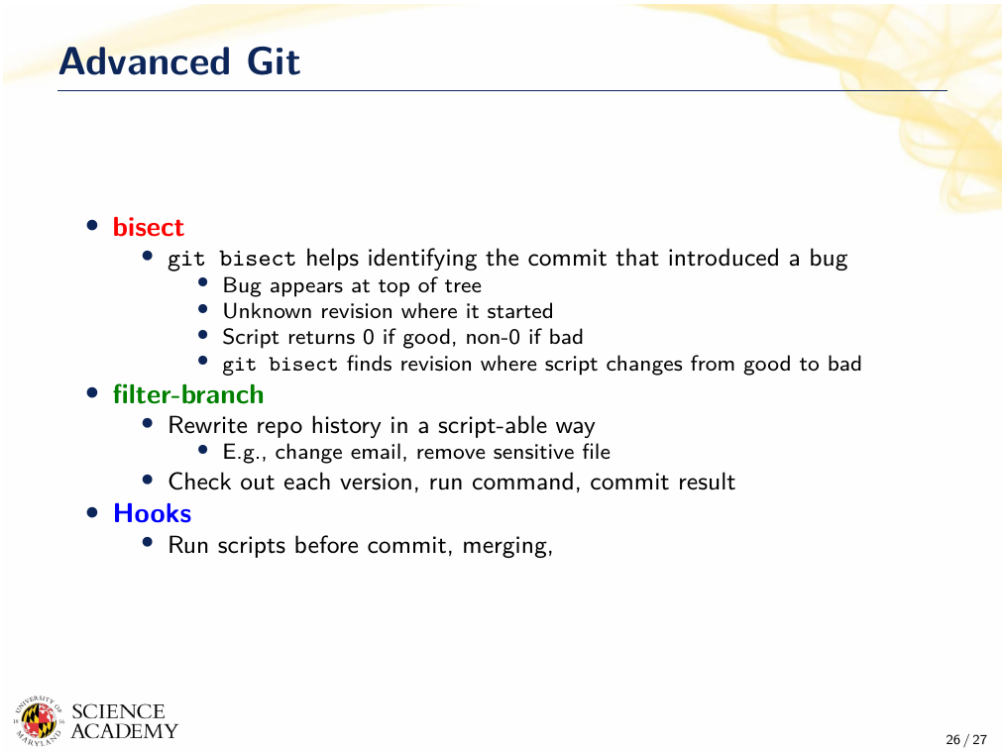
25 / 27: Advanced Git



- **Advanced Git**
- **Stashing**
  - *Stashing* is a handy feature in Git that allows you to temporarily save changes in your working directory. This includes both modified and staged files. Imagine you're in the middle of working on a feature, but suddenly need to switch to another branch to fix a bug. Instead of committing incomplete work, you can stash it. This saves your changes in a stack-like structure, allowing you to apply them later when you're ready to continue.
- **Cherry-picking**
  - *Cherry-picking* is a useful tool when you want to apply a specific commit from one branch to another. This is particularly helpful when you have a commit with a bug fix or a feature that you want to include in another branch without merging the entire branch. It allows for precise control over what changes are incorporated into your branch.
- **rerere**
  - The term *rerere* stands for "Reuse Recorded Resolution." This feature in Git helps you manage merge conflicts more efficiently. When you resolve a conflict, Git can remember how you resolved it. If the same conflict arises again, Git can automatically apply the previous resolution, saving you time and effort.
- **Submodules / subtrees**
  - These are methods for including one Git project within another. *Submodules* allow you to keep a Git repository as a subdirectory of another Git repository. This is useful

for managing dependencies. *Subtrees* offer a more integrated approach, allowing you to merge and split repositories more seamlessly. Both methods help manage complex projects that rely on multiple repositories.

26 / 27: Advanced Git

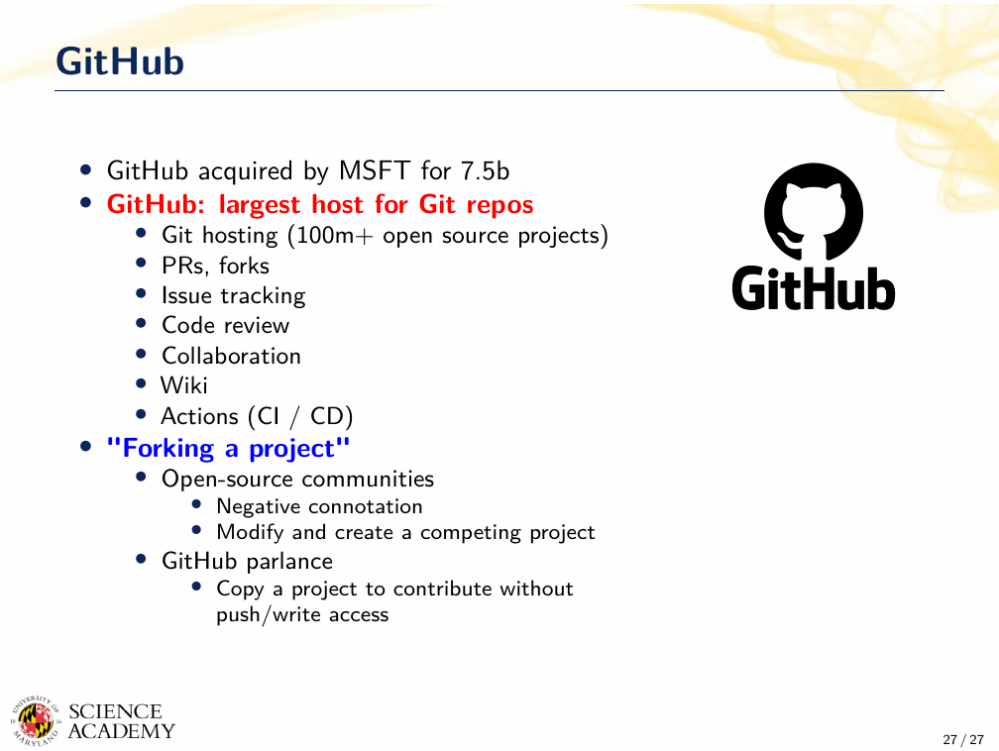


- **bisect**
  - *Purpose*: `git bisect` is a powerful tool used to pinpoint the exact commit that introduced a bug in your codebase.
    - Imagine you have a bug in your project, and you know it exists in the latest version, but you're not sure when it first appeared. `git bisect` helps you find that specific commit.
    - You start by marking the current version as "bad" (where the bug is present) and a previous version as "good" (where the bug was absent).
    - You can use a script that returns 0 if the code is good and a non-zero value if it's bad. This script automates the process of checking each commit.
    - `git bisect` will then automatically check out different commits and run your script to determine where the code changes from good to bad, effectively narrowing down the problematic commit.
- **filter-branch**
  - *Purpose*: This command allows you to rewrite the history of your repository in a programmable way.
    - It's useful for making changes across many commits, such as altering author information or removing sensitive data from the history.
    - The process involves checking out each commit, applying your changes (like running a command), and then committing the result back into the history.
    - This is a powerful tool but should be used with caution, as rewriting history can affect collaborators and shared repositories.
- **Hooks**
  - *Purpose*: Hooks are scripts that Git can run automatically at certain points in your

workflow.

* They can be set to run before actions like committing or merging, allowing you to enforce rules or automate tasks.
* For example, you might use a pre-commit hook to check code style or run tests before allowing a commit to proceed.
* Hooks help maintain code quality and consistency across a team by automating checks and processes.

27 / 27: GitHub



- **GitHub acquired by MSFT for 7.5b**
  - In 2018, Microsoft acquired GitHub for $7.5 billion. This acquisition was significant because GitHub is a major platform for developers, and Microsoft's purchase indicated their commitment to supporting open-source software and developer communities.
- **GitHub: largest host for Git repos**
  - GitHub is the largest platform for hosting Git repositories, with over 100 million open-source projects. It provides a space for developers to store and manage their code.
  - **Git hosting**: GitHub allows developers to host their code repositories, making it easier to manage and share code.
  - **PRs, forks**: Pull requests (PRs) and forks are essential features for collaboration, allowing developers to propose changes and work on copies of projects.
  - **Issue tracking**: GitHub offers tools to track bugs and feature requests, helping teams manage their projects efficiently.
  - **Code review**: Developers can review each other's code, ensuring quality and consistency.
  - **Collaboration**: GitHub fosters teamwork by providing tools for developers to work together on projects.
  - **Wiki**: Each project can have its own wiki, which is useful for documentation and sharing information.
  - **Actions (CI / CD)**: GitHub Actions enable continuous integration and continuous deployment (CI/CD), automating the testing and deployment of code.
- **"Forking a project"**
  - In open-source communities, *forking* can have a negative connotation, as it sometimes means creating a competing project by modifying an existing one.

– In GitHub parlance, forking is a positive action where a developer copies a project to contribute to it, especially when they don't have push or write access to the original repository. This allows for collaboration and improvement of the project.