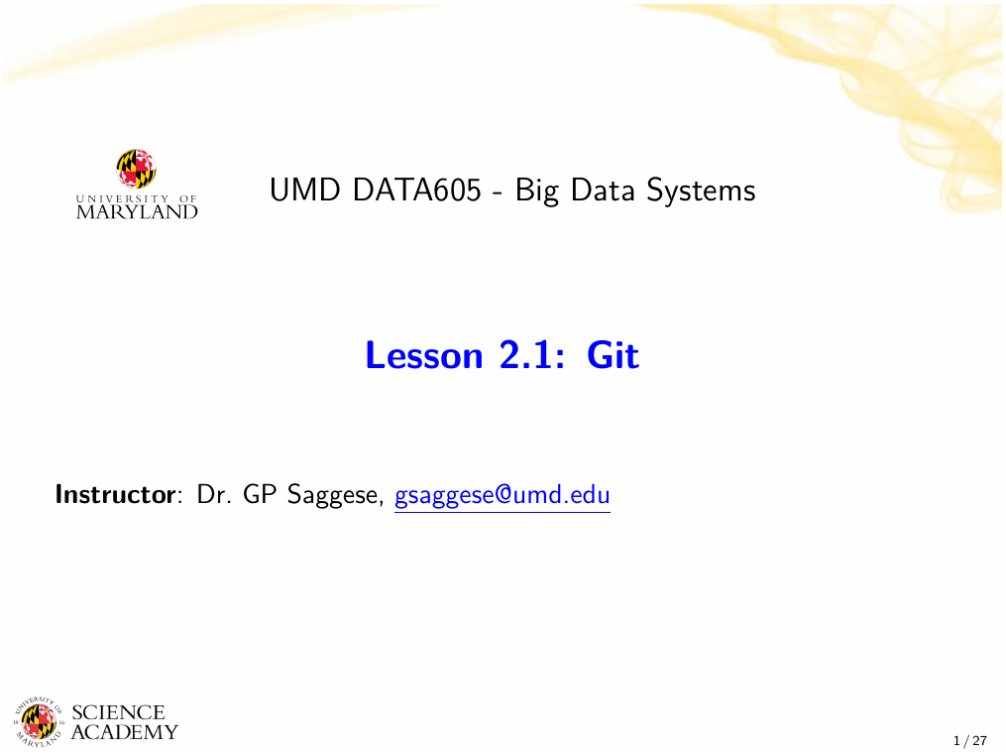
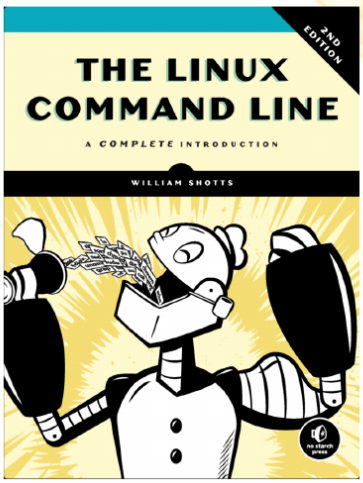
Lesson 2.1: Git



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

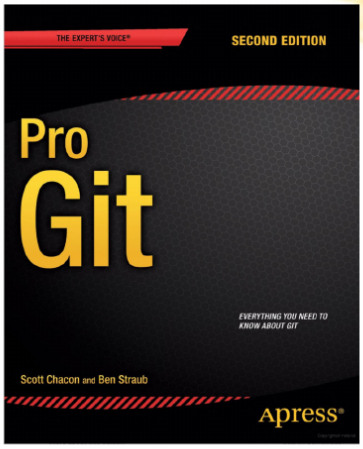
- **How Linux works**
  - *Processes*: In Linux, a process is an instance of a running program. Understanding processes is crucial because they are the basic units of execution in Linux. You can manage processes using commands like `ps`, `top`, and `kill`.
  - *File ownership and permissions*: Linux uses a permission system to control who can read, write, or execute a file. Each file has an owner and a group, and permissions are set for the owner, group, and others. Commands like `chmod` and `chown` are used to change these settings.
  - *Virtual memory*: This is a memory management capability that provides an "idealized abstraction of the storage resources" that are actually available on a given machine. It allows for more efficient and secure use of memory.
  - *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 unintentional system changes.
- **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 interface.
  - *E.g., `find`, `xargs`, `chmod`, `chown`, symbolic, and hard links*: These are fundamental commands and concepts in Linux. `find` is used to search for files, `xargs` is used to build and execute command lines from standard input, `chmod` and `chown` are used to change file permissions and ownership, and symbolic and hard links are methods of creating pointers to files.

- **Mastery**
  - *The Linux Command Line*: This resource is aimed at users who want to deepen their understanding of Linux. It covers more advanced topics and provides a comprehensive guide to mastering the command line, which is essential for efficient system administration and automation.

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

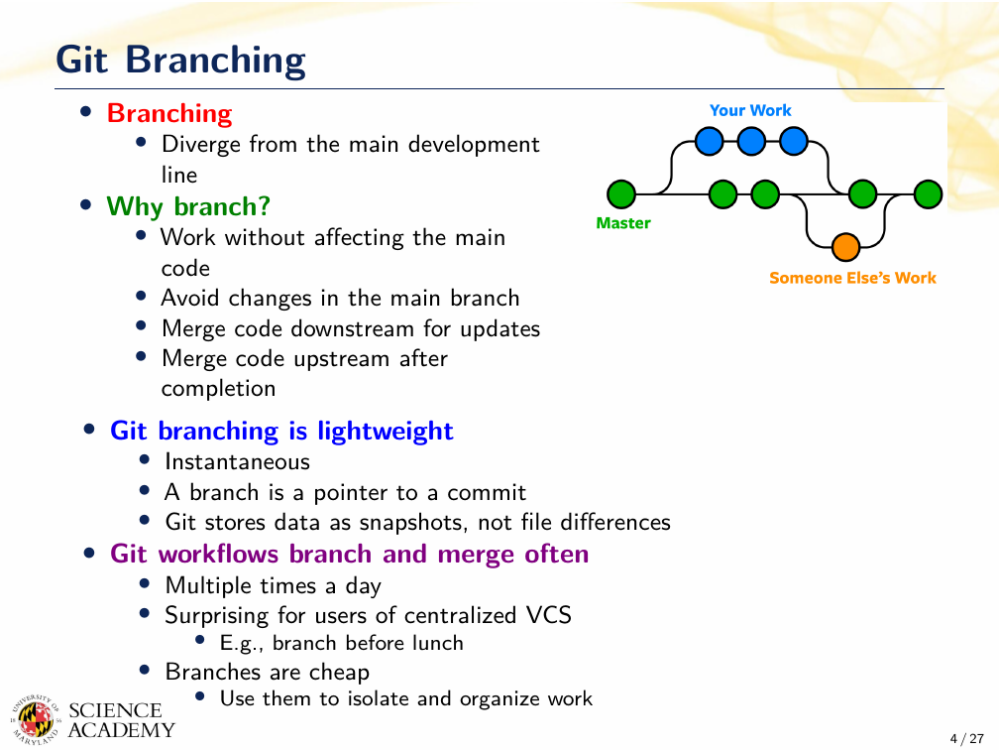
3 / 27

- **Concepts in the slides**: This bullet point suggests that the slide contains key concepts related to Git, a version control system. Understanding these concepts is crucial for managing code and collaborating on projects effectively.
- **Tutorial: Tutorial Git**: This link directs you to a Git tutorial, which is a practical guide to help you get started with Git. It's a hands-on resource that will walk you through the basics of using Git, making it easier to follow along and practice.
- **We will use Git during the project**: This point emphasizes the importance of Git in your coursework or project. It indicates that Git will be a tool you'll need to use, so gaining familiarity with it is essential for successful project completion.
- **Mastery: Pro Git (free)**: This is a link to a comprehensive book on Git, available for free. It's an excellent resource for those who want to deepen their understanding and become proficient in using Git.
- **Web resources**:
  - **https://githowto.com**: A website offering step-by-step Git tutorials, ideal for beginners.

  - **dangitgit.com (without swearing)**: A humorous resource that provides solutions to common Git problems without using offensive language.
  - **Oh Sh*t, Git!?! (with swearing)**: Similar to the previous resource but includes swearing, offering a light-hearted approach to solving Git issues.

- **Playgrounds**:
  - **https://learngitbranching.js.org**: An interactive platform where you can practice Git commands and concepts in a visual and engaging way. It's a great tool for experimenting with Git without affecting real projects.

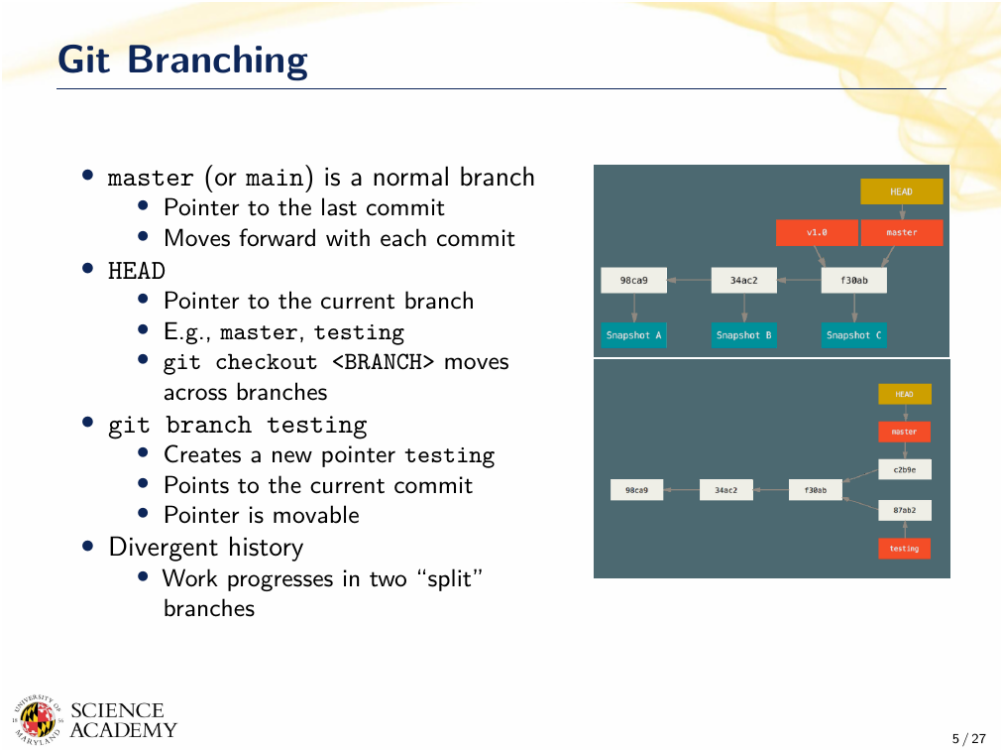
4 / 27: Git Branching



- **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 environments.
  - By keeping changes isolated, you avoid introducing errors or incomplete features into the main branch.
  - Once your work is ready, you can merge your changes downstream to update your branch with the latest code or upstream to integrate your completed work back into the main branch.
- **Git branching is lightweight**
  - Creating a branch in Git is quick and doesn't require much space. It's essentially just a

pointer to a specific commit in the project's history.
  - Unlike some other version control systems, Git stores data as snapshots of the entire project at a given time, rather than just the differences between files. This makes branching and switching between branches very efficient.
- **Git workflows branch and merge often**
  - In Git, it's common to create and merge branches multiple times a day. This might be surprising for those used to centralized version control systems, where branching can be more cumbersome.
  - Because branches are easy and inexpensive to create, they are used frequently to keep work organized and isolated. For example, you might create a branch for a new feature before lunch and merge it back into the main branch after testing it in the afternoon.

5 / 27: Git Branching

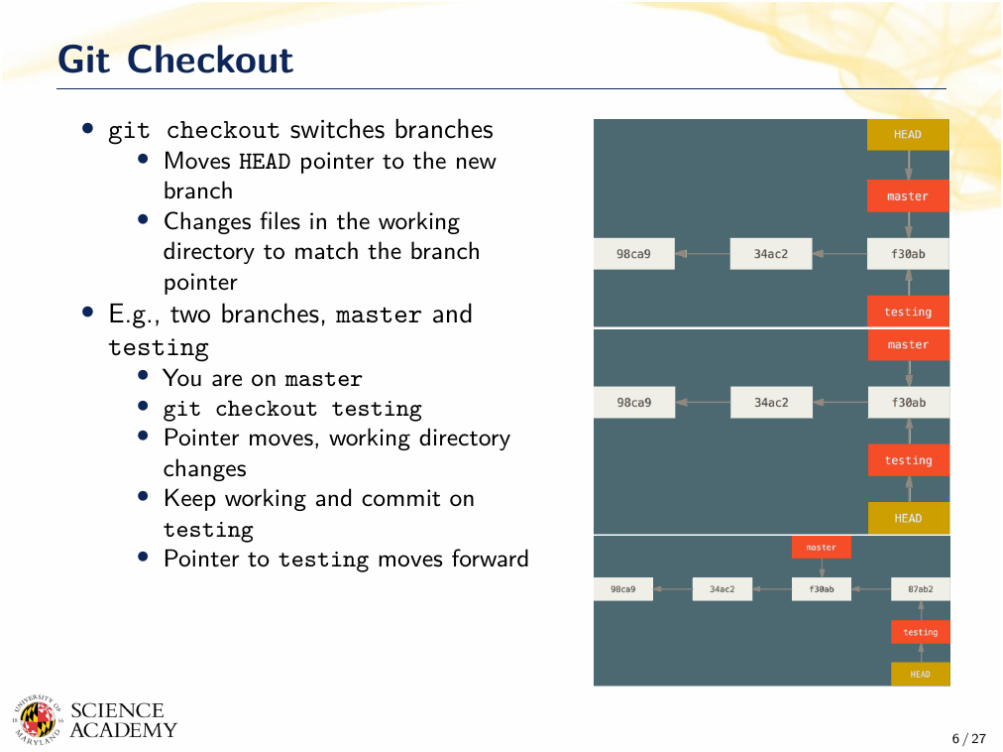


- **`master` (or `main`) is a normal branch**
  - In Git, the `master` or `main` branch is the default branch where the main development happens. It's essentially a *pointer* to the latest commit in the project. Every time you make a new commit, this pointer moves forward to include the new changes. This is why it's called a "branch"—it represents a line of development.
- **`HEAD`**
  - `HEAD` is a special pointer in Git that tells you which branch you are currently working on. For example, if you're working on the `master` branch, `HEAD` will point to `master`. When you switch branches using `git checkout <BRANCH>`, `HEAD` moves to point to the new branch, allowing you to work on different lines of development.
- **`git branch testing`**
  - This command creates a new branch called `testing`. It acts as a new pointer that starts

at the current commit you're on. This new branch is independent and can move forward as you make new commits. This is useful for experimenting or developing new features without affecting the main branch.

- **Divergent history**
  - When you have multiple branches, like `master` and `testing`, they can develop independently. This is known as a divergent history, where each branch can have its own set of commits. This allows for parallel development, where different features or fixes can be worked on simultaneously without interfering with each other.

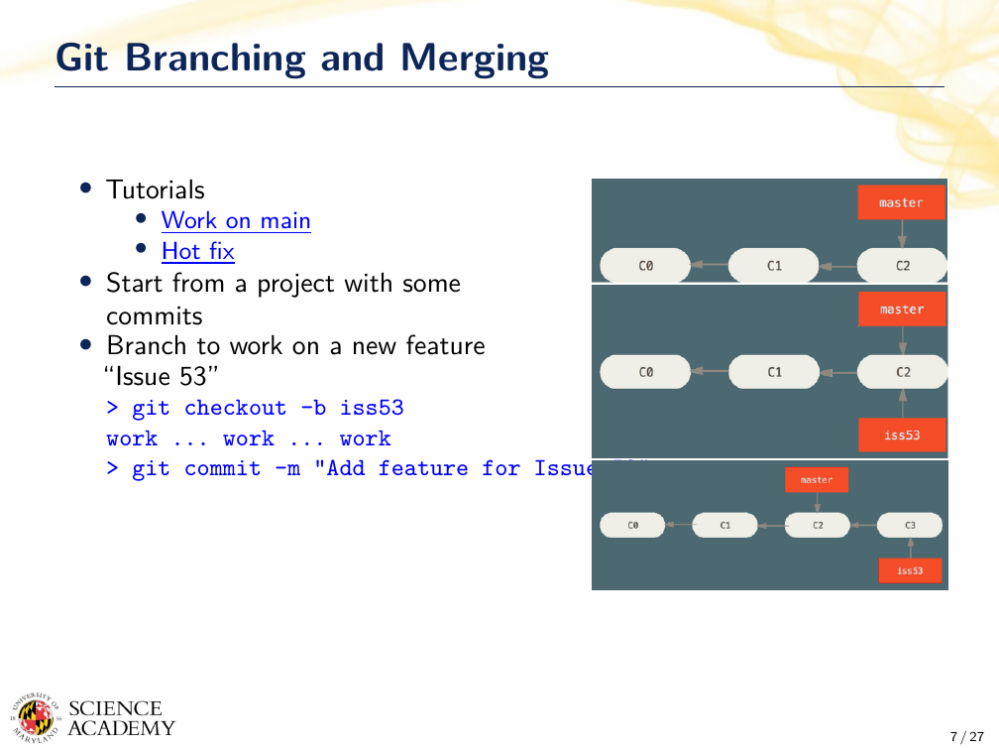
6 / 27: Git Checkout



- **`git checkout` switches branches**
  - The `git checkout` command is used to switch between different branches in a Git repository. This is a fundamental operation when working with Git, as it allows you to move between different lines of development.
  - When you use `git checkout`, it moves the *`HEAD` pointer* to the branch you want to switch to. The `HEAD` is a reference to the current branch you are working on.
  - It also updates 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 reflect the latest commit on that branch.
- **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* from `master` to `testing`, 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 you make will move the pointer of the `testing` branch forward, capturing your progress.
- **Images**
  - The images likely illustrate the process of switching branches and how the `HEAD` pointer and working directory change. They provide a visual representation of the concepts discussed, making it easier to understand how `git checkout` operates in practice.

7 / 27: Git Branching and Merging



- **Git Branching and Merging**: This slide is about using Git, a version control system, to manage changes in a project. It focuses on branching and merging, which are key concepts in Git that help in organizing and integrating different lines of work.
- **Tutorials**: The slide provides links to tutorials that explain how to work on the main branch and how to apply a hot fix. These tutorials are useful for understanding basic Git operations and handling urgent bug fixes without disrupting the main workflow.
- **Start from a project with some commits**: Before branching, it's important to have a project with existing commits. This ensures that you have a stable base to work from and that your changes are built on top of a solid foundation.
- **Branch to work on a new feature "Issue 53"**: The slide demonstrates how to create a new branch for a specific feature or issue. Using the command `git checkout -b iss53`, you create a branch named "iss53" to work on a new feature. This allows you to make changes without affecting the main codebase. After completing the work, you commit the changes with a message describing the feature added.

- **Images**: The images likely illustrate the branching and merging process visually, showing how branches diverge from the main line of development and how they are eventually merged back. This visual representation helps in understanding the flow of changes and how different branches relate to each other.

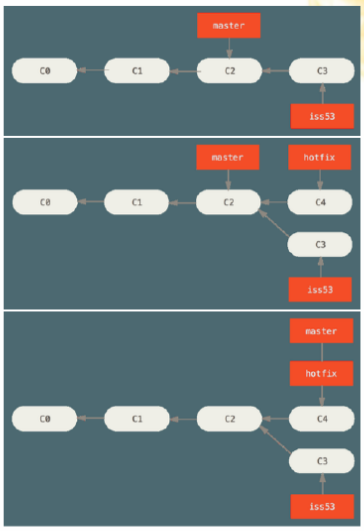
8 / 27: Git Branching and Merging

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 no divergent history between `master` and `iss53`



SCIENCE ACADEMY

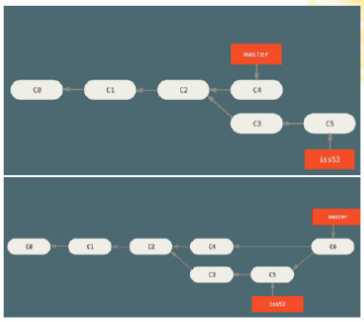
8 / 27

- **Need a hotfix** to `master`
  - This section explains how to apply a quick fix, known as a *hotfix*, to the `master` branch in Git.
  - The process begins by switching to the `master` branch using `git checkout master`.
  - A new branch named `hotfix` is created with `git checkout -b hotfix`. This allows you to make changes without affecting the `master` branch directly.
  - After making the necessary fixes, you commit the changes with `git commit -am "Hot fix"`. The `-am` flag is used to add and commit changes with a message in one step.
  - Finally, you merge the `hotfix` branch back into `master` using `git merge hotfix`, integrating the changes.
- **Fast forward**
  - This concept refers to a type of merge in Git where the `master` branch is updated to match another branch, like `iss53`, without creating a new commit.
  - A fast forward merge occurs when there is no divergent history between the branches, meaning one branch is directly ahead of the other.
  - This is a simple and efficient way to update branches when no conflicts exist, keeping the commit history clean and linear.

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`. This is where you continue to make changes and improvements related to the issue or feature you're working on. The phrase "work … work … work" implies that you are actively developing or fixing something on this branch.
  - **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 that the two branches are becoming different from each other.
- **At some point you are done with `iss53`**
  - Once you have completed your work on `iss53`, you will want to integrate these changes back into the main branch, typically `master`.
  - **Go to the target branch**: You switch to the `master` branch using `git checkout master` and then merge the changes from `iss53` into `master` with `git merge iss53`.
- **Git can't fast forward**
  - Sometimes, Git cannot simply move the `master` branch pointer forward to include the changes from `iss53` because the branches have diverged. In such cases, Git performs a 3-way merge.
- **Git creates a new snapshot with the 3-way "merge commit"**
  - A 3-way merge involves creating a new commit that has more than one parent, effectively combining the histories of both branches. This new commit is called a "merge commit."
- **Delete the branch**
  - After successfully merging `iss53` into `master`, you can delete the `iss53` branch using `git branch -d iss53`. This is a way to clean up your repository by removing branches that are no longer needed.

10 / 27: Fast Forward Merge

Fast Forward Merge

- Merge a commit `Y` with a commit `X` that can be reached by following the history of commit `Y`
- There is no divergent history to merge
  - Git simply moves the branch pointer forward from X to Y
- **Mental model**: a branch is just a pointer that indicates 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

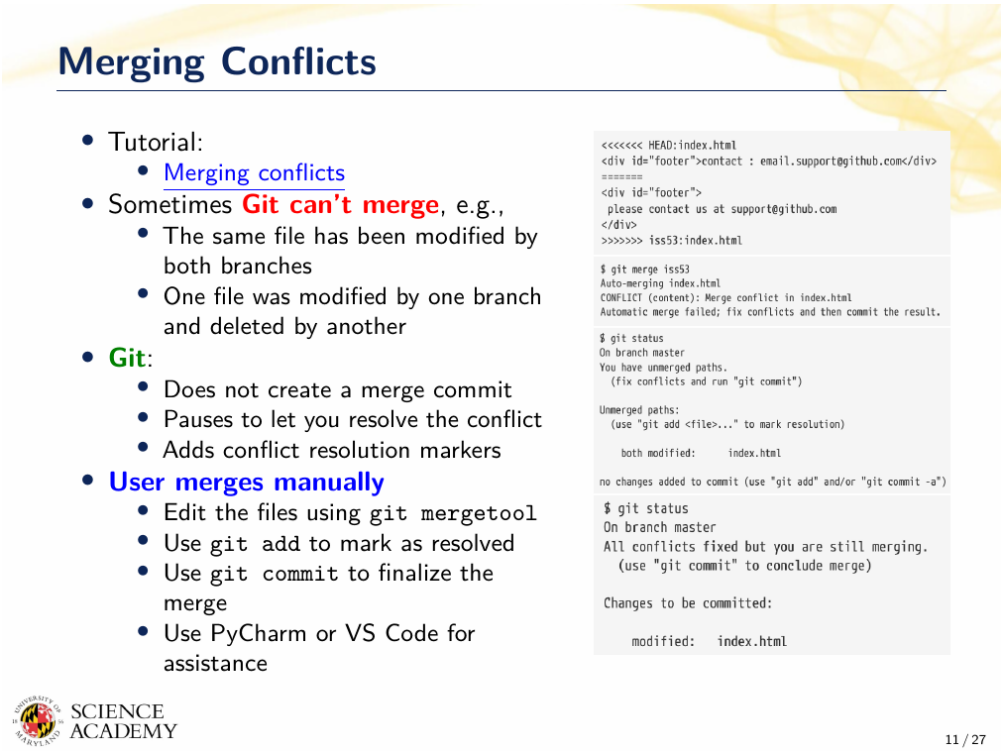
10 / 27

- **Fast Forward Merge**: This is a type of merge in Git where you combine two commits, `Y` and `X`, but `X` is already part of the history of `Y`. This means that `Y` is just an extension of `X` without any separate paths or branches that need to be reconciled.
- **No Divergent History**: In this scenario, there are no conflicting changes or separate lines of development. Git can simply update the branch pointer to the latest commit, `Y`, without creating a new merge commit. This is efficient and keeps the commit history clean.
- **Mental Model**: Think of a branch as a simple pointer that marks the latest commit in that branch. When you perform a fast forward merge, you're essentially just moving this pointer forward to include the new commits.
- **Example**: If you have a branch `master` and another branch `experiment` with a new commit `C4'`, and `C4'` is directly reachable from `C3`, you can merge `experiment` into `master` using:

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

  This command moves the `master` branch pointer to `C4'`, effectively incorporating the changes without creating a new merge commit.
- **Visual Representation**: The images likely illustrate how the branch pointer moves from `C3` to `C4'`, showing the simplicity and efficiency of a fast forward merge. This visual aid helps in understanding how Git handles such merges seamlessly.

11 / 27: Merging Conflicts

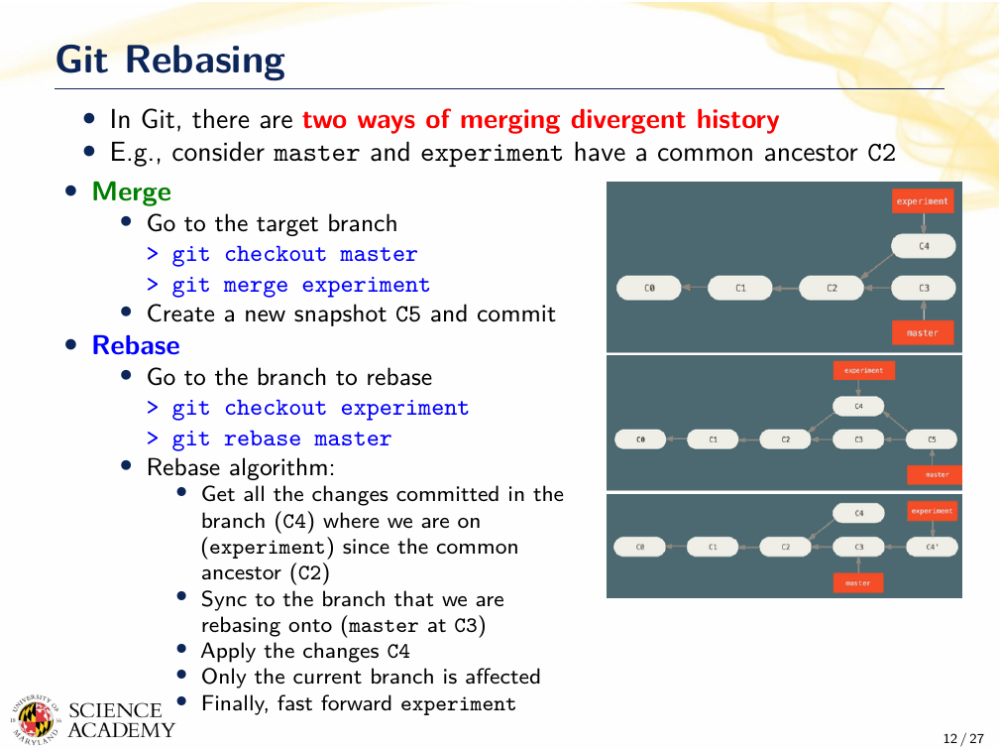


- **Merging Conflicts**: This slide is about handling situations where Git, a version control system, encounters conflicts during the merging process. Merging is when you combine changes from different branches of a project. Sometimes, Git can't automatically merge changes, leading to conflicts.
- **Tutorial**: A link is provided to a tutorial on merging conflicts. This is a helpful resource for understanding how to handle these situations step-by-step.
- **When Git Can't Merge**: Conflicts occur when:
  - Both branches have modified the same file. This means changes have been made to the same lines or sections of code, and Git doesn't know which changes to keep.
  - One branch has modified a file while another branch has deleted it. Git can't decide whether to keep the modifications or accept the deletion.
- **Git's Behavior During Conflicts**:
  - Git won't automatically create a merge commit when it detects conflicts. Instead, it pauses the process.
  - It adds conflict resolution markers in the files, which are special lines that show where the conflicts are.
- **User's Role in Resolving Conflicts**:
  - You need to manually edit the files to resolve conflicts. Tools like `git mergetool` can help with this.
  - Once you've resolved the conflicts, use `git add` to mark the files as resolved.
  - Finally, use `git commit` to complete the merge process.

- Integrated development environments (IDEs) like PyCharm or VS Code can provide additional support and make the process easier by offering visual tools to resolve conflicts.

This slide emphasizes the importance of understanding how to handle merge conflicts, as they are a common part of collaborative software development.

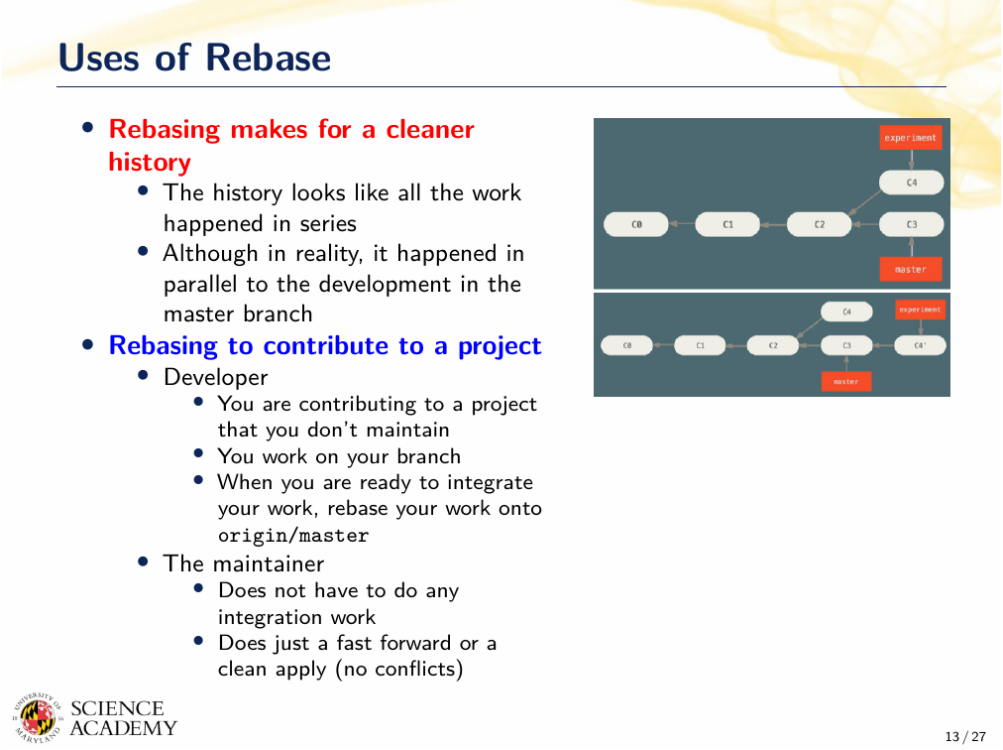
12 / 27: Git Rebasing



- **Two Ways of Merging Divergent History in Git**
  - In Git, when you have two branches that have diverged from a common point, there are two main strategies to bring them back together: *merge* and *rebase*. Both methods aim to integrate changes from one branch into another, but they do so in different ways.
- **Merge**
  - To merge, you first switch to the target branch where you want to integrate changes. For example, if you want to merge changes from `experiment` into `master`, you would first check out `master`.
  - The command `git merge experiment` combines the histories of the two branches. This creates a new commit, often called a "merge commit" (e.g., `C5`), which has two parent commits, preserving the history of both branches.
- **Rebase**
  - Rebasing involves changing the base of your branch to a different commit. You start by checking out the branch you want to rebase, such as `experiment`.
  - The rebase process involves taking all the changes made in `experiment` since the common ancestor (`C2`) and applying them on top of the `master` branch at its current state (`C3`).
  - This effectively rewrites the history of `experiment`, making it appear as if the changes were made after `master`'s latest commit. Only the `experiment` branch is altered, and it results in a cleaner, linear history.

– After rebasing, the `experiment` branch is fast-forwarded to reflect these changes.

13 / 27: Uses of Rebase

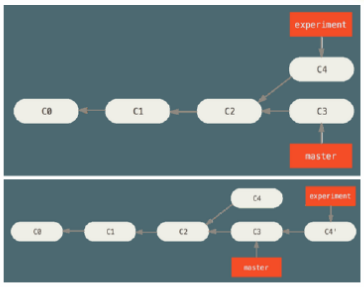


- **Rebasing makes for a cleaner history**
  - When you rebase, it rearranges the commit history to make it look like all the changes were made one after the other, in a straight line. This is called a linear history.
  - In reality, multiple developers might have been working on different features or fixes at the same time, but rebasing makes it look like these changes were made sequentially. This can make the history easier to read and understand.
- **Rebasing to contribute to a project**
  - *Developer's role*: If you're contributing to a project that someone else maintains, 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 `origin/master` branch. This ensures your changes are up-to-date with the main project.
  - *Maintainer's role*: The project maintainer benefits because they don't have to manually integrate your changes. They can simply fast forward or apply your changes cleanly, without dealing with conflicts. This makes the process smoother and less error-prone.

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

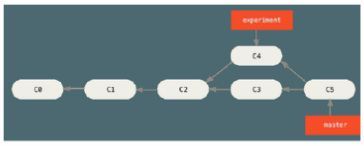
- **Golden Rule of Rebasing**
  - **Remember**: When you rebase, you are essentially taking your existing commits and creating new versions of them. These new commits are similar to the old ones but are technically different. This means that the history of your project changes, which can have significant implications when working with others.
- **Problem**
  - Imagine you have pushed your commits to a remote repository, and your teammates have pulled these commits to work on them.
  - If you decide to rewrite your commit history using `git rebase`, you are altering the commit history that your teammates are also using.
  - When you push these changes back to the remote repository using `git push --force`, it can create confusion and extra work for your collaborators, as they now have to re-merge their work with the new commit history.
- **Solution**
  - A strict approach is to *never rebase commits that have been shared outside your personal repository*. This avoids any potential conflicts or confusion for your collaborators.
  - A more flexible approach allows for rebasing if you are the only one working on a branch, even if it has been pushed to a server. This is because no one else is relying on that commit history, so there is no risk of disrupting others' work.
- **Visual Aids**
  - The images likely illustrate the process of rebasing and the potential issues that can arise when it is not done carefully. They serve as a visual reminder of the importance of following the golden rule of rebasing.

15 / 27: Rebase vs Merge: Philosophical Considerations

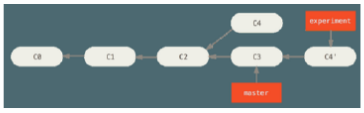
Rebase vs Merge: Philosophical Considerations

- Deciding **Rebase-vs-merge** depends on the answer to the question:
  - *What does the commit history of a repo mean?*

1. **History is the record of what actually happened**
   - *"History should not be tampered with, even if messy!"*
   - Use `git merge`



2. **History represents how a project should have been made**
   - *"You should tell the history in the way that is best for future readers"*
   - Use `git rebase` and `filter-branch`



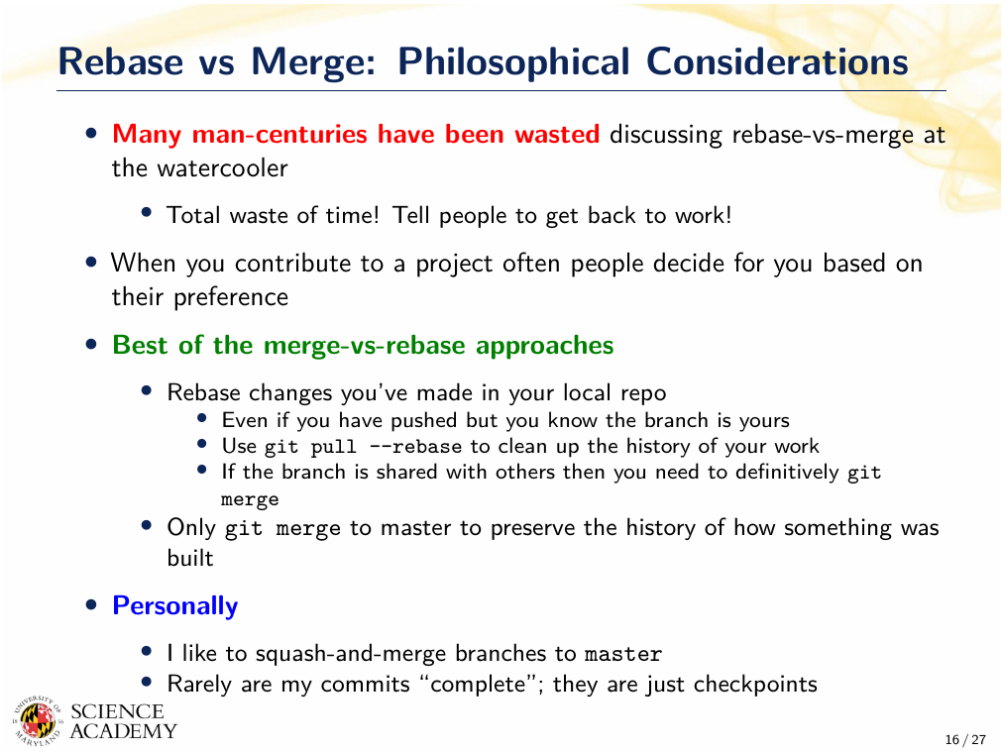
SCIENCE ACADEMY

15 / 27

- **Rebase vs Merge: Philosophical Considerations**
  - The choice between **Rebase** and **Merge** is fundamentally about how you view the commit history of a repository. This decision is not just technical but also philosophical, as it reflects your perspective on the importance and purpose of commit history.
- **History is the record of what actually happened**
  - This viewpoint suggests that the commit history should be an accurate reflection of all the changes that have occurred, even if it appears messy or complex. The idea is that history should remain untouched to preserve the authenticity of the development process.
  - In this case, using `git merge` is recommended. Merging keeps all the commits intact, showing exactly how the project evolved over time, including all branches and merges.
- **History represents how a project should have been made**
  - From this perspective, the commit history is seen as a narrative that should be crafted to be as clear and understandable as possible for future readers. This means organizing and simplifying the history to make it more logical and easier to follow.
  - Here, using `git rebase` and `filter-branch` is suggested. Rebasing allows you to rewrite commit history, making it linear and cleaner, which can help in understanding the progression of the project without the clutter of all the branching and merging details.

In summary, the choice between rebase and merge depends on whether you value an accurate historical record or a clean, understandable narrative for future developers.

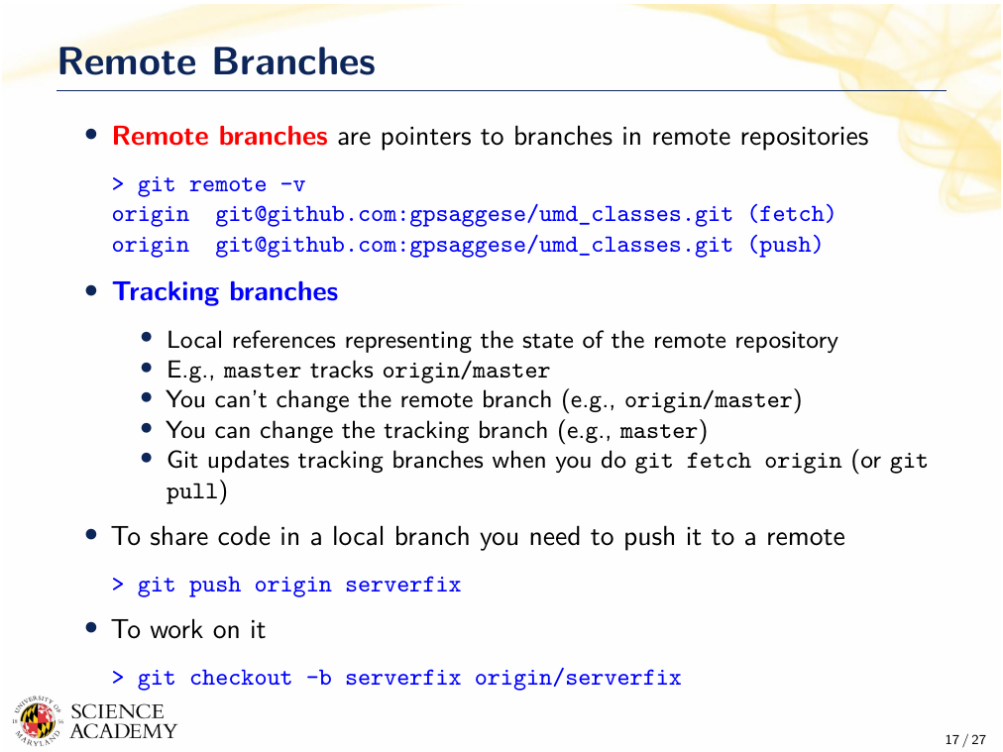
16 / 27: Rebase vs Merge: Philosophical Considerations



- **Many man-centuries have been wasted** discussing rebase-vs-merge at the watercooler
  - This point humorously highlights how much time developers spend debating whether to use rebase or merge in version control systems like Git. The suggestion to "get back to work" implies that these discussions can be unproductive and that the choice between rebase and merge might not be as critical as some make it out to be.
- When you contribute to a project often people decide for you based on their preference
  - In many projects, the decision to use rebase or merge is often made by the project maintainers or team leads. This means that individual contributors might not have a say in the matter and should follow the established workflow to maintain consistency.
- **Best of the merge-vs-rebase approaches**
  - *Rebase changes you've made in your local repo*: Rebasing is useful for keeping your local changes up-to-date with the main branch. It helps in cleaning up the commit history, making it linear and easier to understand.
    - Even if you have pushed but you know the branch is yours: If you are the only one working on a branch, rebasing after pushing is generally safe.
    - Use `git pull --rebase` to clean up the history of your work: This command helps in integrating changes from the main branch into your branch without creating unnecessary merge commits.
    - If the branch is shared with others then you need to definitively `git merge`: When working on a shared branch, merging is safer to avoid disrupting others' work.
  - Only `git merge` to master to preserve the history of how something was built: Merging into the main branch (often called master) keeps a record of how features were developed and integrated, which can be important for understanding the project's history.
- **Personally**

- I like to squash-and-merge branches to `master`: Squashing combines all commits from a branch into a single commit before merging, which simplifies the commit history.
- Rarely are my commits "complete"; they are just checkpoints: This suggests that the speaker uses commits as a way to save progress rather than as final, polished changes. Squashing helps in presenting a cleaner history when merging into the main branch.

17 / 27: Remote Branches



- **Remote branches** are essentially bookmarks that point to the state of branches in a remote repository. They help you keep track of what others are working on and the latest updates from the remote repository. When you run the command `git remote -v`, it shows you the URLs of the remote repositories you have configured, both for fetching and pushing changes. This is crucial for collaboration, as it ensures everyone is working with the most current version of the code.
- **Tracking branches** are local branches that have a direct relationship with a remote branch. For example, your local `master` branch might track `origin/master`, meaning it reflects the state of the `master` branch on the remote named `origin`. You can't directly modify a remote branch like `origin/master`; instead, you make changes to your local tracking branch and then push those changes to the remote. When you run `git fetch origin` or `git pull`, Git updates your tracking branches to reflect the latest changes from the remote repository.
- To share your work with others, you need to push your local branch to the remote repository. For instance, if you have a local branch named `serverfix`, you would use the command `git push origin serverfix` to upload it to the remote repository. This makes your changes available to others who have access to the repository.
- If you want to start working on a branch that exists on the remote but not locally, you can

create a new local branch that tracks the remote branch. The command `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 levels of stability 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

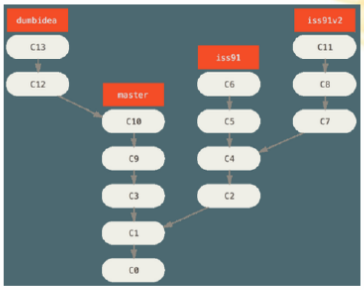
18 / 27

- **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, tested, and deployed.
- **Long-running branches**: These are branches in a Git repository that are continuously maintained and updated. 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 is where new features and fixes are integrated before they are considered stable enough to be merged into the `master` branch.
  - **Topic/feature branches**: These are temporary branches created to work on specific features or fixes. Once the work is complete and stable, these branches are merged into the `develop` branch.
  - The process of merging branches ensures that only stable and tested code is integrated into the main branches, maintaining the integrity of the project.

19 / 27: Git Workflows

Git Workflows

- **Topic branches** = short-lived branches for a single feature
  - E.g., `hotfix`, `wip-XYZ`
  - Easy to review
  - Siloed 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 a `dumbidea` branch and merges it to `master` (!)
    - You squash-and-merge your `iss91v2`



SCIENCE ACADEMY

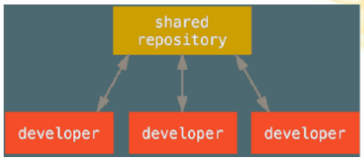
19 / 27

- **Topic branches** are short-lived branches created for working on a single feature or fix. They are temporary and focused, allowing developers to work on specific tasks without affecting the main codebase.
  - Examples of topic branches include `hotfix` for urgent bug fixes and `wip-XYZ` for work-in-progress features.
  - These branches are *easy to review* because they contain changes related to a single feature or fix, making it simpler for team members to understand and provide feedback.
  - Topic branches are *siloed from the rest* of the codebase, meaning they are isolated and do not interfere with the main or other branches until changes are ready to be integrated.
  - Git excels at handling branches, unlike some other version control systems (VCS) where branch support might not be as robust.
  - For example, you might start working on a branch named `iss91`, realize some changes need to be canceled, and then continue on a new branch `iss91v2`.
  - Sometimes, someone might create a branch called `dumbidea` and mistakenly merge it into the `master` branch, which can lead to issues if not reviewed properly.
  - Once your work on `iss91v2` is complete, you can use a *squash-and-merge* strategy to combine all changes into a single commit before merging it into the main branch, keeping the history clean and concise.

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 the latest copy, otherwise they need to merge)



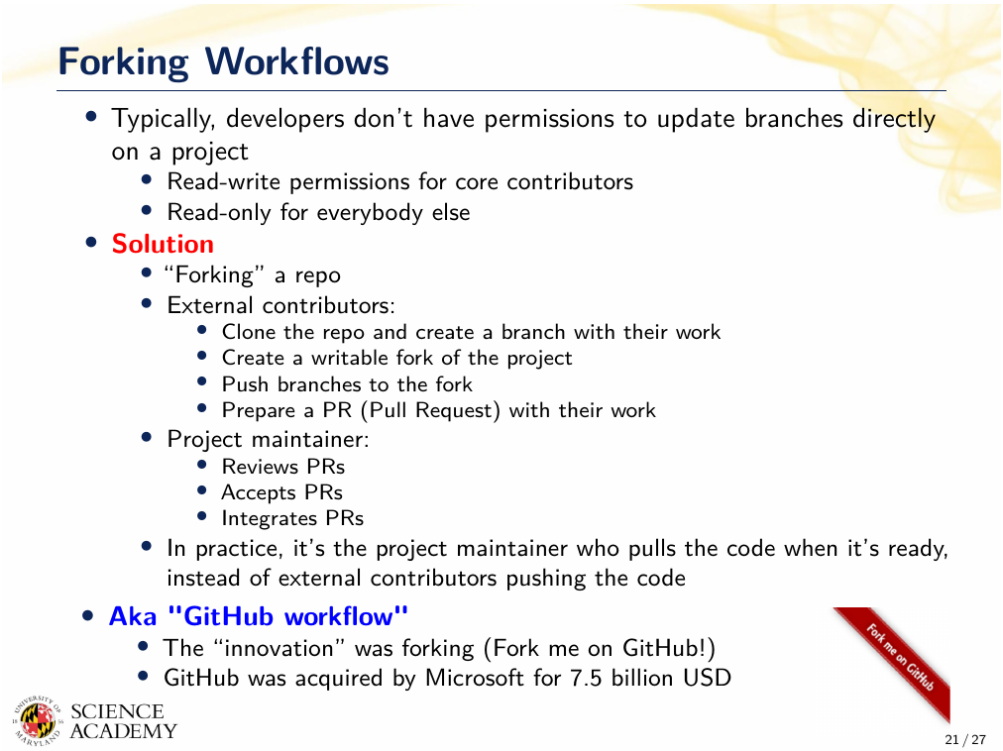
- **Centralized workflow in Git**
  - Developers:
    - Have push (i.e., write) access to the central repo
    - Need to fetch and then merge
    - Cannot push code that will overwrite each other's code (only fast-forward changes)

SCIENCE ACADEMY

20 / 27

- **Centralized workflow in centralized VCS**
  - *Developers*:
    - **Check out the code from the central repo on their computer**: This means that developers download the latest version of the code from a central repository to their local machine. This is the starting point for any changes they want to make.
    - **Modify the code locally**: Developers work on their own computers to make changes to the code. This allows them to test and develop features independently.
    - **Push it back to the central hub**: Once changes are made, developers upload their modified code back to the central repository. If someone else has changed the code in the meantime, developers may need to merge those changes with their own before pushing.
- **Centralized workflow in Git**
  - *Developers*:
    - **Have push (i.e., write) access to the central repo**: In Git, developers can directly update the central repository, but they need permission to do so.
    - **Need to fetch and then merge**: Before pushing changes, developers must fetch the latest updates from the central repository and merge them with their local changes. This ensures that their work is up-to-date with the central version.
    - **Cannot push code that will overwrite each other's code**: Git prevents developers from overwriting each other's work. They can only push changes that are compatible with the current state of the central repository, often requiring a fast-forward merge. This helps maintain the integrity of the codebase.

21 / 27: Forking Workflows



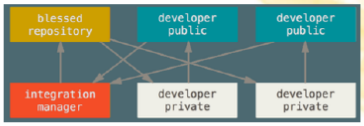
- **Forking Workflows**
  - In many open-source projects, not all developers have the ability to make changes directly to the project's codebase. This is because only core contributors, who are trusted members of the project, have read-write permissions. Everyone else typically has read-only access, meaning they can view the code but cannot make changes directly.
- **Solution**
  - The concept of "forking" a repository is a common solution to this problem. When developers want to contribute to a project but don't have direct access, they can create a "fork," which is essentially a personal copy of the repository.
  - **External contributors**:
    - They start by cloning the repository, which means they download a copy of the code to their local machine.
    - They then create a branch in their fork where they can make changes and add new features.
    - Once their work is complete, they push these changes to their forked repository.
    - Finally, they submit a Pull Request (PR) to the original project, proposing that their changes be merged into the main codebase.
  - **Project maintainer**:
    - The maintainer of the project reviews the submitted PRs to ensure the changes are beneficial and do not introduce issues.
    - If the PR is satisfactory, the maintainer accepts it and integrates the changes into the main project.
    - This process ensures that only vetted and approved code is added to the project, maintaining its quality and integrity.

- **Aka "GitHub workflow"**
  - This workflow is often referred to as the "GitHub workflow" because GitHub popularized the concept of forking repositories. The phrase "Fork me on GitHub!" became synonymous with encouraging contributions to open-source projects.
  - GitHub's success and influence in the software development community were significant factors in its acquisition by Microsoft for a substantial sum of 7.5 billion USD, highlighting 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 their own public copy
   - Sends a pull request to the maintainer asking to merge changes
3. **The maintainer**
   - Adds the contributor's repo as a remote
   - Merges the changes into a local branch
   - Tests changes locally
   - Pushes the 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 a structured way to manage contributions from multiple developers, ensuring that the main project remains stable and organized. This model is commonly seen in projects like Linux and on platforms like GitHub, where the concept of "forking" is prevalent.
- **One repo is the official project**
  In this workflow, there is a single repository that is considered the "official" version of 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`, this would be the main repository everyone refers to.
- **Each contributor**
  Contributors have the ability to read from all public repositories, which means they can see what others are working on. They create a "fork" of the project, which is essentially a personal copy where they have write access. For instance, a contributor might fork the project to `gpsaggese/csfy`. They make their changes in this fork and then push these changes to

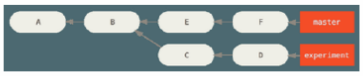
their own public repository. Once satisfied, they send a "pull request" to the maintainer, requesting that their changes be merged into the official project.

- **The maintainer**
  The maintainer plays a crucial role in this workflow. They add the contributor's repository as a "remote," which allows them to pull in the changes. They then merge these changes into a local branch on their machine, where they can test and verify that everything works as expected. Once the changes are confirmed to be stable and beneficial, the maintainer pushes the updated branch to the official repository, thus integrating the contributor's work into the main project.

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 the second parent of a merge commit
  - A merge commit has multiple parents



SCIENCE ACADEMY

23 / 27

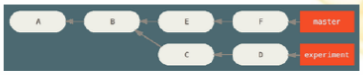
- **Git log**
  - The `git log` command is a powerful tool in Git that provides detailed information about the history of commits in a repository. It allows you to see what changes have been made, who made them, and when they were made. This is crucial for tracking the evolution of a project and understanding the context of changes.
- **Refs (References)**
  - *Refs* are pointers to specific commits or branches in your Git repository. They help you navigate through the history of your project.
  - **HEAD**: This is 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, typically the main branch on the remote repository. It helps you keep track of changes that have been pushed to the central repository.
  - **experiment**: This is an example of a local branch. Local branches are used to develop

features or fixes independently before merging them into the main branch.
  - **d921970**: This is an example of a specific commit identifier (SHA-1 hash). Each commit in Git has a unique identifier that allows you to reference it directly.
- **Commit Parent References**
  - The ^ symbol is used to navigate through the commit history by referring to parent commits.
  - **HEAD^**: This refers to the commit immediately before the current `HEAD`. It's useful for looking at the previous state of the project.
  - **^2**: This is used in the context of merge commits, which have more than one parent. It refers to the second parent of a merge commit, allowing you to see the other branch that was merged.
  - Understanding parent commits is essential for resolving conflicts and understanding how different branches have been integrated.

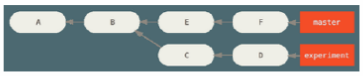
24 / 27: Dot notation

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`



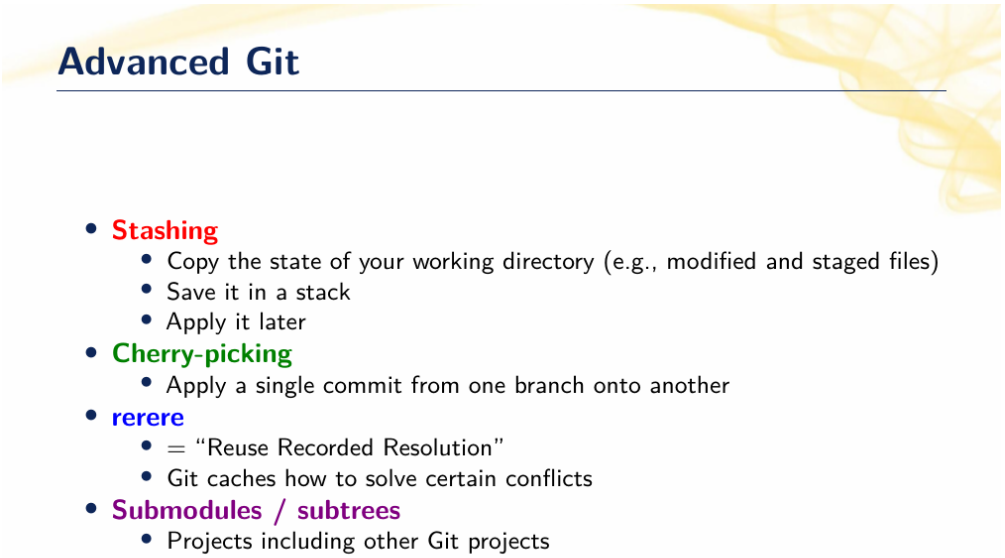
SCIENCE ACADEMY

24 / 27

- **Double-dot notation**
  - This concept is used in Git to compare two branches or commits. When you see `1..2`, it means you are looking at all the commits that can be reached from commit 2 but not from commit 1. Think of it as finding the "difference" between two points in your project's history.
  - 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 list commits `D` and `C`.
  - Conversely, `git log experiment..master` will show you the commits that are in `master` but not in `experiment`, which would be `F` and `E`.

- **Triple-dot notation**
  - This notation is slightly different. When you see `1...2`, it refers to all the commits that are reachable from either of the two branches but not from both. It's like taking a "union" of the two sets of commits and then excluding the ones that are common to both.
  - For instance, `git log master...experiment` will show you all the commits that are unique to either `master` or `experiment`, which would be `F`, `E`, `D`, and `C`. This helps you see what changes are unique to each branch without the overlap.

25 / 27: Advanced Git

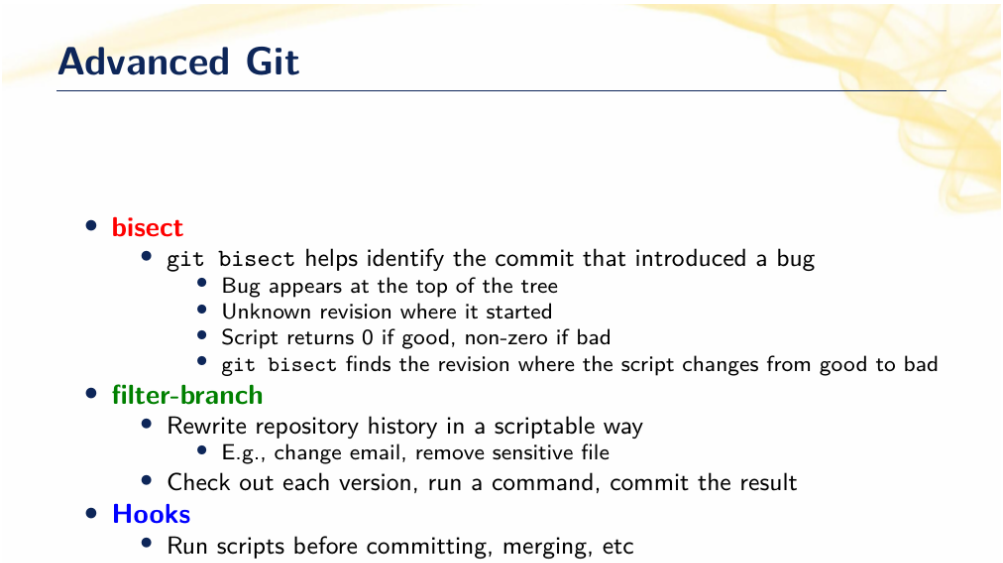


- **Stashing**
  - *Stashing* is a useful feature in Git that allows you to temporarily save changes in your working directory. This is particularly helpful when you need to switch branches but aren't ready to commit your current changes. By stashing, you create a snapshot of your modified and staged files and store it in a stack. You can then apply these changes later when you're ready to continue working on them. This helps maintain a clean working directory and prevents incomplete changes from being committed.
- **Cherry-picking**
  - *Cherry-picking* is a powerful tool in Git that lets you apply a specific commit from one branch to another. This is useful when you want to incorporate a particular change without merging entire branches. For example, if a bug fix is made in a feature branch, you can cherry-pick that commit to include it in the main branch without merging all other changes from the feature branch.
- **rerere**
  - The term *rerere* stands for "Reuse Recorded Resolution." This feature in Git helps auto-

mate the resolution of merge conflicts. When you resolve a conflict, Git can remember how you resolved it. If the same conflict occurs again, Git can automatically apply the previously recorded resolution, saving time and reducing repetitive work.

- **Submodules / subtrees**
  - *Submodules* and *subtrees* are methods for including one Git repository within another. This is useful for managing dependencies or incorporating external projects. Submodules link to a specific commit of another repository, while subtrees allow you to merge the entire history of the included project. Both methods help manage complex projects with multiple components, ensuring that each part can be developed and maintained independently.

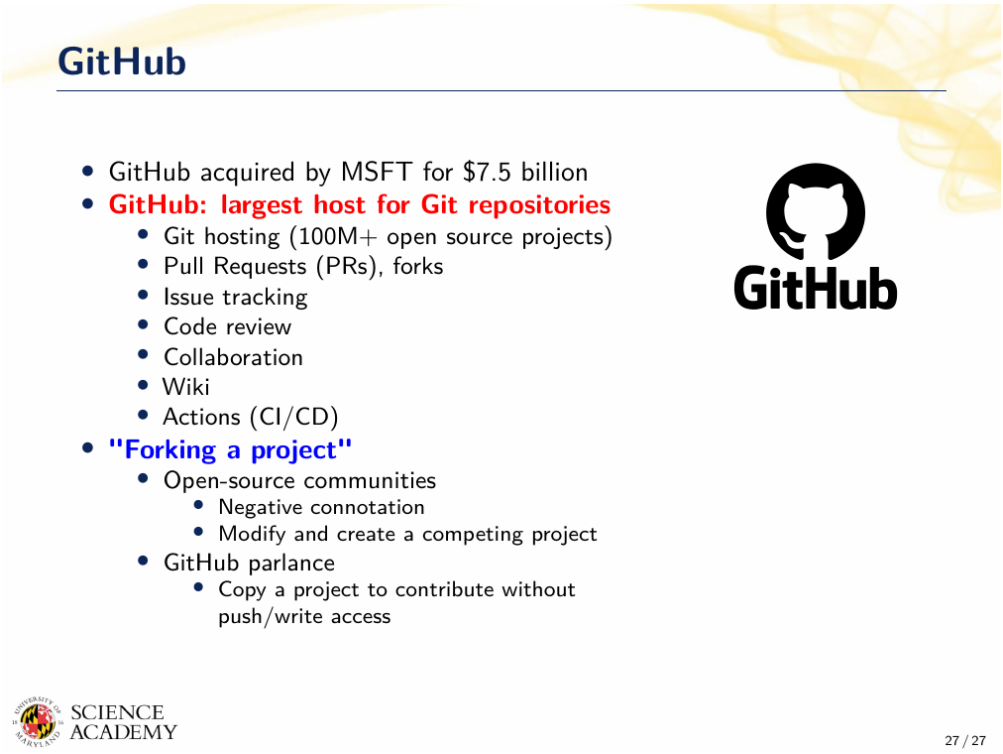
26 / 27: Advanced Git



- **bisect**
  - *What it does*: `git bisect` is a powerful tool that helps you find the specific commit where a bug was introduced in your code.
  - *How it works*: Imagine your code is like a tree, and the bug is at the top. You know the bug exists now, but you're not sure when it started. `git bisect` allows you to mark the current version as "bad" and a previous version as "good." It then helps you narrow down the exact commit where the bug first appeared by checking out different commits and running a test script. If the script returns 0, the commit is good; if it returns a non-zero value, the commit is bad. This process continues until the problematic commit is identified.
- **filter-branch**
  - *Purpose*: This command is used to rewrite the history of a Git repository in a way that can be automated with scripts.

  - *Use cases*: You might use `filter-branch` if you need to change the author email in past commits or if you need to remove a file that contains sensitive information from the entire history. It works by checking out each commit, applying a command to it, and then committing the changes back.
- **Hooks**
  - *Functionality*: Hooks are scripts that Git can run automatically at certain points in your workflow, such as before you commit changes or merge branches.
  - *Why they're useful*: They can help enforce rules or automate tasks. For example, you might use a pre-commit hook to run tests or a linter to ensure code quality before changes are committed. This helps maintain consistency and quality in your codebase.

27 / 27: GitHub



- **GitHub acquired by MSFT for $7.5 billion**
  - In 2018, Microsoft acquired GitHub, a major platform for software development, for $7.5 billion. This acquisition highlighted the importance of GitHub in the tech industry as a central hub for developers to collaborate and share code.
- **GitHub: largest host for Git repositories**
  - GitHub is the largest platform for hosting Git repositories, with over 100 million open-source projects. It provides essential tools for developers, such as:
    * **Git hosting**: Allows developers to store and manage their code.
    * **Pull Requests (PRs) and forks**: Facilitate collaboration by enabling developers to propose changes and work on copies of projects.
    * **Issue tracking**: Helps manage bugs and feature requests.
    * **Code review**: Allows peers to review code changes for quality and consistency.
    * **Collaboration**: Encourages teamwork and communication among developers.

    - **Wiki**: Provides documentation and information about projects.
    - **Actions (CI/CD)**: Automates workflows for continuous integration and continuous deployment, streamlining the development process.
- **"Forking a project"**
  - In open-source communities, "forking" can have a negative connotation, as it sometimes means creating a competing project by modifying the original. However, in GitHub's context, forking is a positive action. It allows users to copy a project to contribute improvements or features without needing direct write access to the original repository. This process is crucial for open-source collaboration, enabling widespread participation and innovation.