# WEEK 8:

## GIT
## 1. Git-HOL

STEP 1: Git Configuration

```
# Check if Git is installed
git --version
Output:
   git version 2.41.0.windows.1

# Set your Git user name and email
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"

# View global Git configuration
git config --list
Output:
   user.name=Your Name
   user.email=your.email@example.com
```

STEP 2: Set Notepad++ as Editor

```
notepad++
Output:
   (Notepad++ application should open)

# Optional alias for Notepad++
alias notepad="'/c/Program Files/Notepad++/notepad++.exe'"

# Set Notepad++ as default Git editor
git config --global core.editor "'/c/Program Files/Notepad++/notepad++.exe' -multiInst -
nosession"

# Verify the editor setting
git config --global -e
Output:
   (Opens configuration in Notepad++ with editor setting)
```

STEP 3: Create Repository and Track File

```
mkdir GitDemo
cd GitDemo
git init
Output:
    Initialized empty Git repository in C:/Users/YourUser/GitDemo/.git/

ls -a
Output:
    .  ..  .git

# Create a file and add content
echo "Hello, Git World!" > welcome.txt

# Check that file exists
ls
Output:
    welcome.txt

# View content inside file
cat welcome.txt
Output:
    Hello, Git World!

# Check Git status
git status
Output:
    Untracked files:
      (use "git add <file>..." to include in what will be committed)
        welcome.txt

# Add the file to staging area
git add welcome.txt

# Commit file (will open Notepad++)
git commit
Output:
    [master (root-commit) abc1234] Initial commit of welcome.txt
     1 file changed, 1 insertion(+)
     create mode 100644 welcome.txt

# Check status again
git status
Output:
    On branch master
    nothing to commit, working tree clean
```

STEP 4: Push to Remote Repository (Optional)

```
# Add remote repo (replace with your GitLab URL)
git remote add origin https://gitlab.com/yourusername/GitDemo.git
```

```
# Push changes to remote
git push origin master
Output:
    Enumerating objects: 3, done.
    Counting objects: 100% (3/3), done.
    Writing objects: 100% (3/3), done.
    Total 3 (delta 0), reused 0 (delta 0)
    To https://gitlab.com/yourusername/GitDemo.git
     * [new branch]      master -> master
```

## 2. Git-HOL

### Explain `.gitignore`

1. `.gitignore` is a special file that tells Git which files or folders to ignore (not track or commit).
2. It helps prevent unwanted files like logs, temp files, and system/IDE configs from entering the repository.

### • Explain how to ignore unwanted files using `.gitignore`

1. Add the file or folder patterns to the `.gitignore` file (e.g., `*.log`, `node_modules/`).
2. Save and commit the `.gitignore` file so Git skips those specified files during staging and committing.

#### Step 1: Initialize Git Repository

- Open Git Bash, create a folder, and initialize it using `git init`.
  **Expected Output:**
  `Initialized empty Git repository in C:/Users/jhansi/GitIgnoreDemo/.git/`

#### Step 2: Create Unwanted Files

- Create a file named `error.log` and a folder named `log`, then add sample content inside both.
  **Expected Output:**
  (No visible output — files/folders created in the working directory)

#### Step 3: Check Git Status Before Ignoring

- Use `git status` to see which files are being tracked.
  **Expected Output:**

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        error.log
        log/
```

**Step 4: Create and Update `.gitignore`**

- Create a file named `.gitignore`. Inside it, add:
  `*.log` — to ignore all `.log` files
  `log/` — to ignore the entire log folder
  **Expected Output:**
  (No output — `.gitignore` is updated successfully)

**Step 5: Check Git Status After Ignoring**

- Run `git status` again to verify if `error.log` and `log/` are now ignored.
  **Expected Output:**

```
Untracked files:
        .gitignore
```

Git is now ignoring `.log` files and the `log/` folder.

**Step 6: Add and Commit `.gitignore`**

- Add the `.gitignore` file using `git add` and commit it with a message.
  **Expected Output:**

```
 [master (root-commit) abc1234] Add .gitignore to ignore .log files and log
folder
 1 file changed, 2 insertions(+)
 create mode 100644 .gitignore
```

**Step 7: Final Git Status Check**

- Run `git status` one last time to confirm everything is clean.
  **Expected Output:**

```
On branch master
nothing to commit, working tree clean
```

**Final Result:**

- The `.gitignore` file is working correctly.
- Git is no longer tracking `error.log` and the `log/` folder.
- Only necessary files are committed.

- ## 3. Git-HOL

**OBJECTIVES:**

### • Explain branching and merging

1. **Branching** in Git allows you to create a separate version of the codebase to work on new features or fixes without affecting the main project.
2. **Merging** combines changes from one branch (e.g., feature branch) into another (usually the main branch), keeping the project up to date with all contributions.

### • Explain about creating a branch request in GitLab

1. In GitLab, to create a **branch request**, you first create a **new branch** from the repository (usually from the main or master branch).
2. You push your changes to this branch and then use it to initiate a **merge request** for code review and approval.

### • Explain about creating a merge request in GitLab

1. A **merge request** in GitLab is a way to request that your code from a feature branch be merged into another branch (like `main`).
2. It allows team members to review changes, leave comments, and approve or reject the request before merging into the main project.

### Step 1: Create and Work on a New Branch

```
git branch GitNewBranch
git branch
```

**Expected Output:**

```
 master
  GitNewBranch
git checkout GitNewBranch
```

**Expected Output:**

```
Switched to branch 'GitNewBranch'
echo "New content" > feature.txt
git add feature.txt
git commit -m "Add feature.txt in GitNewBranch"
```

**Expected Output:**

```
 [GitNewBranch abc1234] Add feature.txt in GitNewBranch
 1 file changed, 1 insertion(+)
 create mode 100644 feature.txt
git status
```

**Expected Output:**

```
On branch GitNewBranch
nothing to commit, working tree clean


git checkout master
```

**Expected Output:**

```
Switched to branch 'master'
git diff GitNewBranch
```

**Expected Output:**
Shows differences between `master` and `GitNewBranch` in CLI.

```
git difftool GitNewBranch
```

**Expected Output:**

Launches **P4Merge** tool to show visual differences.

```
git merge GitNewBranch
```

**Expected Output:**

```
Updating abc1234..def5678
Fast-forward
 feature.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 feature.txt
git log --oneline --graph --decorate
```

**Expected Output:**
Shows a graph-style commit history with branch merges.

## Step 3: Delete the Merged Branch

```
git branch -d GitNewBranch
```

## Expected Output:

```
Deleted branch GitNewBranch (was def5678).
bash
CopyEdit
git status
```

## Expected Output:

```
On branch master
nothing to commit, working tree clean
```

## 4. Git-HOL

### Step 1: Start with a Clean Master

```
git checkout master
git status
```

**Expected Output:**

```
On branch master
nothing to commit, working tree clean
```

### Step 2: Create and Work in a New Branch

```
git checkout -b GitWork
echo "<message>Hello from branch</message>" > hello.xml
git add hello.xml
git commit -m "Add hello.xml in GitWork"
```

**Expected Output:**

```
 [GitWork abc1234] Add hello.xml in GitWork
 1 file changed, 1 insertion(+)
 create mode 100644 hello.xml
```

### Step 3: Switch to Master and Create Conflict

```
git checkout master
echo "<message>Hello from master</message>" > hello.xml
git add hello.xml
git commit -m "Add hello.xml in master with different content"
```

**Expected Output:**

```
 [master def5678] Add hello.xml in master with different content
 1 file changed, 1 insertion(+)
 create mode 100644 hello.xml
```

### Step 4: View History

```
git log --oneline --graph --decorate --all
```

**Expected Output:**
Graph showing both `master` and `GitWork` diverged.

**Step 5: Check Differences**

```
git diff GitWork
```

**Expected Output:**
Shows text differences between files in master and GitWork.

```
git difftool GitWork
```

**Expected Output:**
Launches **P4Merge** to show visual differences.

## Step 6: Merge Branch into Master

```
git merge GitWork
```

**Expected Output (conflict):**

```
Auto-merging hello.xml
CONFLICT (add/add): Merge conflict in hello.xml
Automatic merge failed; fix conflicts and then commit the result.
```

## Step 7: Resolve Conflict

Open `hello.xml`, and you'll see:

```
<<<<<<< HEAD
<message>Hello from master</message>
=======
<message>Hello from branch</message>
>>>>>>> GitWork
```

Manually edit to keep desired content, e.g.:

```
<message>Hello from both sides</message>
```

## Step 8: Mark Conflict as Resolved and Commit

```
git add hello.xml
git commit -m "Resolved merge conflict in hello.xml"
```

**Expected Output:**

```
 [master ghi9012] Resolved merge conflict in hello.xml
```

## Step 9: Ignore Backup Files and Commit

```
echo "*.orig" >> .gitignore
git add .gitignore
git commit -m "Add .gitignore to exclude backup files"
```

**Expected Output:**

```
 [master jkl3456] Add .gitignore to exclude backup files
```

## Step 10: Delete Merged Branch

```
git branch -d GitWork
```

**Expected Output:**

```
Deleted branch GitWork (was abc1234).
```

## Step 11: Final Log View

```
git log --oneline --graph --decorate
```

**Expected Output:**
Graph view showing the successful merge of `GitWork` into `master`.

## 5. Git-HOL
## Objectives:

### Explain how to clean up and push back to remote Git

1. Cleaning up in Git usually means ensuring your local branch (e.g., master) has no uncommitted or conflicting changes before syncing with the remote repository.
2. Pushing back to remote involves uploading your committed changes to the remote Git repository so others can access and use them.

### Step 1: Verify Clean Working Directory

```
git status
```

**Expected Output:**

```
On branch master
nothing to commit, working tree clean
```

### Step 2: List All Available Branches

```
git branch -a
```

**Expected Output:**

```
* master
  remotes/origin/master
  remotes/origin/Git-T03-HOL_002
```

### Step 3: Pull Remote Repository into Local Master

```
git pull origin master
```

**Expected Output:**

```
Already up to date.
```

### Step 4: Push Local Changes to Remote

```
git push origin master
```

**Expected Output:**

```
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Writing objects: 100% (3/3), done.
```

```
To https://github.com/yourusername/repo-name.git
   abc1234..def5678  master -> master
```

## Final Result:

- Local and remote `master` branches are in sync.
- All updates from the previous hands-on are successfully pushed to the remote repository.