# GIT & GITHUB TRAINING

***For Developers***

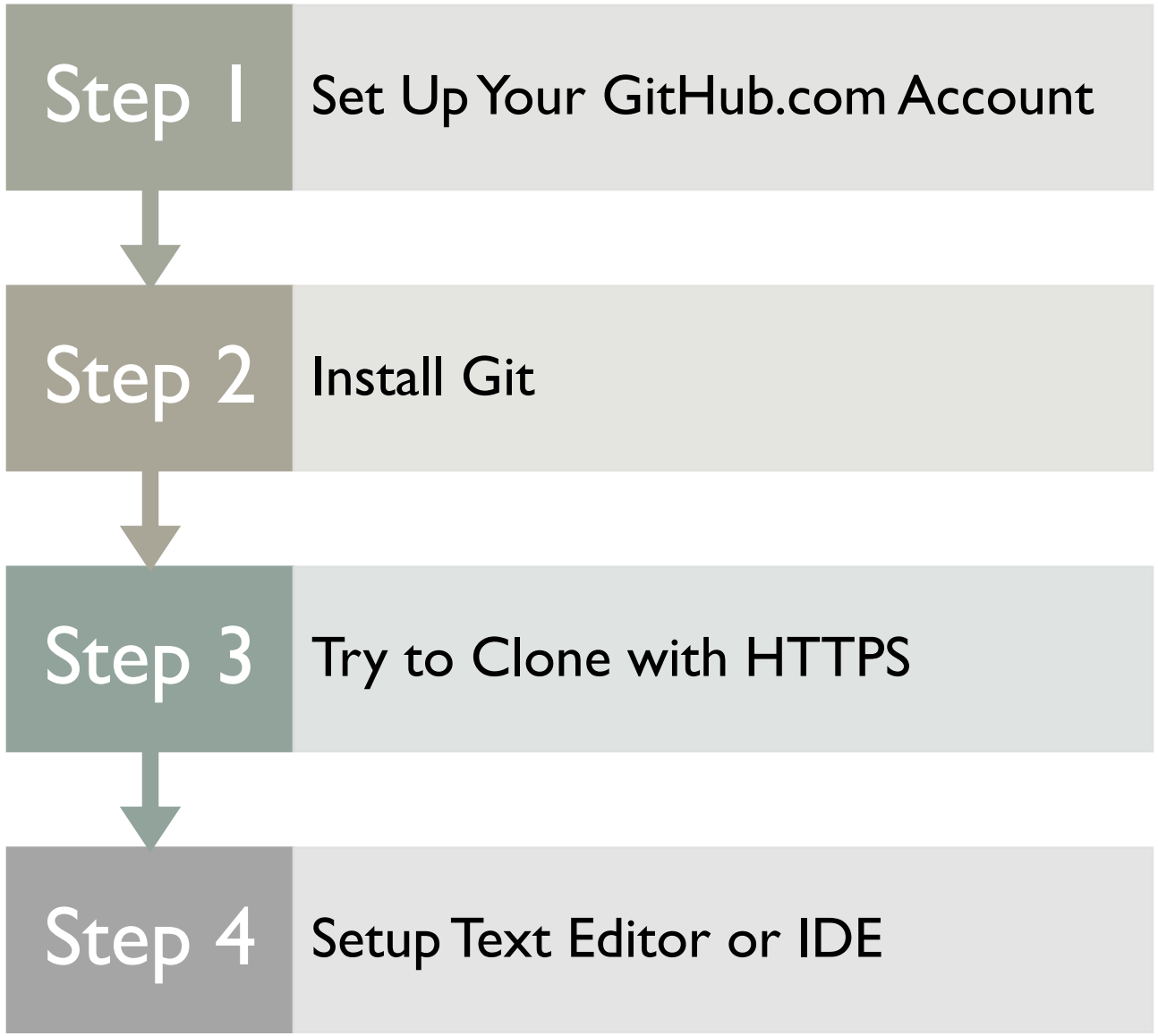
*Andrew Scoppa*
*andrew-scoppa@github.com*

# AGENDA

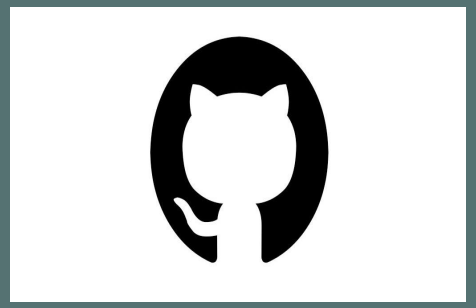
- **Getting Started**
  - Getting Ready for Class
  - Getting Started
  - GitHub Flow
- **Project 1: Caption This**
  - Branching with Git
  - Local Git Configs
  - Working Locally
  - Collaborating on Code
  - Editing on GitHub
  - Merging Pull Requests
  - Local History
  - Streamline Workflow with Aliases
- **Project 2: Merge Conflicts**
  - Defining a merge conflict
  - Resolving merge Conflicts
- **Project 3: GitHub Games**
  - Workflow Review
  - Protected Branches & CODEOWNERS
  - Git Bisect
  - Reverting Commits
  - Helpful Git Commands
  - Viewing Local Changes
  - Tags & Releases
  - Workflow Discussion
- **Project 4: Local Repository**
  - Create a Local Repo
  - Fixing Commit Mistakes
  - Forgot to branch?
  - Rewriting History with Git Reset
  - Cherry Picking
  - Merge Strategies

# GETTING STARTED

# GETTING READY FOR CLASS




Getting Ready for Class



# WHAT IS GITHUB?

## Hosting platform for collaboration and version control

- Built on top of a distributed version control system called **Git**

## Supports the principles of open source

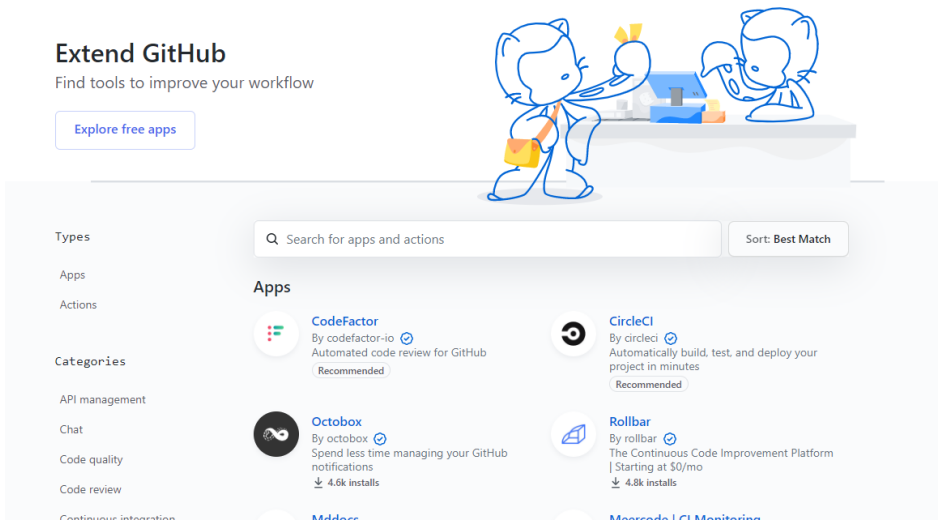
- Transparency, Participation and Collaboration
- To use, study, modify and distribute software for any purpose without legal restraint

## Provides many features to help you and your team collaborate

- Issues
- Pull Requests
- Projects
- Organizations and Teams

Getting Started with Collaboration

# THE GITHUB ECOSYSTEM




- Rather than force you into a "one size fits all" ecosystem, GitHub strives to be the place that brings all of your favorite tools together
- You'll find some new, indispensable tools to help with continuous integration, dependency management, code quality and much more
- For more information on integrations, check out GitHub integrations.

# WHAT IS GIT?

## Git is a distributed version control system

- The complete codebase, including its full history, is mirrored on every developer's computer

## Optimized for local operations

- When you clone a copy of a repository to your local machine, you receive a copy of the entire repository and its history
- Local repositories make it is easy to work offline or remotely

## Git is explicit

- It does not do anything until you tell it to
- Developers commit their work locally, and then explicitly sync their copy of the repository with the copy on the server

# GIT ESSENTIAL CONCEPTS

## Does not store your information as series of changes

- Takes snapshots of  your repository at a point in time
- This snapshot is called a commit

## Optimized for Local Operations

- When you clone a copy of a repository to your local machine, you receive a copy of the entire repository and its history

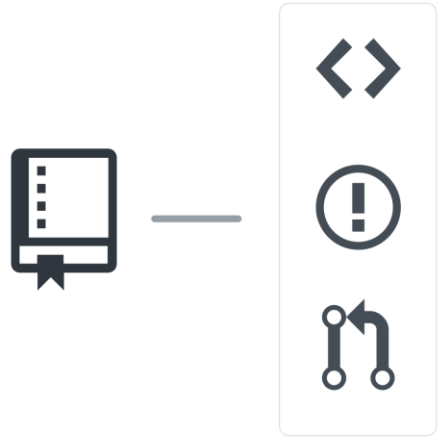
## Branches are Lightweight and Cheap

- Branch is a pointer that corresponds to the most recent commit in a line of work.
- Git keeps the commits for each branch separate until you explicitly tell it to merge those commits into the main line of work

## Git is Explicit

- It does not do anything until you tell it to.
- No auto-saves or auto-syncing with the remote, Git waits for you to tell it when to take a snapshot and when to send that snapshot to the remote.

# ACTIVITY: EXPLORING A GITHUB REPOSITORY



- A GitHub repository offers simple yet powerful tools for collaborating with others
- There are two account types in GitHub, user accounts and organization accounts
- A repository contains all of the project files (including documentation) and stores each file's revision history
- Let's now explore a repository

# ACTIVITY: USING GITHUB ISSUES AND MARKDOWN

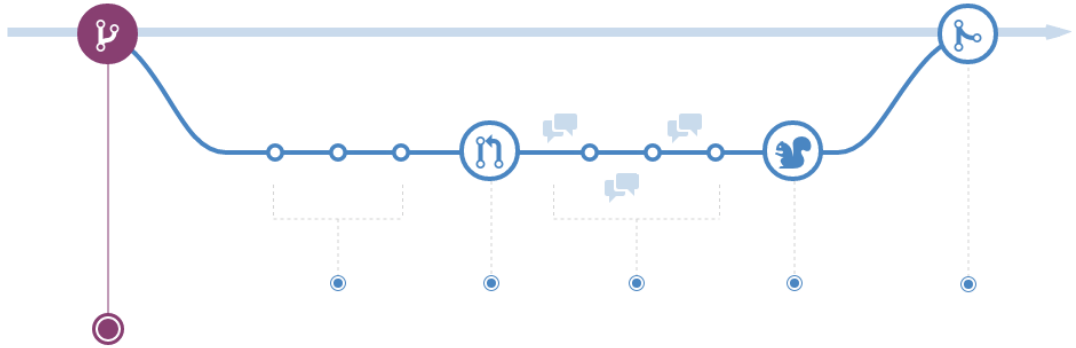
- In GitHub, you will use issues to record and discuss ideas, enhancements, tasks, and bugs. Issues make collaboration easier by:
  - Replacing email for project discussions, ensuring everyone on the team has the complete story, both now and in the future
  - Allowing you to cross-link to related issues and pull requests
  - Creating a single, comprehensive record of how and why you made certain decisions
  - Allowing you to easily pull the right people into a conversation with @ mentions and team mentions
- Let's learn how to create and add collaborate using issues  with markdown

Getting Started

# GITHUB PAGES

- GitHub Pages enable you to host free, static web pages directly from your GitHub repositories
- You can create two types of websites, a user/organization site or a project site. We will be working with project websites.
- For a project site, GitHub will only serve content from a specified branch. You can also choose to publish your site from a /docs folder on the specified branch.
- The rendered sites for our projects will appear at domain.github.io/repo-name.

# REVIEW THE COLLABORATIVE WORKFLOW ENABLED BY GITHUB



- GitHub flow is a lightweight, branch-based workflow that supports teams and projects where deployments are made regularly
- This guide explains how and why GitHub flow works: Understanding the GitHub Workflow

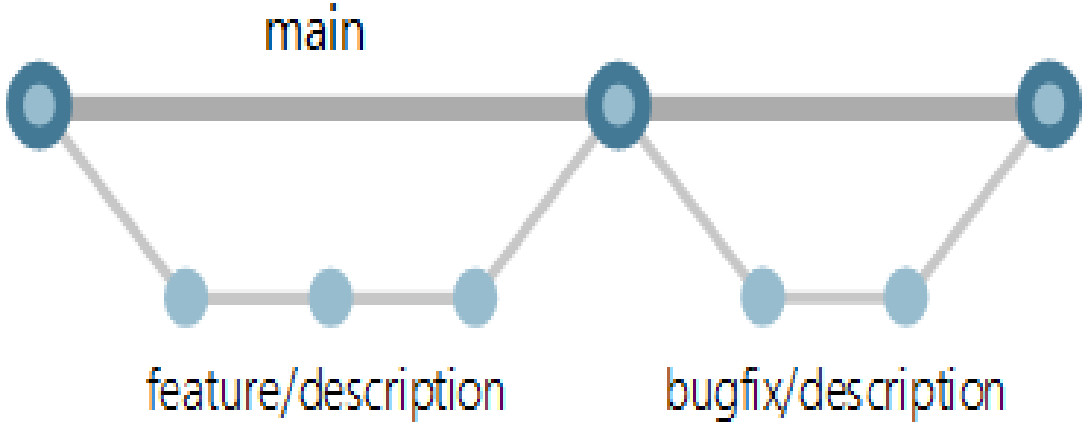
GitHub Flow

# REVIEW THE COLLABORATIVE GIT FLOW


- Many companies follow different models of deployment and git flow.
- Some only require the simple GitHub flow, while others use a more legacy git flow.
- If you follow more of the Git Flow, you will likely need to create additional protected branches like DEV, QA, or PROD.

# PROJECT 1: CAPTION THIS

# ACTIVITY: CREATING A BRANCH WITH GITHUB




- The first step in the GitHub Workflow is to create a branch, which is like a "sandbox"
- When you create a branch, you are not creating a physical copy on disk. In the background, a *branch is just a pointer*.
- Now let's create one

Branching with Git

Branching Video

# GIT CONFIGURATION LEVELS

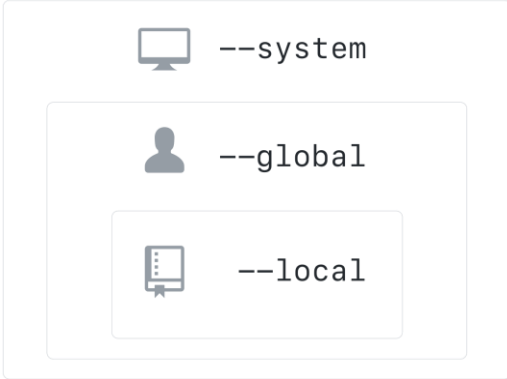
- Git allows you to set configuration options at three different levels.
  - ~system

  These are system-wide configurations. They apply to all users on this computer.
  - ~global

  These are the user level configurations. They only apply to your user account.
  - ~local

  These are the repository level configurations. They only apply to the specific repository where they are set.

  The default value for git config is ~local




Local Git Configs

# ACTIVITY: USING CONFIGURATION SETTINGS

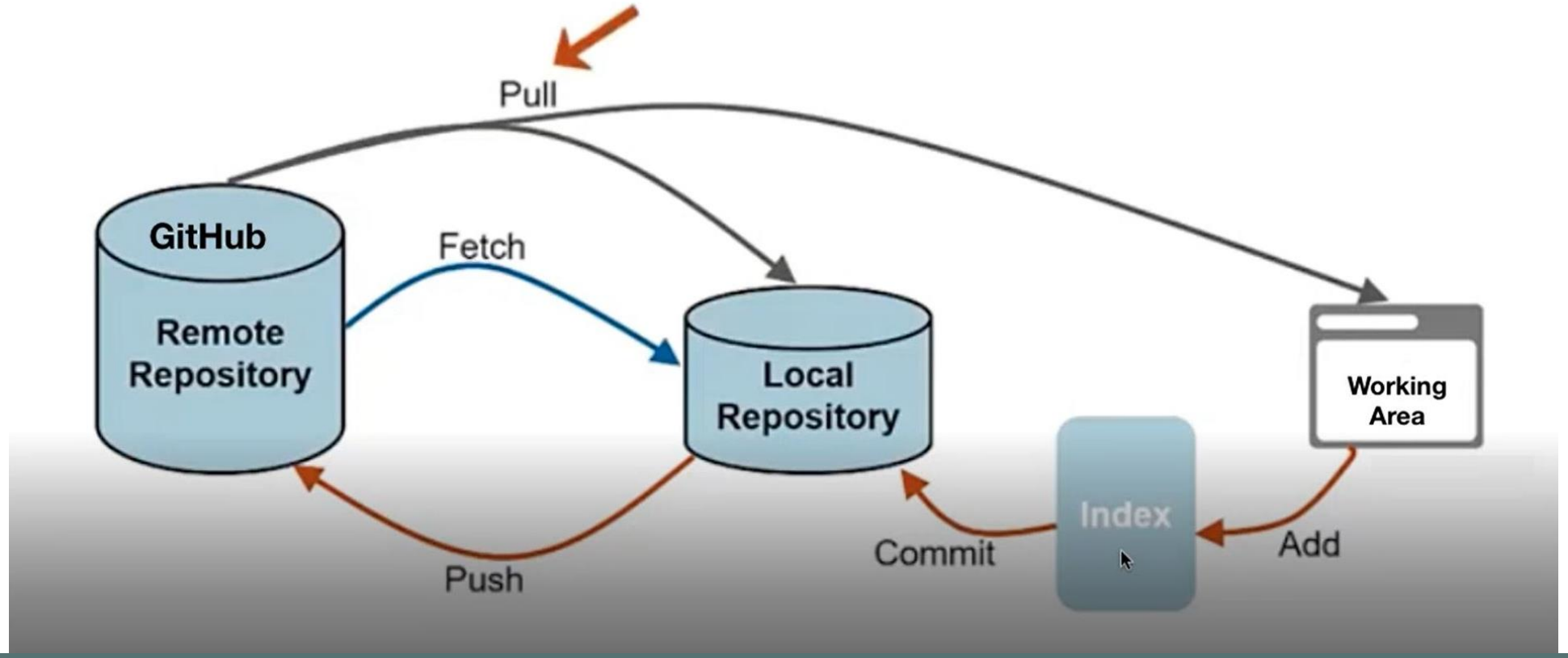
GIT Configuration

Let's prepare your local environment to work with Git

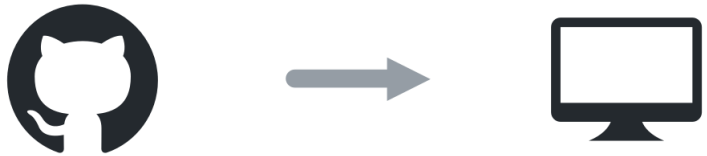
- Viewing Your Configurations
- Configuring Your User Name and Email
- Git Config and Your Privacy
- Configuring autocrlf

Local Git Configs

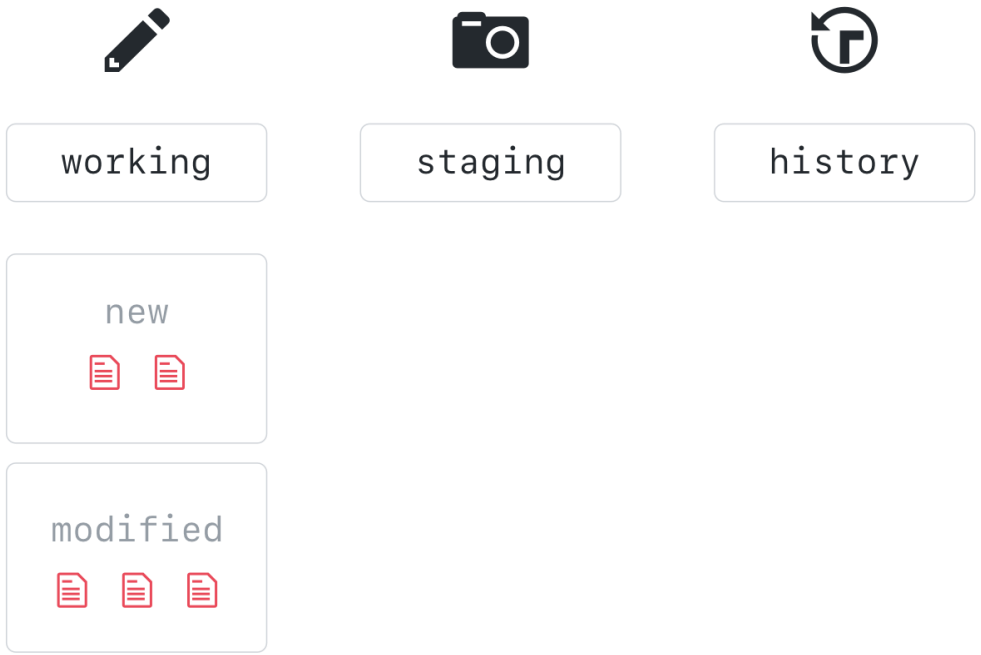
# REVIEW: LOCAL GIT COMMANDS



# ACTIVITY: CLONE, BRANCH, AND EDIT

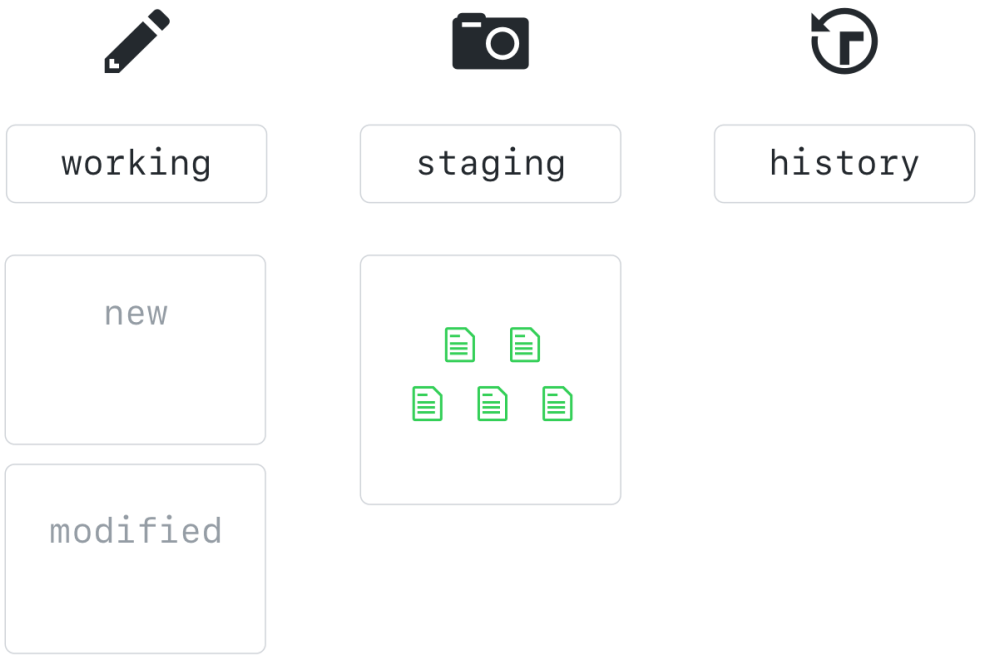


- Clone a repository
  - Creates a copy of everything in that repository, including its history
- Check status
- Review branches and switching
- Edit your file



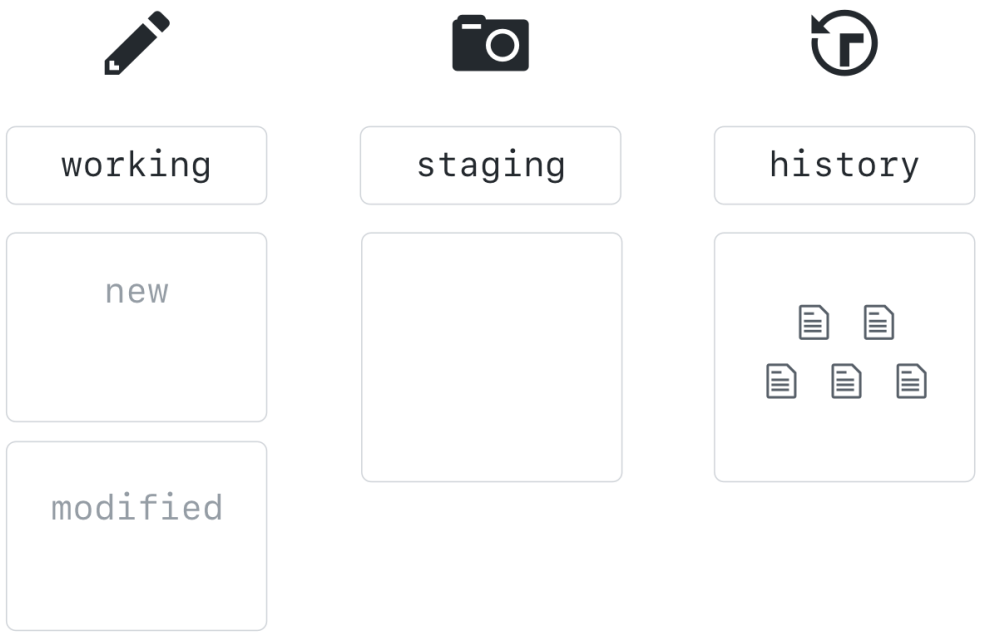

# TWO STAGE COMMIT

- When you work locally (from command line), your files exist in one of four states.
  - They are either untracked, modified, staged, or committed.
  - An untracked file is a new file that has never been committed.
- Git tracks these files and keeps track of your history by organizing your files and changes in three working trees.
- When we are actively making changes to files, this is happening in the working tree.




# STAGE

- To add these files to version control, you will create a collection of files that represent a discrete unit of work.
- We build this unit in the staging area



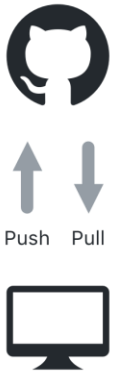

# COMMIT

- When we are satisfied with the unit of work we have assembled, we will take a snapshot of everything in the staging area.
- This snapshot is called a commit
- This commit becomes part of the development history

# ACTIVITY: COMMIT

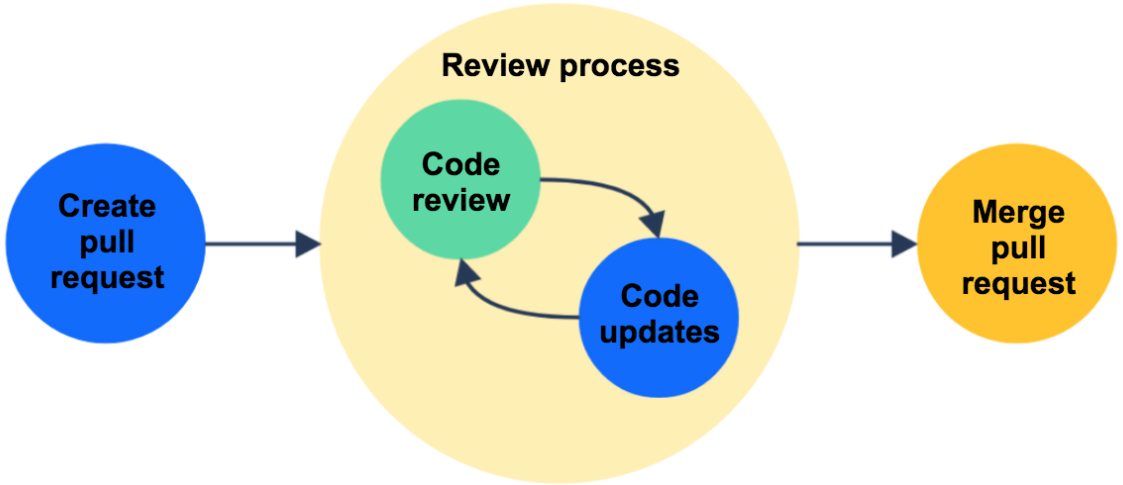
- In order to make a file part of the version-controlled directory we will first do a **git add** and then we will do a **git commit**
- Be sure to use a descriptive commit message!

# ACTIVITY: PUSHING CHANGES TO GITHUB



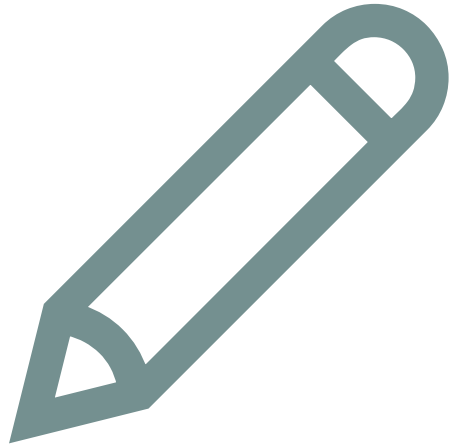

- Now that you have made changes locally, let's learn how to push your changes back to the shared class repository for collaboration
  - Push your changes to GitHub
  - Use the command **git push**

# ACTIVITIES: CREATING AND EXPLORING A PULL REQUEST, AND CODE REVIEW




- Now that you have started to change your file, you will open a pull request to discuss the file with your team
- Pull Requests are used to propose changes to the project files. A pull request introduces an action that addresses an Issue.
- A Pull Request is considered a "work in progress" until it is merged into the project.

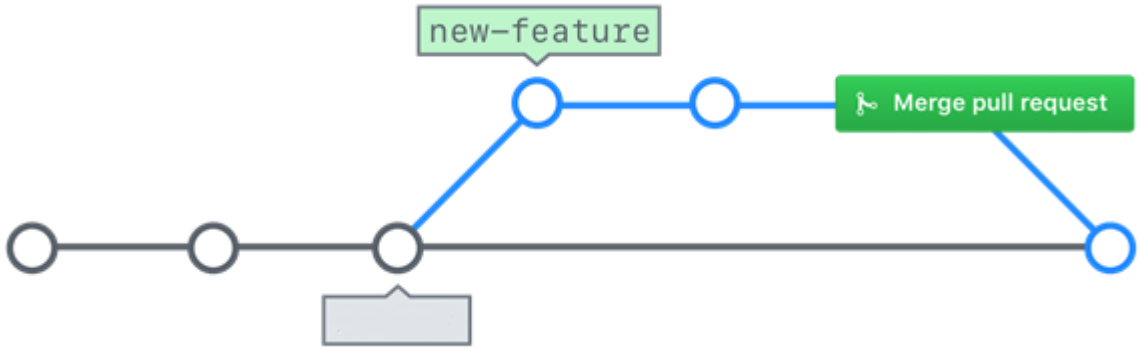
# ACTIVITY: EDITING FILES IN PULL REQUESTS



- When you create a pull request, you will be notified when someone adds a comment or a review.
- Sometimes, the reviewer will ask you to make a change to the file you just created.
- Let's see how GitHub makes this easy

Editing on GitHub

# ACTIVITY: MERGING PULL REQUEST AND UPDATING LOCAL REPO




Now that you have made the requested changes, your pull request should be ready to merge

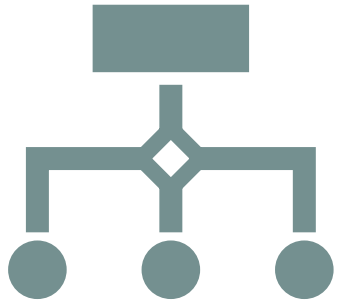
When you merge your branch, you are taking the content and history from your feature branch and adding it to the content and history of the main branch

Who is responsible for performing a merge on GitHub?

**Be sure to update your local repository when finished**

Merging Pull Requests

# ACTIVITY: REVIEW USING GIT LOG



- When you clone a repository, you receive the history of all of the commits made in that repository
- The log command allows us to view that history on our local machine
- Let's take a look at some of the option switches you can use to customize your view of the project history

Local History

# STREAMLINE WORKFLOWS WITH ALIASES

- An alias allows you to type a shortened command to represent a long string on the command line
  - Original Command

    git rev-parse --show-toplevel
  - Alias

    git config –global alias.root-folder “rev-parse --show-toplevel”
  - Use

    cd $(git root-folder)

# PROJECT 2: MERGE CONFLICTS

# WHAT IS A MERGE CONFLICT?

- Merge conflicts happen when you merge branches that have competing commits, and Git needs your help to decide which changes to incorporate in the final merge
  - Changes to the same "hunk" of the same file
  - Two different branches
  - Changes on both branches happened since the branches have diverged
- It's important to know who to ask in case you aren't sure how to resolve the conflict on your own



Defining a merge conflict

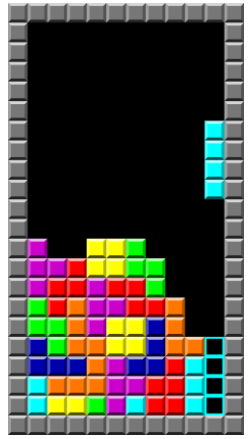
# ACTIVITY: RESOLVING A MERGE CONFLICT

- Let's try to create a merge conflict, and fix it together

Resolving merge Conflicts

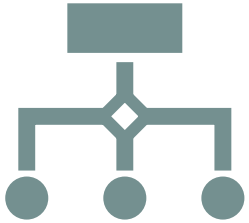
# PROJECT 3: GITHUB GAMES

# ACTIVITY: INTRODUCING GITHUB GAMES



- In this section, we will work on a project repository called github-games
- A github-games repository has been created for you in the githubschool organization.
- Your  repository name is github-games-USERNAME

# PROTECTED BRANCHES & CODEOWNERS



**With protectect branches you can require pull requests to pass a set of checks before they can be merged**

For example, you can block pull requests that don't pass status checks or require that pull requests have a specific number of approving reviews before they can be merged

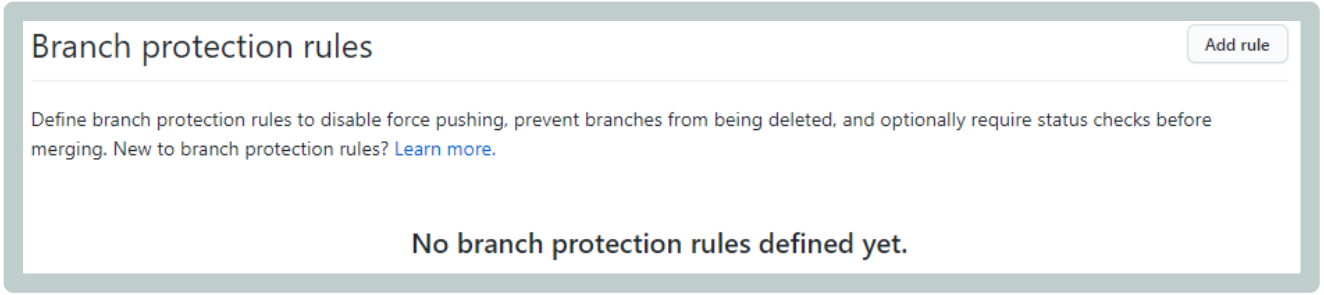
**You can use a CODEOWNERS file to define individuals or teams that are responsible for code in a repository**

Code owners are automatically requested for review when someone opens a pull request that modifies code that they own



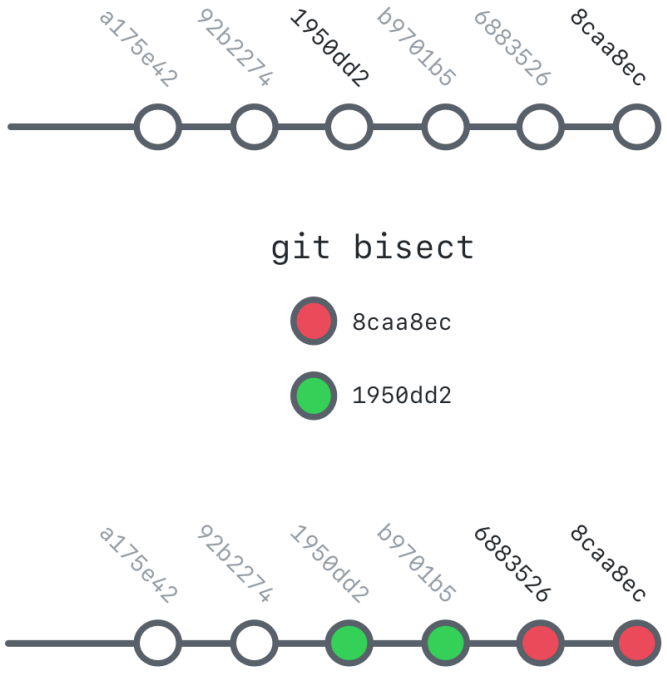
Protected Branches & CODEOWNERS

# ACTIVITY: ENABLING PROTECTED BANCHES & CREATING CODEOWNERS FILE




- First let's enable protected branches
- Next, we'll create a CODEOWNERS file

Protected Branches & CODEOWNERS

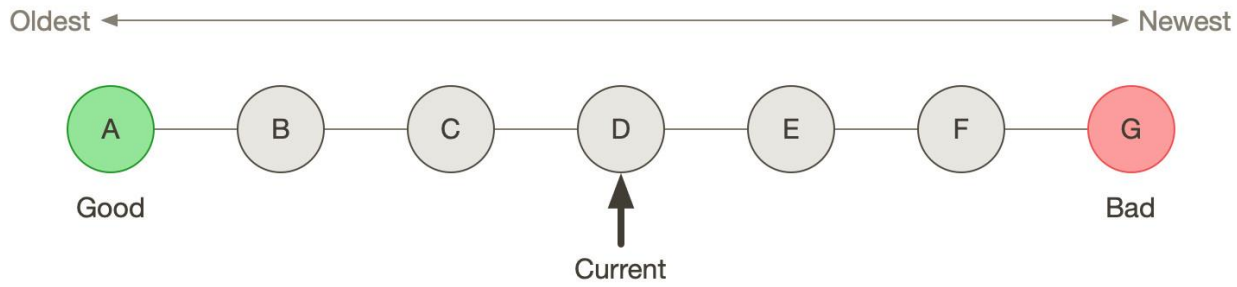



# WHAT IS GIT BISECT?

- Using a binary search, git bisect can help us detect specific events in our code. For example, you could use bisect to locate the commit where:
  - a bug was introduced.
  - a new feature was added.
  - a benchmark's performance improved.

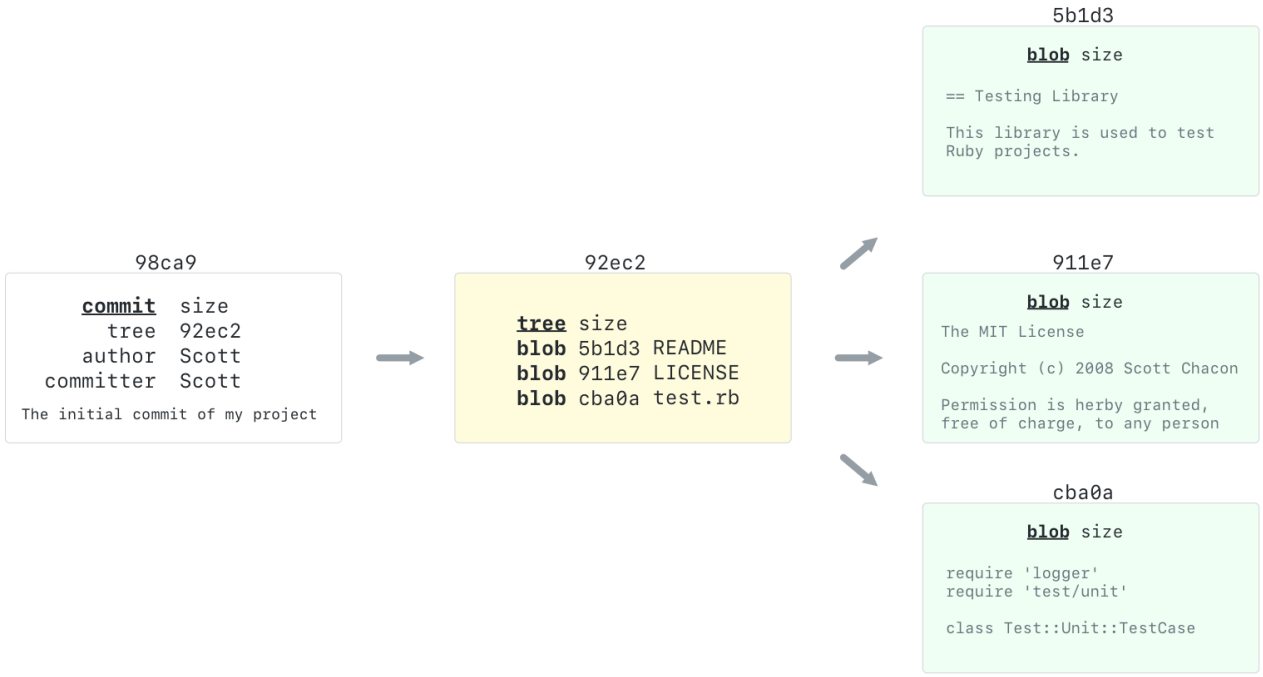
Git Bisect

# ACTIVITY: FINDING THE BUG IN OUR PROJECT



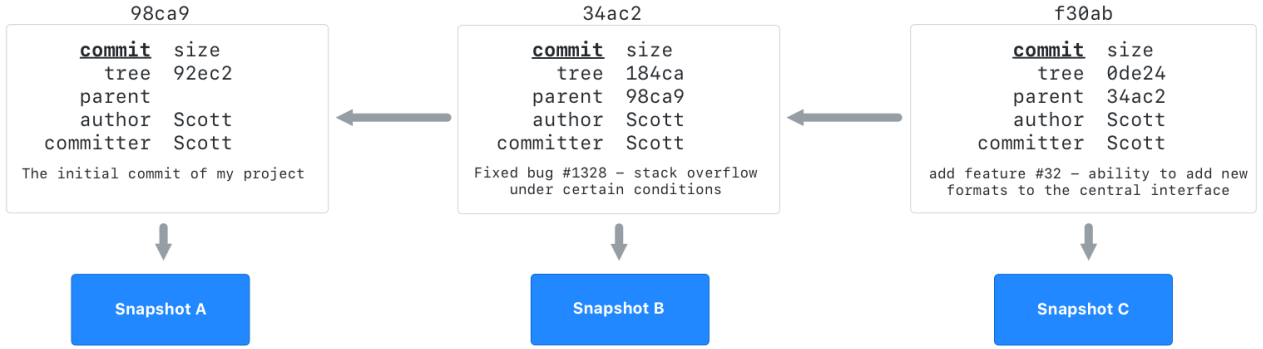

- Let's use **git bisect** to locate the commit that introduced a bug in our project

Git Bisect




# HOW ARE COMMITS MADE? (1/2)

- Every commit in Git is a unique snapshot of the project at that point in time. It contains the following information:
  - Pointers to the current objects in the repository
  - Commit author and email (from your config settings)
  - Commit date and time
  - Commit message




# HOW ARE COMMITS MADE? (2/2)

- Each commit also contains the commit ID of its parent commit

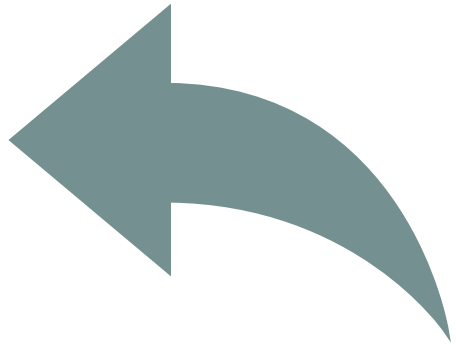
| Command | Cautions |
| --- | --- |
| revert | Generally safe since it creates a new commit. |
| commit --amend | Only use on local commits. |
| reset | Only use on local commits. |
| cherry-pick | Only use on local commits. |
| rebase | Only use on local commits. |

Reverting Commits

# COMMANDS THAT RE-WRITE HISTORY

- Git's data structure gives it integrity, but its distributed nature also requires us to be aware of how certain operations will impact the commits that have already been shared
- If an operation will change a commit ID that has been pushed to the remote (also known as a public commit), we must be careful in choosing the operations to perform

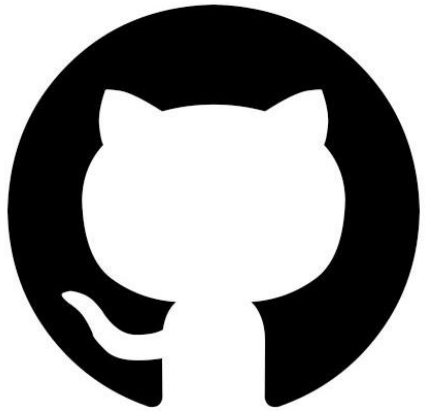
# ACTIVITY: REVERTING A COMMIT



- Let's get the game working
  - Reverse the commit that incorrectly renames index.html

Reverting Commits

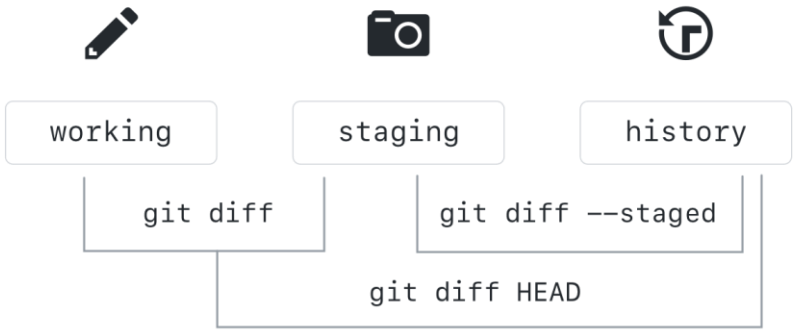
# ACTIVITY: EXPLORE SOME HELPFUL GIT COMMANDS



- Let's explore some helpful techniques with Git commands
  - Moving and renaming files with git
  - Staging hunks of changes

Helpful Git Commands

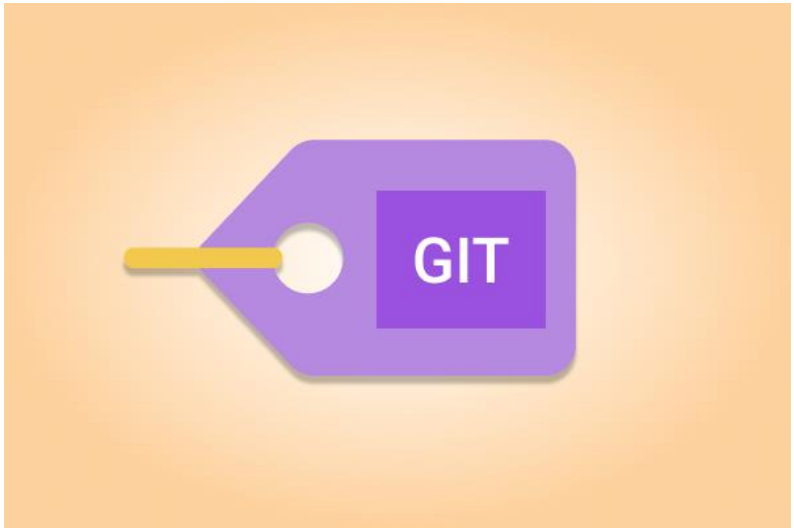
# ACTIVITY: VIEWING LOCAL CHANGES




- Now that you have some files in the staging area and the working directory, let's explore how you can compare different points in your repository
- **git diff** allows you to see the difference between any two refs in the repository

Viewing Local Changes

# ACTIVITY: TAGS AND RELEASE




- Let's add a release to GitHub Games
- A tag is a pointer that points to a specific commit
  - two variants, an annotated tag and a lightweight tag

  git tag -a v1.0 <SHA>
- Releases are a GitHub feature that allow you to add an executable to the tag for easier access by visitors who just want to download and install your software
  - Releases are tags, because they point to a specific commit and can be named like any other tag. However, releases can also include attached binaries

Tags & Releases

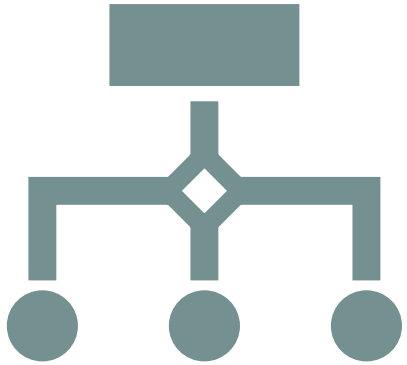
# DISCUSSION GUIDE: TEAM WORKFLOWS AND BRANCHING STRATEGIES

| | | | |
|---|---|---|---|
| Which branching strategy will we use? | Which branch will serve as our "main" or deployed code? | How will you protect your code? | Will we use naming conventions for our branches? |
| How will we use labels and assignees? | Will we use milestones? | Will we have required elements of Issues or Pull Requests (e.g. shipping checklists)? | Who is expected to review your work? Do you plan to involve other teams? |
| How will we indicate sign-off on Pull Requests? | Who will merge pull requests? | How will you teach your workflow to your team? If it already exists, how is it taught to new hires? | If users have questions about Git, GitHub, or their workflows, who do they ask? How do they know who to ask? |

Workflow Discussion

# PROJECT 4: LOCAL REPOSITORY

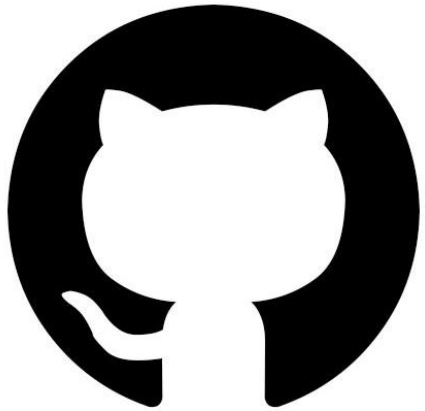
# ACTIVITY: INITIALIZING A NEW LOCAL REPOSITORY



- Let's create a local repository that we can use to practice the next set of commands

Create a Local Repo

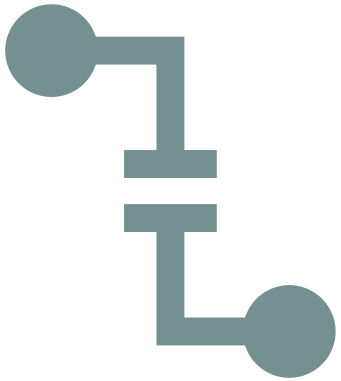
# ACTIVITY: FIXING COMMON MISTAKES



- Let's explore some of the ways Git and GitHub can help us shape our project history
  - Revise your last commit with **git commit --amend**
  - Let's see this in action

Fixing Commit Mistakes

# ACTIVITY: FORGOT TO BRANCH? NO PROBLEM!



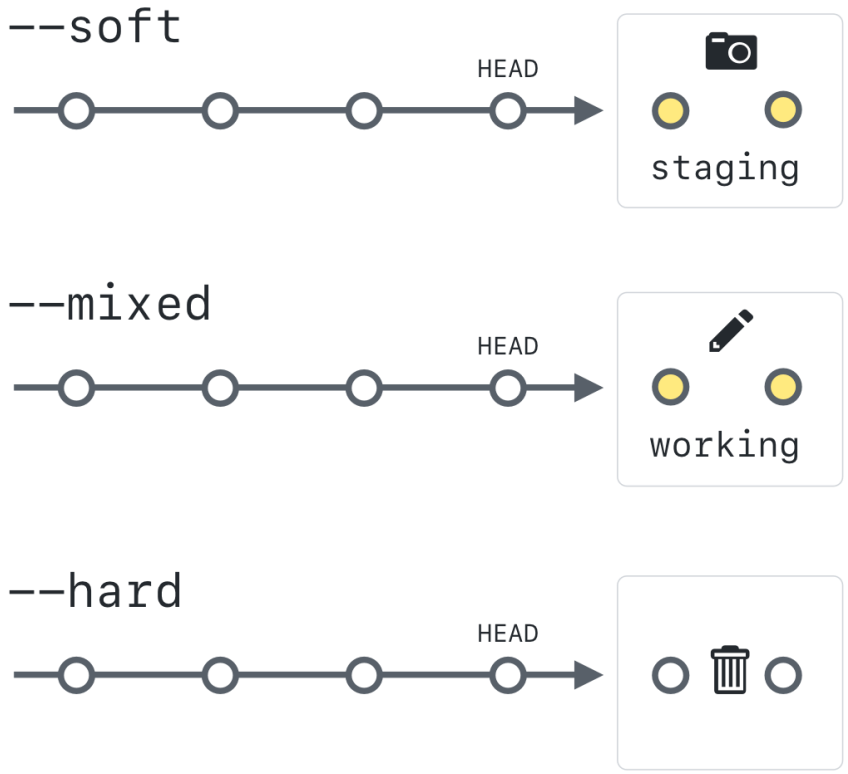
- If you begin working, but you are still on main, don’t panic!
  - Changes aren't “on” a branch until they are committed
  - Your working directory and staging area are attached to HEAD
  - HEAD points to the last commit on the current checked-out branch
  - So, you can change branches, and any changes that are not committed will come with you
- Let’s create a branch after making changes

Forgot to branch?


# REWRITING HISTORY WITH GIT RESET

- When you want to make changes to commits further back in history, you will need to use a powerful command: **git reset**
- You can reset some, or even all, of your files to look like what they were at a different point in history

Rewriting History with Git Reset




# RESET MODES

- There are three different snapshots of our project at any given time.

1. Most recent commit (also known as HEAD).
2. Staging area (also called the index).
3. Working directory containing any new, deleted, or modified files.

- The **git reset** command has three modes, and they allow us to change some or all of these three snapshots.

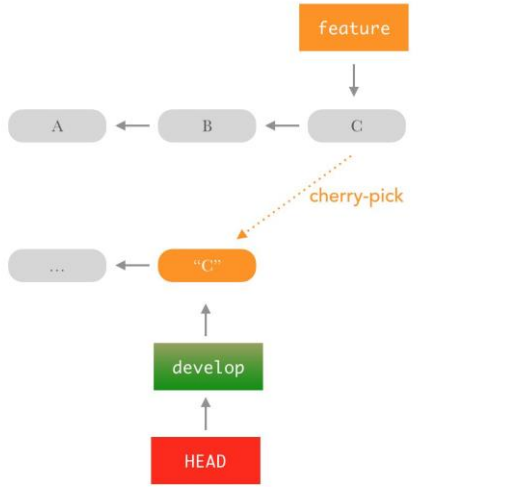
[Rewriting History with Git Reset]

# ACTIVITY: USING SOFT, MIXED, AND HARD RESET


- Let's work with the three modes for git reset:
  - **--soft**, **--mixed**, and **--hard**
- For this activity, assume that we have a "clean" working directory, i.e. there are no uncommited changes

Rewriting History with Git Reset

# ACTIVITY: GET IT BACK WITH CHERRY PICKING!




- We just learned how **reflog** can help us find local changes that have been discarded.
- Cherry picking allows you to pick up a commit from your reflog or another branch of your project and move it to your current branch
  - *gone* doesn't really mean *gone!*
- Let's cherry pick a commit

Cherry Picking

# UNDERSTANDING GIT MERGE STRATEGIES

Fast Forward - A fast forward merge assumes that no changes have been made on the base branch since the feature branch was created. This means that the branch pointer for base can simply be "fast forwarded" to point to the same commit as the feature branch.

Recursive - A recursive merge means that changes have been made on both the base branch and the feature branch and git needs to recursively combine them. With a recursive merge, a new "merge commit" is made to mark the point in time when the two branches came together. This merge commit is special because it has more than one parent.

Octopus - A merge of 3 or more branches is an octopus merge. This will also create a merge commit with multiple parents.

Merge Strategies

# ACTIVITY: USE GIT REBASE TO MODIFY COMMIT HOSTORY


`git rebase dev`

- **git rebase** enables you to modify your commit history in a variety of ways.
  - For example, you can use it to reorder commits, edit them, squash multiple commits into one, and much more
- To enable all of this, rebase comes in several forms.
- We'll be using interactive rebase: **git rebase --interactive**, or **git rebase -i** for short

Merge Strategies

# ADDITION TOPICS OF INTEREST

| | | | |
|---|---|---|---|
| Actions | Automate releases | .gitignore file | Commit signature verification |
| Stash | IFS | Submodules | Subtrees |

# APPENDIX

.gitignore and .gitattributes

# .GITIGNORE

- A .gitignore file is a plain text file where each line contains a pattern for files/directories to ignore
  - often located in the root folder of the repo
  - can reside in any folder in the repo
  - can have multiple .gitignore files
  - patterns are relative to the location of the .gitignore file
- Use check-ignore to report on files ignored by git
  - git check-ignore –verbose ignore.log
- Use config.excludesfile to setup a global .gitignore across all repos on your local system
  - git config core.excludesfile C:\Users\instructor\.gitignore_global

# .GITIGNORE SAMPLES

```
# ignore a single file
mycode.class

# ignore an entire directory
mydebugdir/

# ignore a file type
*.json

# add an exception (using !) to the preceding rule to track a specific file
!package.json

# match any number of directories with double asterisk
logs/**/*.log
```

A collection of .gitignore templates

# .GITATTRIBUTES

- Lets you specify the attributes that git will apply to pathnames to use when performing actions such as merge, commit, and diff
- Configured in either a .gitattributes file in one of your directories (usually the root of your repo)
- Or in the .git/info/attributes file if you don't want the attributes file committed with your project

- A Collection of Useful .gitattributes Templates