Git Commands and Scenario Solutions

1. Combining Changes from a Feature Branch into the Main Branch

To merge changes from a feature branch into the main branch:

1. First, switch to the main branch:

   ```
   git checkout main
   ```

2. Pull the latest changes to ensure the main branch is up-to-date with any remote modifications:

   ```
   git pull origin main
   ```

3. Merge the feature branch into the main branch:

   ```
   git merge feature-branch
   ```

4. **Conflict Resolution:** If you encounter a merge conflict, Git will pause the merge and mark the conflicting files. To resolve the conflict:

   - Open each conflicting file and manually review the conflict markers `<<<<<`, `=====`, and `>>>>>` to see the conflicting sections.
   - Edit the file to resolve the conflict, keeping both modifications or prioritizing the necessary changes.
   - Once resolved, add the resolved files back to the staging area:

     ```
     git add <file>
     ```

   - After resolving all conflicts, complete the merge:

     ```
     git commit
     ```

2. Comparison of Two Methods for Integrating Changes: Merge vs. Rebase

- **Merging:** This creates a merge commit that combines both branches' histories, maintaining a complete history of all commits in both branches.
  - **Use Case:** Merging is best when you want to preserve the individual history of the feature branch, especially if multiple developers are working on the feature.
- **Rebasing:** This involves moving or "replaying" commits from the feature branch onto the tip of the main branch, making it appear as though your work started from the latest main branch commit.
  - **Use Case:** Rebasing is ideal for keeping a clean, linear commit history, especially for personal branches. It's helpful if you're working alone on the feature branch and want to avoid merge commits.
- **Key Consideration:** Use rebase carefully, as it rewrites history. Avoid rebasing commits that have been pushed to a shared repository since it can cause conflicts for others working on the same branch.

3. Temporarily Saving Ongoing Modifications (Stashing)

- **Saving Changes with Stash:** Use `git stash` to temporarily save changes without committing them:

  ```
  git stash
  ```

- **Use Case:** This is useful when you need to switch branches quickly without finalizing your work, for instance, if you need to address an urgent bug in a different branch.
- **Restoring Stashed Changes:** To reapply the stashed changes, use:

  ```
  git stash apply
  ```

  If you've stashed multiple times and want to apply a specific stash, list them with `git stash list` and apply using:

  ```
  git stash apply stash@{n}
  ```

4. Undoing Changes: Selective, Partial, and Complete

- **Selective Undo (git restore –staged ¡file¿):** Removes specific changes from the staging area while keeping them in the working directory.
  - **Use Case:** Useful when you've accidentally staged files you don't want in your next commit.
- **Partial Undo (git restore –source ¡commit¿ – ¡file¿):** Allows you to selectively reset changes in a specific file or part of a file.
  - **Use Case:** Helpful if you only want to keep parts of a change, like reverting certain lines but not the entire file.
- **Complete Undo (git reset or git checkout):** Resets the working directory to a previous commit completely.
  - **Use Case:** Applicable if you want to revert all changes to a previous stable state.
- **Considerations:** Each undo method affects the working directory differently. Partial undos maintain recent changes selectively, which may be useful for minor edits. However, completely resetting should be done with caution, as it could result in losing uncommitted changes. Always check the status with `git status` to ensure you're only undoing what's intended.