```
package gitlet;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Objects;

/**Gitlet is a miniaturized version control system based upon the core functionalities (add, commit, checkout, merge, etc.) of Git
 * (Full repo unavailable for UC Berkeley privacy concerns)
 * This is the main class that performs all operations and holds all information about the Gitlet Repository.
 * @author Ansh Vashisth
 */
public class Repository {
    /** Creates a new .gitlet directory if one does not already
     * exist. */
    public Repository() {
        File repo = new File(".gitlet");
    }
    /** Initlizes a Git repository in the current directory. */
    public void init() {
        File f = new File(".gitlet/Head.txt");
        if (f.length() != 0) {
            System.out.println("A Gitlet version-control system "
                    + "already exists in the current directory.");
        } else {
            new File(".gitlet/blobs").mkdirs();
            new File(".gitlet/branches").mkdirs();
            new File(".gitlet/commits").mkdirs();

            new File(".gitlet/Head.txt");
            new File(".gitlet/Staging.txt");
            new File(".gitlet/Removed.txt");
            new File(".gitlet/branches/master.txt");
            new File(".gitlet/branches/HEAD.txt");

            _staging = new HashMap<>();
            _remove = new HashMap<>();
```

```
        _branchMap = new HashMap<>();

        _root = new Commit("initial commit", null, _staging, true);
        Utils.writeObject(new File(".gitlet/commits/"
                + _root.getHash()), _root);

        updateHead(_root.getHash());
        updateBranch(_root, "master");

        Utils.writeContents(new File(".gitlet/branches/HEAD.txt"),
                "master");
        Utils.writeContents(new File(".gitlet/branches/master"),
                    _root.getHash());

        _branchMap.put("master", _root);

        updateStagingTxt();
        updateRemoveTxt();
    }
}

/** Adds a file new File(file) to the
 * staging area.
 * @param file The file being added to staging. */
public void add(String file) {
    updateStagingHashMap();
    File addMe = new File(file);
    if (addMe.exists()) {
        String hashed = Utils.sha1((Object) Utils.readContents(addMe));

        if (getCurrentCommit().getBlobs().containsKey(file)) {
            if (Objects.equals(getCurrentCommit().getBlobs().get(file),
                    hashed)) {
                return;
            }
            _staging.remove(file);
        }

        _staging.put(addMe.toString(), hashed);
        updateStagingTxt();

        Utils.writeContents(new File(".gitlet/blobs/"
                    + hashed + ".txt"), Utils.readContents(addMe));
    } else {
```

```
            System.out.println("File does not exist.");
        }
    }

    /** Creates a new commit.
     *
     * @param msg The message for the new commit.
     */
    public void commit(String msg) {
        updateStagingHashMap();
        updateRemoveHashMap();
        updateBranchMap();

        if (_staging.isEmpty() & _remove.isEmpty()) {
            System.out.println("No changes added to the commit.");
            return;
        }
        if (msg.equals("")) {
            System.out.println("Please enter a commit message.");
            return;
        }

        HashMap newBlobs = (HashMap) getCurrentCommit().getBlobs().clone();
        newBlobs.putAll(getCurrentCommit().getBlobs());
        newBlobs.putAll(_staging);

        for (Map.Entry<String, String> next: _remove.entrySet()) {
            newBlobs.remove(next.getKey(), next.getValue());
        }

        Commit newCommit = new Commit(msg, getCurrentCommit(), newBlobs, false);
        Utils.writeObject(new File(".gitlet/commits/"
                + newCommit.getHash()), newCommit);

        updateHead(newCommit.getHash());

        _staging.clear();
        _remove.clear();
        updateStagingTxt();
        updateRemoveTxt();
    }

    /** Removes fileName from the staging area
     * and/or working directory, if it exists.
```

```
 * @param fileName The file to be removed.
 */
public void rm(String fileName) {
    updateStagingHashMap();
    updateRemoveHashMap();
    boolean found = false;
    if (getCurrentCommit().getBlobs().containsKey(fileName)) {
        _remove.put(fileName, getCurrentCommit().getBlobs().get(fileName));
        Utils.restrictedDelete(fileName);
        found = true;
    }
    if (_staging.containsKey(fileName)) {
        _staging.remove(fileName);
        found = true;
    }
    if (!found) {
        System.out.println("No reason to remove the file.");
    }
    updateStagingTxt();
    updateRemoveTxt();
}

/** Prints a log of all the commits in the
 * current branch. */
public void log() {
    Commit curr = getCurrentCommit();
    while (curr != null) {
        curr.print();
        curr = curr.getParent();
    }
}
/** Prints a log of all commits. */
public void globalLog() {
    for (String f: Objects.requireNonNull
            (Utils.plainFilenamesIn
                    (new File(".gitlet/commits")))) {
        Utils.readObject(Utils.join
                (new File(".gitlet/commits"),
                        f), Commit.class).print();
    }
}

/** Finds all commits with the given
 * message msg.
```

```
 * @param msg The message. */
public void find(String msg) {
    boolean exists = false;
    for (String f: Objects.requireNonNull
            (new File(".gitlet/commits").list())) {
        Commit curr = Utils.readObject(Utils.join
                (new File(".gitlet/commits"), f), Commit.class);
        if (Objects.equals(curr.getMsg(), msg)) {
            System.out.println(curr.getHash());
            exists = true;
        }
    }
    if (!exists) {
        System.out.println("Found no commit with that message.");
    }
}

/** Displays all branches, staged files,
 * and files that will be removed. */
public void status() {
    File f = new File(".gitlet/Head.txt");
    if (f.length() == 0) {
        System.out.println("Not in an initialized Gitlet directory.");
        return;
    }
    updateStagingHashMap();
    updateRemoveHashMap();
    updateBranchMap();
    List<String> branches = Utils.plainFilenamesIn(".gitlet/branches");
    System.out.println("=== Branches ===");
    String head = Utils.readContentsAsString
            (new File(".gitlet/branches/HEAD.txt"));
    System.out.println("*" + Utils.readContentsAsString
            (Utils.join(".gitlet/branches", "HEAD.txt")));
    for (String b: Objects.requireNonNull(branches)) {
        if (!Objects.equals(b, head) & (!Objects.equals(b, "HEAD.txt"))) {
            System.out.println(b);
        }
    }
    System.out.println();
    System.out.println("=== Staged Files ===");
    if (_staging != null) {
        for (Map.Entry<String, String> s: _staging.entrySet()) {
            System.out.println(s.getKey());
```

```
        }
    }
    System.out.println();
    System.out.println("=== Removed Files ===");
    if (_remove != null) {
        for (String r: _remove.keySet()) {
            System.out.println(r);
        }
    }
    System.out.println();
    System.out.println("=== Modifications Not Staged For Commit ===");
    System.out.println();
    System.out.println("=== Untracked Files ===");
}

/** Will return the directory to a previous state as detailed
 * by the arguments.
 * @param args Either 2, 3, or 4 arguments long, decides whether
 * checkout will deal with commits or branches.
 */
public void checkout(String...args) {
    updateStagingHashMap();
    updateRemoveHashMap();
    File workDir = new File(System.getProperty("user.dir"));
    switch (args.length) {
    case 2:
        checkoutBranch(args[1]);
        break;
    case 3:
        String fName = args[2];
        if (getCurrentCommit().getBlobs().get(fName) == null) {
            System.out.println("File nonexistent in commit.");
            return;
        } else if (Utils.join(workDir, fName).exists()) {
            Utils.restrictedDelete(Utils.join(workDir, fName));
        }
        File b = Utils.join(new File(".gitlet/blobs"),
                getCurrentCommit().getBlobs().get(fName) + ".txt");
        File newFile = new File(workDir.getPath(), fName);
        Utils.writeContents(newFile, Utils.readContents(b));
        break;
    case 4:
        String commit = args[1];
        String file = args[3];
```

```
        boolean found = false;
        for (String s: Objects.requireNonNull
                (Utils.plainFilenamesIn
                        (new File(".gitlet/commits")))) {
            if (s.toString().equals(commit)) {
                commit = s;
                found = true;
                break;
            }
        }
        Commit comm = Utils.readObject(new File(".gitlet/commits/"
                + commit), Commit.class);
        if (!found) {
            System.out.println("No commit with that ID exists.");
        } else if (!comm.getBlobs().containsKey(file)) {
            System.out.println("File does not exist in that commit.");
        } else {
            Utils.restrictedDelete(System.getProperty("user.dir") + file);
            File newBlob = new File(workDir
                    + ".gitlet/blobs/"
                    + comm.getBlobs().get(file) + ".txt");
            Utils.writeObject(new File(System.getProperty("user.dir")
                    + file), newBlob);

            File newF = new File(workDir.getPath() + file);
            Utils.writeContents(newF, Utils.readContents(newBlob));
        }
        break;
    default:
        System.out.println("Incorrect operands.");
    }
}

/** Making this because other checkout function was
 * too long.
 * @param args One argument, name of branch.
 */
public void checkoutBranch(String...args) {
    updateStagingHashMap();
    updateRemoveHashMap();
    File workDir = new File(System.getProperty("user.dir"));
    String bName = args[0];
    if (!Utils.join(new File(".gitlet/branches"), bName).exists()) {
        System.out.println("No such branch exists.");
```

```
        return;
    } else if (Objects.equals(bName, getCurrentBranch())) {
        System.out.println("No need to checkout the current branch.");
    }
    Commit curr = getCurrentCommit();
    String hash = Utils.readContentsAsString(Utils.join
            (new File(".gitlet/branches"), bName));
    Commit newCommit = Utils.readObject(Utils.join
            (new File(".gitlet/commits"), hash), Commit.class);
    for (File f: Objects.requireNonNull(workDir.listFiles())) {
        if (!curr.getBlobs().containsKey(f.getName())) {
            if (newCommit.getBlobs().containsKey(f.toString())) {
                System.out.println(
                        "There is an untracked file in the way; "
                                + "delete it or add it first.");
                return;
            }
        }
    }
    for (File f: Objects.requireNonNull(workDir.listFiles())) {
        if (curr.getBlobs().containsKey(f.getName())) {
            if (!newCommit.getBlobs().containsKey(f.toString())) {
                Utils.restrictedDelete(f);
            }
        }
    }
    for (String f: newCommit.getBlobs().keySet()) {
        File newF = Utils.join(new File(".gitlet/blobs"),
                newCommit.getBlobs().get(f));
        Utils.writeContents(new File(f), Utils.readContents(newF));
    }
    _staging.clear();
    updateStagingTxt();
    new File(".gitlet/branches/HEAD.txt").delete();
    new File(".gitlet/branches/HEAD.txt");
    writeString(bName, ".gitlet/branches/HEAD.txt");
}

/** Creates a new branch bName if it doesn't
 * already exist.
 * @param bName The new branch's name. */
public void branch(String bName) {
    updateBranchMap();
    if (_branchMap.containsKey(bName)) {
```

```
        System.out.println("A branch with that name already exists.");
    } else {
        String branch = Utils.readContentsAsString
                (Utils.join
                        (new File(".gitlet/branches"),
                                "HEAD.txt"));
        String hashed = Utils.readContentsAsString
                (Utils.join
                        (new File(".gitlet/branches"),
                                branch));
        writeString(hashed,
                Utils.join
                        (new File(".gitlet/branches"),
                                bName).toString());
    }
}

/** Removes branch bName if it exists.
 * @param bName The branch to be removed. */
public void rmBranch(String bName) {
    updateBranchMap();
    if (Objects.equals(bName, getCurrentBranch())) {
        System.out.print("Cannot remove the current branch.");
        return;
    } else if (_branchMap.containsKey(bName)) {
        Utils.join(new File(".gitlet/branches"), bName).delete();
        return;
    }
    System.out.println("A branch with that name does not exist.");
}

/** Resets the branch that has commit
 * commitID in it.
 * @param commitID The id of the commit
 * whose branch will be reset.*/
public void reset(String commitID) {

}

/** Merges the given branch txt into the
 * current branch.
 * @param txt The branch to be merged into
 * the given branch. */
public void merge(String txt) {
```

```
}

/** Returns the current Head Commit. */
public Commit getCurrentCommit() {
   String h = Utils.readContentsAsString(new File(".gitlet/Head.txt"));
   Commit n = Utils.readObject
         (new File(".gitlet/commits/" + h),
               Commit.class);
   return n;
}

/** Returns the current active branch. */
public String getCurrentBranch() {
   return Utils.readContentsAsString
         (new File(".gitlet/branches/HEAD.txt"));
}

/** "Refreshes" the staging text from the _staging HashMap. */
public void updateStagingTxt() {
   Utils.writeObject(new File(".gitlet/Staging.txt"), _staging);
}

/** "Refreshes" the staging hashmap from the Staging.txt File. */
public void updateStagingHashMap() {
   _staging = Utils.readObject(new File
   (".gitlet/Staging.txt"),
         HashMap.class);
}

/** Refreshes the remove file from the _remove HashMap. */
public void updateRemoveTxt() {
   Utils.writeObject(new File(".gitlet/Removed.txt"), _remove);
}

/** "Refreshes" the remove hashmap from the Remove.txt File. */
public void updateRemoveHashMap() {
   _remove = Utils.readObject(new File
   (".gitlet/Removed.txt"),
         HashMap.class);
}

/** Updates the Head Commit to be newHead.
 * @param newHead The new Head Commit. */
```

```
public void updateHead(String newHead) {
    File head = new File(".gitlet/Head.txt");
    try {
        PrintWriter pw;
        pw = new PrintWriter(head.getPath());
        pw.close();
        writeString(newHead, ".gitlet/Head.txt");
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}

/** Updates the Head Commit of the branch bName to be
 * commit.
 * @param commit The new Head Commit of the given branch.
 * @param bName The branch whose head is being updated. */
public void updateBranch(Commit commit, String bName) {
    if (Utils.join(new File(".gitlet/branches"), bName).exists()) {
        writeString(commit.getHash(), ".gitlet/branches/" + bName);
    }
}

/** "Refreshes" _branchMap from the list of
 * branches in .gitlet/branches. */
public void updateBranchMap() {
    _branchMap = new HashMap<>();
    List<String> fNames = Utils.plainFilenamesIn
            (new File(".gitlet/branches"));
    assert fNames != null;
    for (String s: Objects.requireNonNull(fNames)) {
        if (!Objects.equals(s, "HEAD.txt")) {
            String hash = Utils.readContentsAsString(Utils.join
                    (new File(".gitlet/branches"), s));
            _branchMap.put(s, Utils.readObject(Utils.join
                    (new File(".gitlet/commits"), hash),
                    Commit.class));
        }
    }
}

/** Returns the active branch. */
public String getHeadBranch() {
    return Utils.readContentsAsString(Utils.join
            (new File(".gitlet/branches"),
```

```
                    "HEAD.txt"));
    }

    /** Writes "tx" to new File(path).
     * @param tx The text that will be written.
     * @param path The path of the file to which tx
     * will be written. */
    public void writeString(String tx, String path) {
        try {
            FileWriter fw = new FileWriter(path, true);
            fw.write(tx);
            fw.close();
        } catch (IOException ioe) {
            System.err.println("IOException: " + ioe.getMessage());
        }
    }

    /** Blobs that are staged for addition. */
    private HashMap<String, String> _staging;
    /** Blobs that are staged for removal. */
    private HashMap<String, String> _remove;
    /** Contains mappings for every BranchName to Head Commit. */
    private HashMap<String, Commit> _branchMap;
    /** Holds the root commit of the entire Git Repository. */
    private Commit _root;
}
```