Git Training Course: Interactive Exercises

This document contains both in-class exercises and take-home challenges to reinforce Git concepts and build practical skills.

In-Class Exercises

Exercise 1: Git Setup and First Commits (15 minutes)

Objective: Establish a Git repository and make initial commits.

Steps:

1. **Setup Git configuration:**

```
git config --global user.name "Your Name"
git config --global user.email "your.email@businessandtrade.gov.uk"
```

2. **Create a new repository:**

```
mkdir git-practice
cd git-practice
git init
```

3. **Create and add files:**

```
# Create an HTML file
echo "<h1>My Git Project</h1>" > index.html

# Create a CSS file
echo "body { font-family: Arial, sans-serif; }" > styles.css

# Create a JavaScript file
echo "console.log('Hello Git!');" > script.js
```

4. **Check status and stage files:**

```
git status
git add index.html
git status
```

5. **Make your first commit:**

```
git commit -m "Add index.html file"
```

6. **Stage and commit remaining files:**

```
git add styles.css script.js
git commit -m "Add CSS and JavaScript files"
```

7. **View commit history:**

```
git log
git log --oneline
```

Discussion Questions:

- What information does Git store with each commit?
- How does the staging area help in organising commits?
- What happens if you modify a file after staging but before committing?

Exercise 2: Branching and Merging (20 minutes)

Objective: Practice creating branches, making changes, and merging.

Steps:

1. **Create and switch to a feature branch:**

```
git branch feature-navbar
git checkout feature-navbar
# OR using newer Git syntax:
git switch -c feature-navbar
```

2. **Make changes on the feature branch:**

```
# Modify the HTML file
echo "<nav>Home | About | Contact</nav>" >> index.html

# Modify the CSS file
echo "nav { background-color: #333; color: white; }" >> styles.css
```

3. **Commit changes on the feature branch:**

```
git add index.html styles.css
git commit -m "Add navigation bar"
```

4. **Switch back to main branch:**

```
git checkout main
# OR
git switch main
```

5. **Make different changes on main:**

```
echo "<footer>&copy; 2025 Git Training</footer>" >> index.html
git add index.html
git commit -m "Add footer to main page"
```

6. **Merge feature branch into main:**

```
git merge feature-navbar
```

7. **Resolve any merge conflicts:** If conflicts occur:
   - Open the conflicted files in an editor
   - Look for conflict markers («««, =======, »»»>)

- Edit files to resolve conflicts
- Stage resolved files with `git add`
- Complete the merge with `git commit`

8. **View the commit graph:**

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

Discussion Questions:

- How does Git determine if a merge can be done automatically?
- What triggers a merge conflict?
- What strategies can prevent merge conflicts?

Exercise 3: Git Workflow Simulation (20 minutes)

Objective: Simulate a team workflow using either Trunk-Based or Git Flow.

Option A: Trunk-Based Development

1. **Create short-lived feature branches:**

```
git checkout -b feature-header main
# Make and commit header changes
git checkout main
git merge feature-header
git branch -d feature-header

git checkout -b feature-content main
# Make and commit content changes
git checkout main
git merge feature-content
git branch -d feature-content
```

Option B: Git Flow

1. **Set up branch structure:**

```
# Create develop branch
git checkout -b develop main

# Create feature branch from develop
git checkout -b feature/login develop

# Make and commit feature changes
# ...

# Complete feature
git checkout develop
```

```
git merge feature/login
git branch -d feature/login

# Create release branch
git checkout -b release/1.0 develop

# Finalise release
git checkout main
git merge release/1.0
git tag -a v1.0 -m "Version 1.0"
git checkout develop
git merge release/1.0
git branch -d release/1.0
```

Discussion Questions:

- Which workflow felt more natural for this exercise?
- What challenges did you encounter with your chosen workflow?
- How would this workflow scale with more team members?

Exercise 4: Git Problem-Solving (25 minutes)

Objective: Practice fixing common Git issues.

Scenario 1: Divergent Histories

1. **Create the scenario:**

```
# Create a remote-simulation branch
git checkout -b remote-simulation
echo "Remote change" >> README.md
git add README.md
git commit -m "Change on remote"

# Make local changes
git checkout main
echo "Local change" >> README.md
git add README.md
git commit -m "Local change"
```

2. **Fix with merge:**

```
git merge remote-simulation
# Resolve any conflicts
```

3. **Reset and fix with rebase:**

```
git reset --hard HEAD~1
echo "Local change" >> README.md
```

```
git add README.md
git commit -m "Local change"
git rebase remote-simulation
# Resolve any conflicts
```

Scenario 2: Detached HEAD

1. **Create the scenario:**

```
# Get a specific commit hash
git log --oneline
# Checkout that commit directly
git checkout <commit-hash>
```

2. **Make changes in detached HEAD:**

```
echo "Detached HEAD change" > detached.txt
git add detached.txt
git commit -m "Change in detached HEAD"
```

3. **Recover the changes:**

```
# Create a branch at current position
git branch recover-detached

# Return to main
git checkout main

# Merge or cherry-pick the changes
git merge recover-detached
# OR
git cherry-pick recover-detached
```

Scenario 3: Accidental Commit

1. **Create the scenario:**

```
echo "password: secret123" > config.txt
git add config.txt
git commit -m "Add configuration"
```

2. **Fix the issue:**

```
# Option 1: Remove the file but keep in history
git rm --cached config.txt
echo "config.txt" >> .gitignore
git add .gitignore
git commit -m "Remove sensitive file and add to gitignore"

# Option 2: Rewrite history (CAUTION: only for unpushed commits)
git reset --soft HEAD~1
git rm --cached config.txt
```

```
echo "config.txt" >> .gitignore
git add .gitignore
git commit -m "Add proper configuration without sensitive data"
```

Discussion Questions:

- When is it appropriate to use rebase vs. merge?
- What are the risks of rewriting history with commands like reset?
- How can you recover from seemingly "lost" commits?

Take-Home Exercises

Basic Exercise: Repository Exploration

Objective: Build comfort with Git repository operations and history exploration.

Exercise:

1. **Create a structured project:**

```
mkdir git-project
cd git-project
git init

# Create directory structure
mkdir -p src/{js,css,img} docs tests

# Add some files
touch src/js/{app.js,utils.js}
touch src/css/styles.css
touch src/index.html
touch README.md
```

2. **Create a proper .gitignore file:**
   - Research appropriate patterns for your development environment
   - Include patterns for common build artifacts, dependencies, and IDE files
   - Test the .gitignore by creating some files that should be ignored
3. **Make a series of meaningful commits:**
   - Add README with project description
   - Add HTML skeleton
   - Add basic styling
   - Add JavaScript functionality
   - Ensure each commit has a clear, descriptive message
4. **Explore the repository history:**

```
# View basic history
git log

# View history with changed files
git log --stat

# View history with patch information
git log -p

# View history as a graph
git log --graph --oneline --all
```

5. **Experiment with history exploration:**

```
# Find commits that modified a specific file
git log -- src/js/app.js

# Search commits by message content
git log --grep="README"

# Search commits by code change
git log -S"function"

# View changes between commits
git diff HEAD~2 HEAD
```

Questions to Answer:

1. What naming patterns make commit messages most useful when reviewing history?
2. How would you find when a specific line of code was introduced?
3. What differences do you notice between viewing history with `--stat` vs `-p`?

Intermediate Exercise: Collaborative Workflow Simulation

Objective: Practice Git collaboration patterns independently.

Exercise:

1. **Create a "central" repository:**

```
mkdir central-repo.git
cd central-repo.git
git init --bare
cd ..
```

2. **Create two developer repositories:**

```
# Developer 1
git clone central-repo.git dev1-repo
cd dev1-repo
# Add initial files and push
touch README.md
git add README.md
git commit -m "Initial commit"
git push origin main
cd ..

# Developer 2
git clone central-repo.git dev2-repo
cd dev2-repo
# Update files
git pull
cd ..
```

3. **Simulate parallel development:**
   - In dev1-repo: Create a feature branch, make changes
   - In dev2-repo: Make changes to main
   - In dev1-repo: Try to merge feature to main and push
   - In dev2-repo: Push changes to main
   - Resolve the resulting conflicts and synchronisation issues
4. **Implement a chosen workflow:**
   - If using Trunk-Based: Create short-lived feature branches and merge frequently
   - If using Git Flow: Establish develop branch, feature branches, and simulate a release
5. **Document the entire process:**
   - Keep a log of commands used
   - Note any issues encountered
   - Describe how you resolved problems
   - Reflect on the workflow effectiveness

Questions to Answer:

1. What communication would have been necessary in a real team setting?
2. What Git hooks might help enforce your chosen workflow?
3. How would you onboard a new team member to this workflow?

Advanced Exercise: Git Internals Deep Dive

Objective: Understand Git's internal object model through direct exploration.

Exercise:

1. **Create a repository for exploration:**

```
mkdir git-internals
cd git-internals
git init
```

2. **Examine the .git directory structure:**

```
find .git -type f | sort
```

3. **Create content and explore object creation:**

```
echo "Hello, Git internals!" > file.txt
git add file.txt

# Find the object hash
git hash-object file.txt

# Examine the object
find .git/objects -type f
```

4. **Use low-level Git commands:**

```
# Explore a blob
git cat-file -p <hash>

# Make a commit
git commit -m "Add file.txt"

# Explore the commit
git cat-file -p HEAD

# Explore the tree
tree_hash=$(git cat-file -p HEAD | grep tree | cut -d' ' -f2)
git cat-file -p $tree_hash
```

5. **Create a commit without high-level commands:**

```
# Create a new blob
echo "Manual commit content" > manual.txt
blob_hash=$(git hash-object -w manual.txt)

# Create a tree
echo "100644 blob $blob_hash\tmanual.txt" | git mktree

# Create a commit
commit_hash=$(echo "Manual commit message" | git commit-tree <tree-hash> -p HEAD)
```

```
# Update HEAD
git update-ref HEAD $commit_hash
```

6. **Map object relationships:**
   - Create a visual diagram showing the relationships between:
     - Blobs
     - Trees
     - Commits
     - Branches (refs)
   - Show how they connect in your repository

Questions to Answer:

1. How does Git's content-addressed storage ensure data integrity?
2. How do branches differ from commits in Git's object model?
3. What happens internally when you run common Git commands like `add`, `commit`, and `branch`?

Advanced Exercise: Rebasing Workshop

Objective: Master Git's history rewriting capabilities.

Exercise:

1. **Create a repository with a messy history:**

```
mkdir rebase-workshop
cd rebase-workshop
git init

# Create some files with messy commit history
echo "Initial content" > file.txt
git add file.txt
git commit -m "Initial commit"

echo "More content" >> file.txt
git commit -am "Add more"

echo "Fix typo" >> file.txt
git commit -am "Fix typo"

echo "WIP stuff" >> file.txt
git commit -am "WIP"

echo "More WIP" >> file.txt
git commit -am "WIP continued"
```

```
echo "Finished feature" >> file.txt
git commit -am "Feature complete"
```

2. **Clean up the history with interactive rebase:**

```
git rebase -i HEAD~5
```

   - Squash the "Fix typo" commit into "Add more"
   - Combine all WIP commits into a single "Development progress" commit
   - Reword "Feature complete" to be more descriptive
   - Reorder commits if it makes logical sense

3. **Split a commit:**

```
# Create a commit with multiple logical changes
echo "Feature A" > featureA.txt
echo "Feature B" > featureB.txt
git add featureA.txt featureB.txt
git commit -m "Add features A and B"

# Use interactive rebase to edit this commit
git rebase -i HEAD~1
# Change "pick" to "edit" for the commit
# When rebase stops:
git reset HEAD^
git add featureA.txt
git commit -m "Add feature A"
git add featureB.txt
git commit -m "Add feature B"
git rebase --continue
```

4. **Fix up earlier commits:**

```
# Make a change that belongs in an earlier commit
echo "Missing part of feature A" >> featureA.txt

# Use interactive rebase with autosquash
git add featureA.txt
git commit --fixup <feature-A-commit-hash>
git rebase -i --autosquash <before-feature-A-commit-hash>
```

5. **Cherry-pick between branches:**

```
# Create and switch to an experimental branch
git checkout -b experimental

# Make some experimental commits
echo "Experiment 1" > exp1.txt
git add exp1.txt
```

```
git commit -m "Experiment 1"

echo "Experiment 2" > exp2.txt
git add exp2.txt
git commit -m "Experiment 2"

# Return to main branch
git checkout main

# Cherry-pick only the good experiment
git cherry-pick <experiment-1-hash>
```

Questions to Answer:

1. What are the ethical considerations of rewriting history in a shared repository?
2. When is interactive rebasing preferable to merge commits?
3. How would you communicate history rewrites to teammates?

Challenge Exercise: Git Bisect Bug Hunt

Objective: Use Git's bisect feature to find when a bug was introduced.

Exercise:

1. **Create a repository with a regression:**

```
mkdir bisect-practice
cd bisect-practice
git init

# Create a simple functioning program
echo 'function add(a, b) { return a + b; }' > math.js
echo 'console.log(add(5, 3));' > main.js
git add math.js main.js
git commit -m "Initial math functions"

# Add more functions
echo 'function subtract(a, b) { return a - b; }' >> math.js
git commit -am "Add subtract function"

echo 'function multiply(a, b) { return a * b; }' >> math.js
git commit -am "Add multiply function"

# Introduce a bug in add function
sed -i 's/return a + b/return a - b/' math.js
echo 'function divide(a, b) { return a / b; }' >> math.js
```

```
git commit -am "Add divide function"

echo 'function square(a) { return a * a; }' >> math.js
git commit -am "Add square function"

echo 'function cube(a) { return a * a * a; }' >> math.js
git commit -am "Add cube function"
```

2. **Discover the "bug":**
   - The add function now returns incorrect results
   - You know it used to work, but not when it broke
3. **Use git bisect to find the bug:**

```
# Start bisect
git bisect start

# Mark current version as bad
git bisect bad

# Mark a known good version
git bisect good <initial-commit-hash>

# For each version Git checks out, test the add function
# Mark as good or bad accordingly
node -e "const {add} = require('./math.js'); console.log(add(5, 3) === 8 ? 'GOOD' : 'BA
git bisect good  # or git bisect bad

# Continue until Git identifies the first bad commit
```

4. **Fix the bug and verify:**

```
# After finding the bad commit, fix the bug
sed -i 's/return a - b/return a + b/' math.js
git commit -am "Fix addition function"

# Verify the fix works
node -e "const {add} = require('./math.js'); console.log(add(5, 3))"
```

Questions to Answer:

1. How could automated tests help with the bisect process?
2. What strategies would you use for bisecting bugs in larger codebases?
3. How would you document the bug and fix for team knowledge sharing?