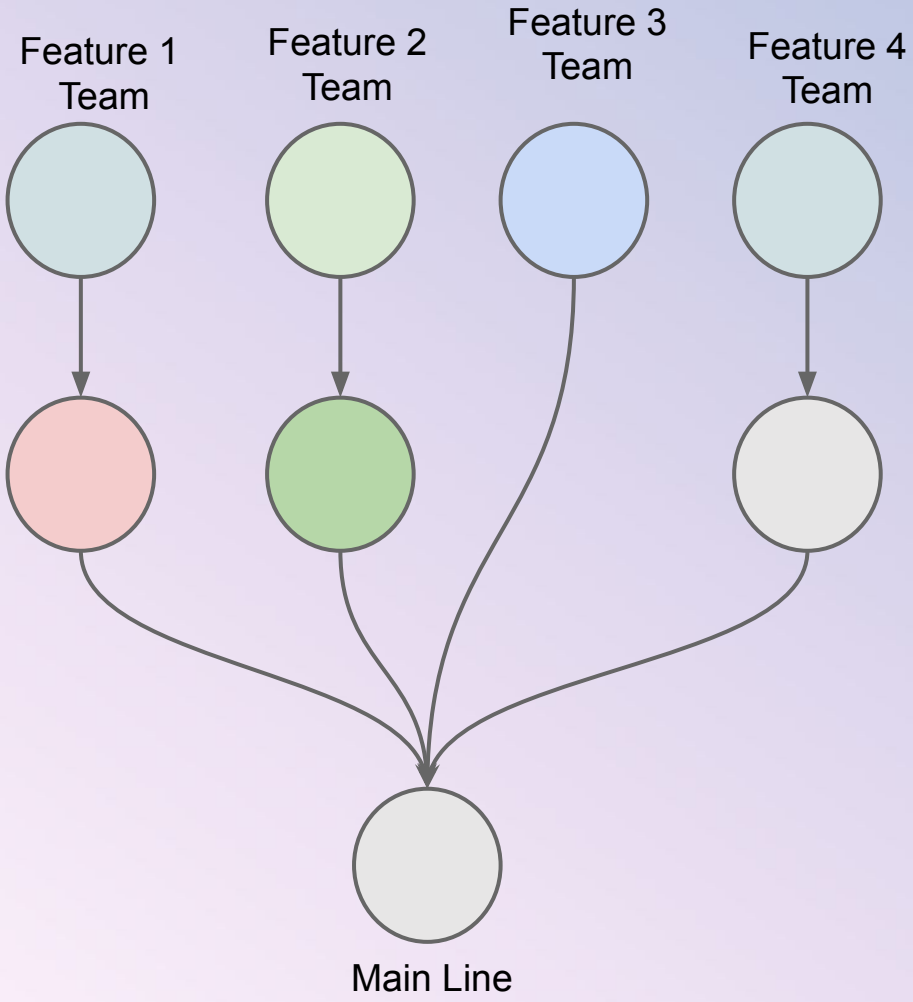
Get better at Git

End chaos in your collaboration

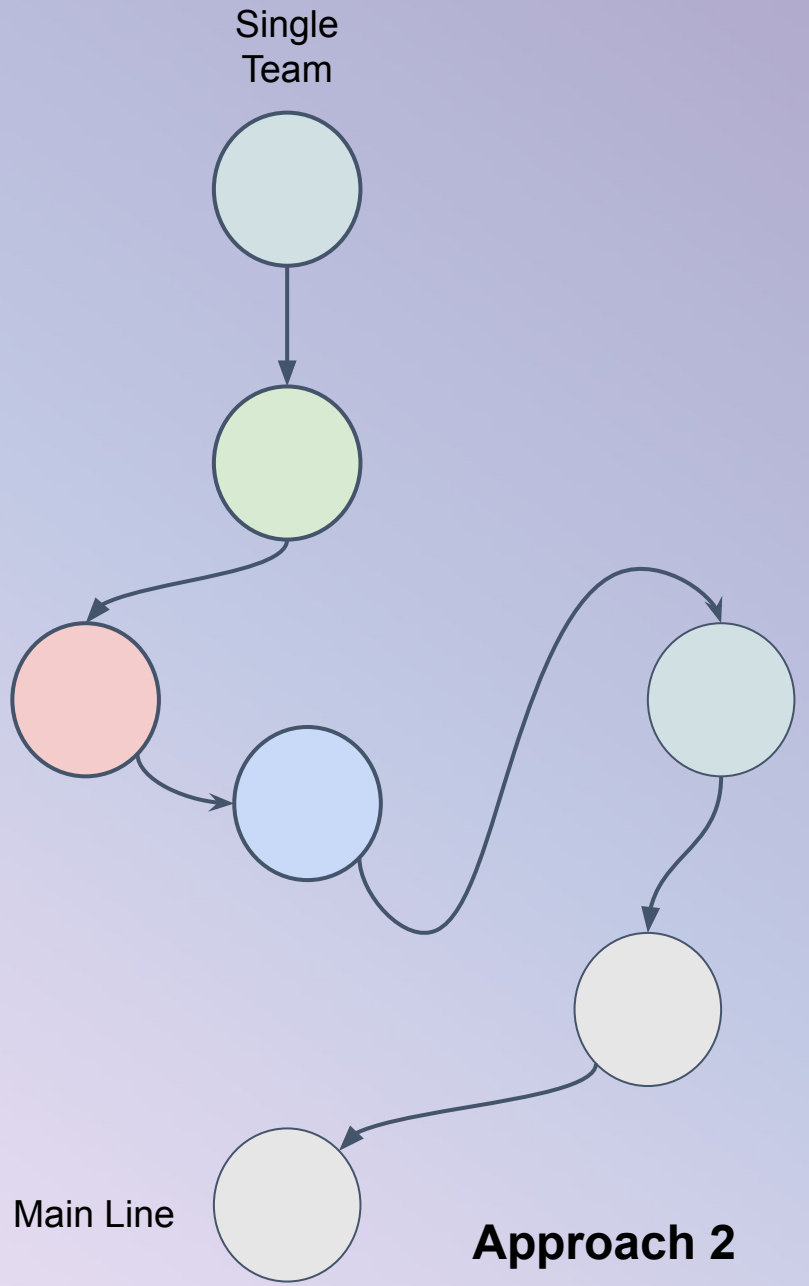
Organizing a party

Feature 1	Selecting Theme
Feature 2	Deciding Food Menu
Feature 3	Curate Guest List
Feature 4	Planning Activities

Two Approaches



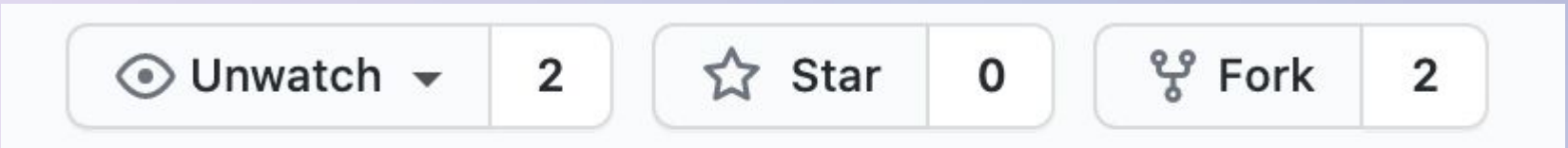
Approach 1



Approach 2

Fork and clone the repo

1. Go to the repo (Link) and fork it



2. Create a new working directory where you want to work

```
mkdir osd-summer
```

3. Change current working directory

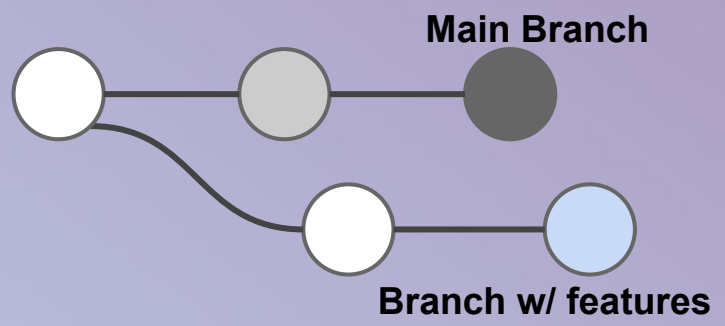
```
cd osd-summer
```

4. Clone the forked repo

```
git clone [repo-link]
```

5. Press Enter. Your local copy will be created

Branching



Branches allow you to build new features, fix bugs and test out ideas without putting your main project at risk.

Benefits of Branching

Independence diverge from the main line of development and continue to do work without messing with the line.

Modularity switching back and forth between branches allows you to maintain each module separately.

Experimentation testing out changes and new features without worrying about breaking things in production.

Types of branches

Local Branch is a branch that only you (the local user) can see. It exists only on your local machine.

Remote Branch is a branch on the remote server. Other users can also see it.

Branching

1. Check your default branch

```
git branch
```

```
* main
```

2. Create a new branch

```
git branch <branch-name>
```

Naming Best Practices

- **Using prefixes** like hotfix/, feature/, bug/ to categorise the task.
- If you are using a task tracking software like Jira, **prefixing with ticket tracker ID** is optimal.
- Prefixes are not enough, you should also to add a **concise description of the task** to provide more context.
- **Separators** - Use hyphens or underscore. Be consistent.

Branching

1. Check your local branches

```
git branch
```

```
  feature/plan-activity
* main
```

2. Check your remote branches

```
git branch -r
```

```
  origin/HEAD -> origin/main
  origin/main
```

3. Switching to another branch

git checkout [branch-name]

```
git checkout feature/plan-activity
```

```
Switched to branch 'feature/plan-activity'
```

Creating a new branch and switching at the same time

git checkout -b [branch-name]

```
git checkout -b feature/select-theme
```

```
Switched to a new branch 'feature/select-theme'
```

Git Status and Git Diff

1. Check status of your branch

```
git status
```

```
nothing to commit, working tree clean
```

2. Create a new file and make changes to the file in your branch
3. Now check the status of your branch

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   plan_activities.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

4. Now we have made changes to the file and want to review those changes. git diff show changes between commits, commit and working tree etc.

```
git diff
```

```
diff --git a/plan_activities.txt b/plan_activities.txt
index f518807..f344a92 100644
--- a/plan_activities.txt
+++ b/plan_activities.txt
@@ -1 +1,2 @@
 # This file will be used to plan activities for the party
+# Add your activity ideas below
```

5. Add your changes using *git add .*

6. Check status

```
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   plan_activities.txt
```

7. Now commit the changes and push

git commit -m [message]

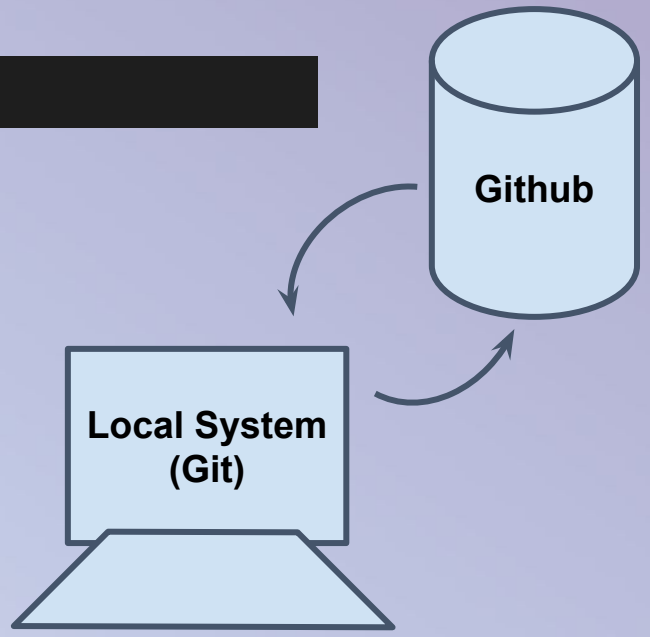
git push --set-upstream origin [branch-name]

Git Fetch, Git Merge, and Git Pull

```
git fetch
```

What it does:

- Retrieves the latest work done by other collaborators on the central repository
- Fetch will grab all the new remote-tracking branches and tags without merging those changes into your own branches.
- It will **not change** any of your **local files**—just updates the tracking data stored in the .git folder.



Git Fetch, Git Merge, and Git Pull

```
git merge [branch-name]
```

What it does:

- combines changes from different branches, typically you will merge your local changes with changes made by others on the central repository.

```
git pull origin [branch-name]
```

It is a convenient shortcut for completing both git fetch and git merge in the same command.

Git Fetch, Git Merge, and Git Pull

1. **Create a new branch** *git checkout -b feature/select-theme*
2. **Add a new file** *vi theme.txt*
3. **Add theme ideas in the file**
4. **Push your changes**

   git add .

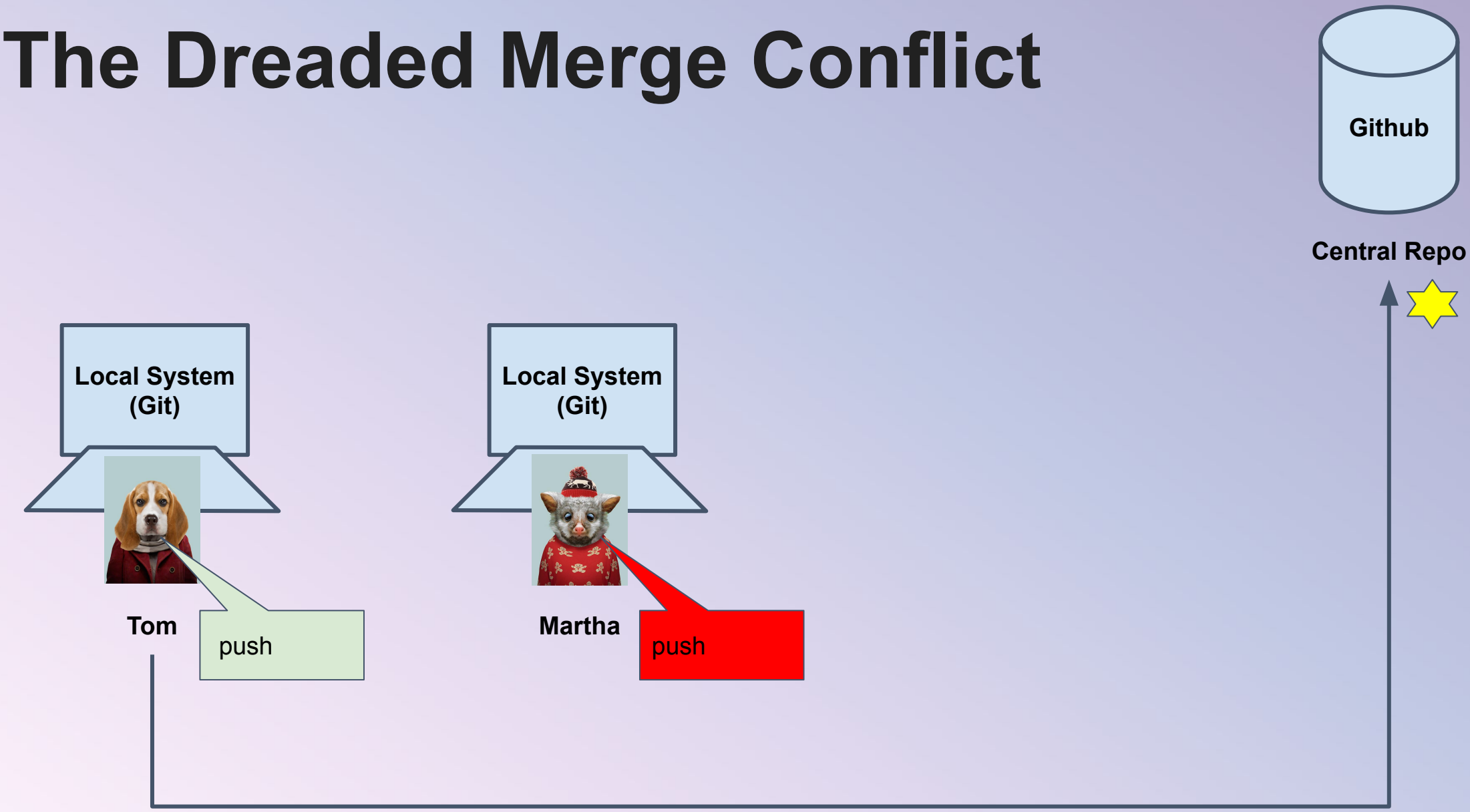
   git commit -m [your message]

   git push --set-upstream origin feature/select-theme
5. **Checkout to main branch** *git checkout main*
6. **Check the existing files in branch using list command** *ls*
7. **Merge the branches and check files again**

   git merge origin/feature/plan_activities

   git merge origin/feature/select_theme

The Dreaded Merge Conflict



Activity: Merge Conflict (Pt. 1)

1. Choose a feature you want to work on
2. Check out a new branch

   ```
   git checkout -b plan_activity
   ```

3. Make changes in the plan_activity.txt file
4. Add, commit and push your changes
5. Create a PR

Activity: Merge Conflict (Pt. 2)

Merge Conflict? Fret not! We will now see how we can resolve then.

1. Pull the main branch changes into your local branch
   ```
   git pull origin main
   ```
2. Open your favourite editor and edit the files to resolve conflict
3. Commit and Push your changes to the remote repository
4. Merge your changes in a PR

Git Log - See your git history

```
codespace → /workspaces/osd-get-better-at-git (main) $ git log
commit f3b9221c64f9550cb805726508c367b48ca90226 (HEAD -> main, origin/main, origin/HEAD)
Merge: 1034dd5 2c0aab2
Author: annu joshi <annuhansa94@gmail.com>
Date:   Thu Jul 15 02:39:47 2021 +0530

    Merge pull request #4 from AnnuJoshi/feature/select-fancy-theme

    Fancy theme file

commit 2c0aab2c798f4b8032e3d0751ade2c9a9ab9206b (origin/feature/select-fancy-theme)
Author: annujoshi <annuhansa94@gmail.com>
Date:   Thu Jul 15 02:38:57 2021 +0530
:...skipping...
```

Revert

```
git revert [--[no-]edit] [--no-commit] <commit-sha>
```

What it does:

- Adds a revert commit at the end of the branch to undo changes in a specific commit
- Passing the **--no-commit** flag allows you to make the changes to your files without committing them

What it doesn't do:

- While reverting will undo your changes to bring your files back to a previous state, it won't remove the commit from your git history. You can still check out the commit and see the changes.

Activity: Revert a Merge (Pt. 1)

1. Check out a new branch:
   `git checkout -b revert-activity`
2. While on the new branch, delete ALL the files in the directory (trust the process here!)
3. Commit your changes:
   `git commit -m "deleted all the files"`
4. Check out the main branch:
   `git checkout main`
5. Merge the revert-activity branch:
   `git merge revert-activity`
6. Type `ls` into the terminal to verify that the files have been deleted.

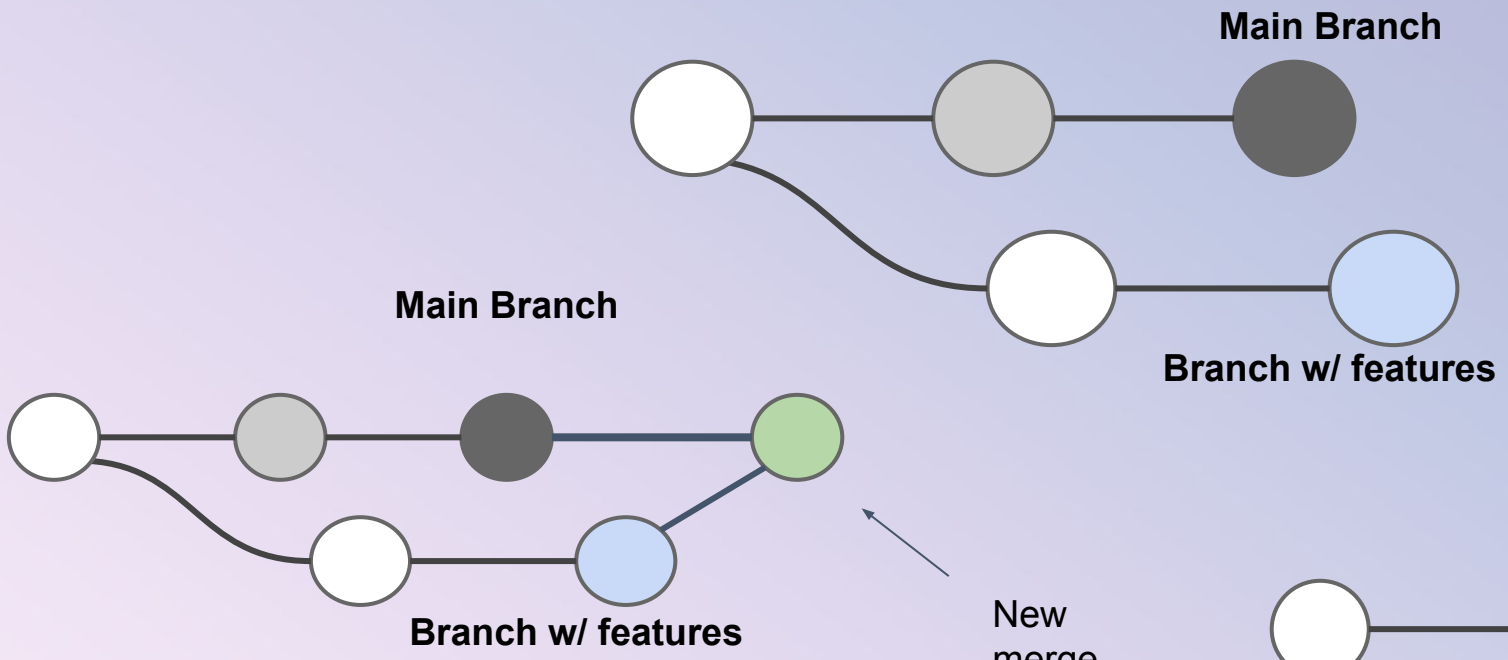
Activity: Revert a Merge (Pt. 2)

Oh no! Our files are all gone! How do we get them back? By reverting the merge, of course!

1. Use `git log` to show the commit history. Find a commit with the message "Merge branch 'revert-activity'..." and copy its commit hash.
2. Run the following command:
   `git revert <commit hash you copied>`
3. Now if you type `ls` you should see that the missing files are back! `git log` will show you that a revert commit has been added to your git history.

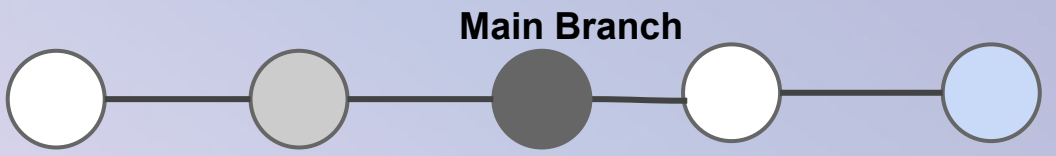
.

Merge vs Rebase



How do we get these changes from the feature branch into the main branch?

New merge commit



Merge:
Creates a new merge commit that pulls in changes from feature branch
Git log shows commits in chronological order

Rebase:
Apply new changes at tip of main branch
Rewrites the git history so the new feature shows up after what was previously in the main branch

Rewriting History with Interactive Rebase

```
git rebase -i <commit or branch>
```

What it does:

- "replays" commits on top of a parent commit or branch that you specify, step by step, resolving any merge conflicts along the way.
- edit specific commits and their commit messages
- squash multiple commits into one commit
- delete a commit from your git history

What it doesn't do:

- Interactive rebase can delete a commit from the history of a remote tracking branch (like on GitHub) but it is still possible to recover that commit locally using **git reflog**

Force Push

If the git history of your local branch diverges from the git history of your remote branch, you will probably need to **force push** to update your remote branch. Force push (`git push --force`) replaces the remote git history with the local one.

Dos and Don'ts:

- **DON'T** force push to the default branch
- **DON'T** force push a branch that others are collaborating on
- **DO** use `git push --force-with-lease` to prevent overwriting commits that are not present in your locally fetched copy of the remote branch

Activity: Accidentally committed a Secret!

In all the chaos of party planning, you realized that when adding the party invitation templates to your repo, you forgot to remove personal information, like your address and phone number! How do we get that information off of the internet before it falls into the wrong hands?

Never fear… it's interactive rebase to the rescue!

.gitignore

You can add a .gitignore file to exclude files from your git history. This can be used to avoid committing files including secrets, generated files like compiled binaries and reports from testing, and dependencies (e.g. node_modules folder). A .gitignore file consists of a list of glob patterns (one on each line) for files to ignore:

```
env
venv
*.pyc
.python-version
dist/
```