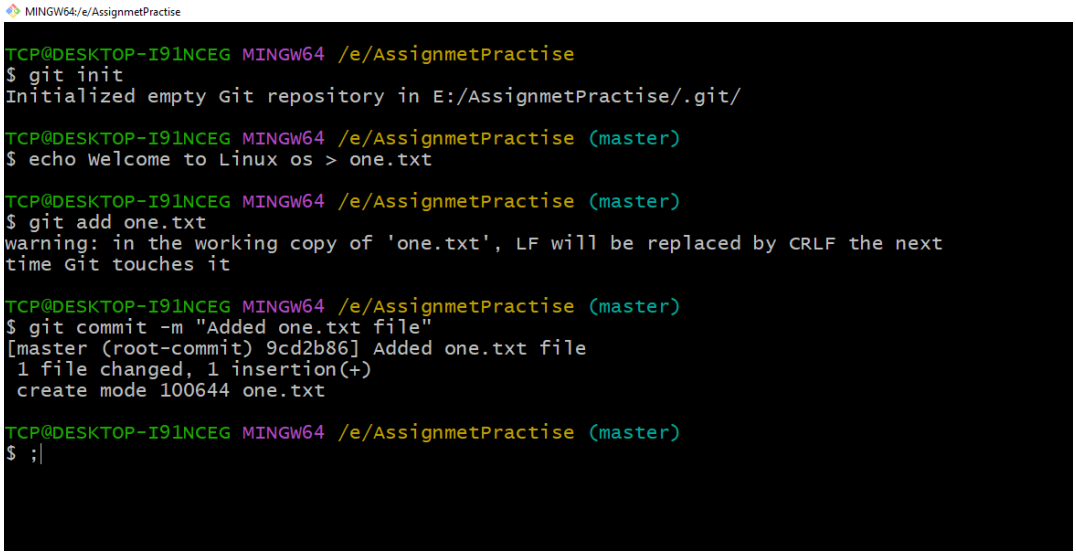
Assignment 1: Initialize a new Git repository in a directory of your choice. Add a simple text file to the repository and make the first commit.



Steps:

1. Open your terminal or command prompt.

2. Navigate to the directory where you want to create the Git repository. Use the cd command to change directories.

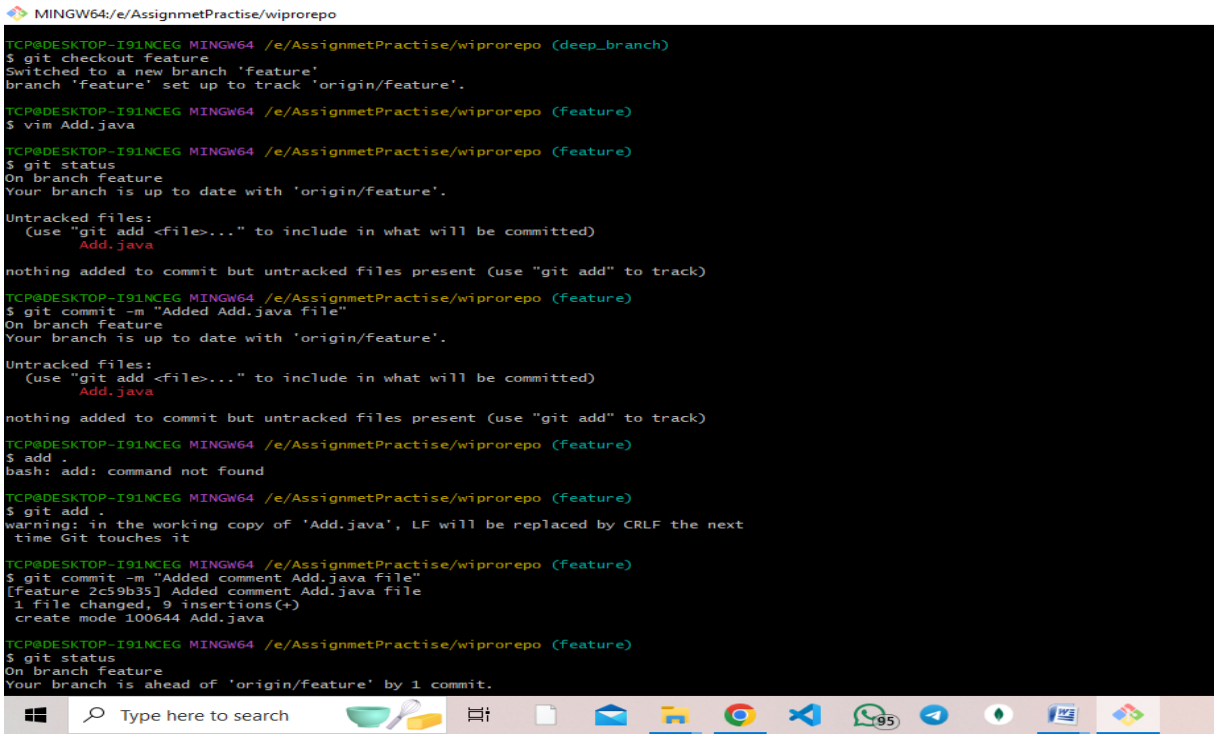
3. Initialize a new Git repository.

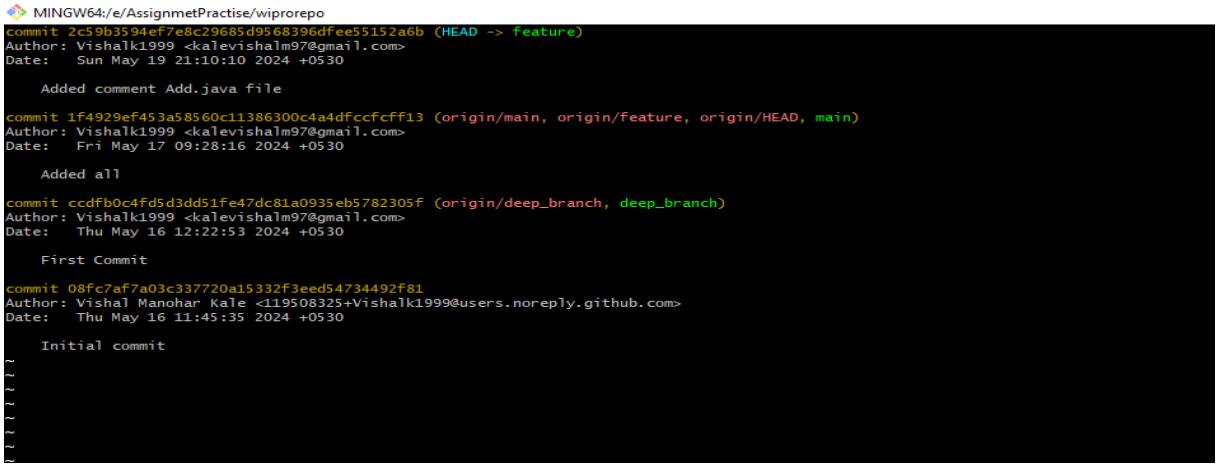
4.Create a simple text file one.txt

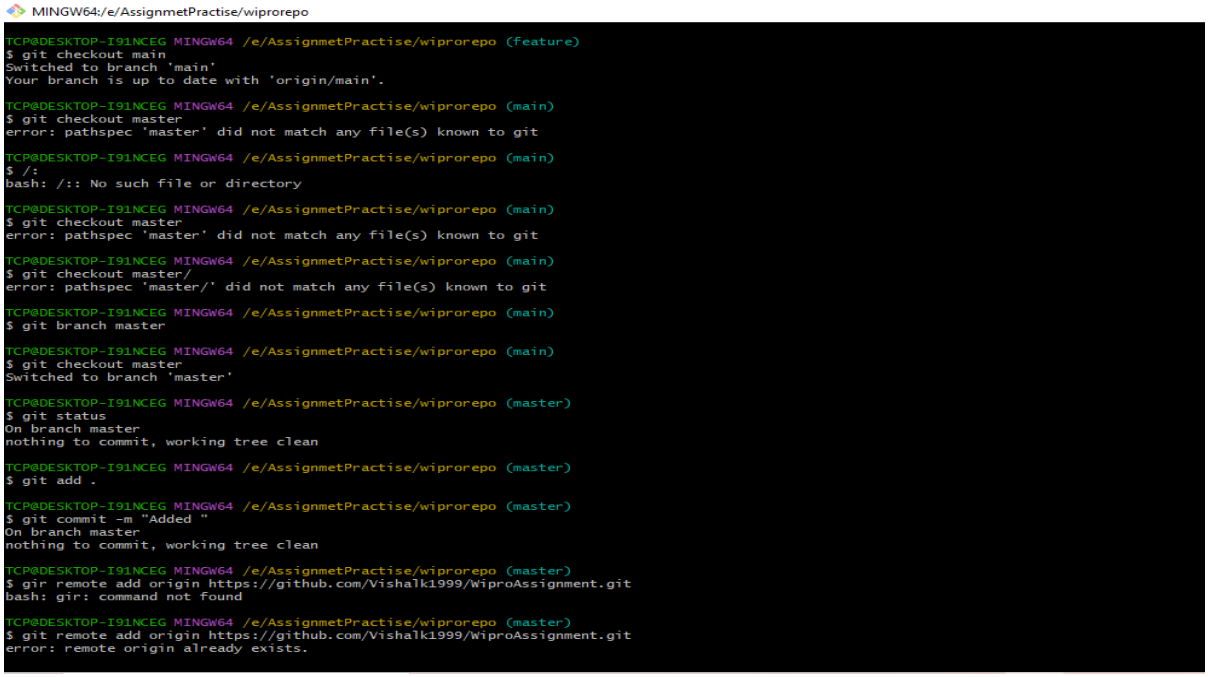
5.Add the file to the Git staging area. Use the git add command to tell Git that you want to track the newly created file:

6.Commit the changes using git -m commit “message”.

Assignment 2: Branch Creation and Switching Create a new branch named 'feature' and switch to it. Make changes in the 'feature' branch and commit them.







Here's how you can create a new branch named 'feature' and switch to it, make changes, and commit them in Git:

1. Create and Switch to Branch:

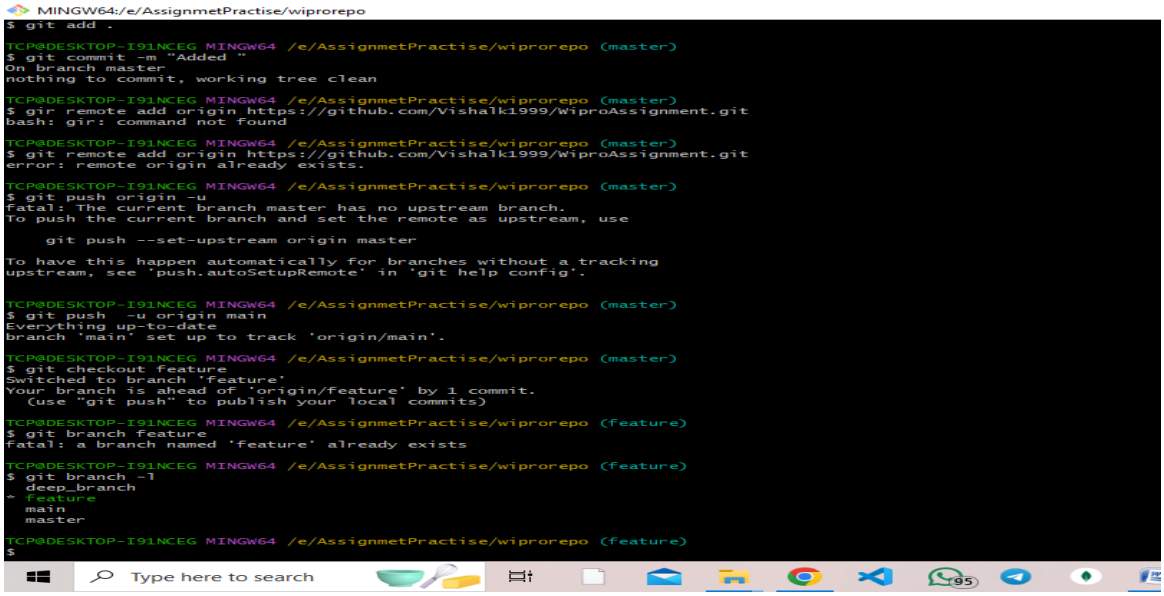
git checkout -b feature

This command combines two Git operations:

- git checkout: This tells Git to switch to a specific branch.
- -b: This flag tells git checkout to create a new branch if it doesn't already exist.

2. Make Changes:

- Now that you're in the 'feature' branch, make your desired changes to the files in your project.



3. Commit Changes:

Once you've made your changes, stage them using git add:

Replace <filename1> and <filename2> with the actual names of the files you modified. You can also use git add . to stage all modified files in the current directory.

Then, commit your staged changes with a descriptive message using git commit:

4.Verification:

- To verify that you're currently in the 'feature' branch, you can use the git branch command:

  Bash

  git branch

This will list all branches in your repository, with the current branch marked with an asterisk (*).

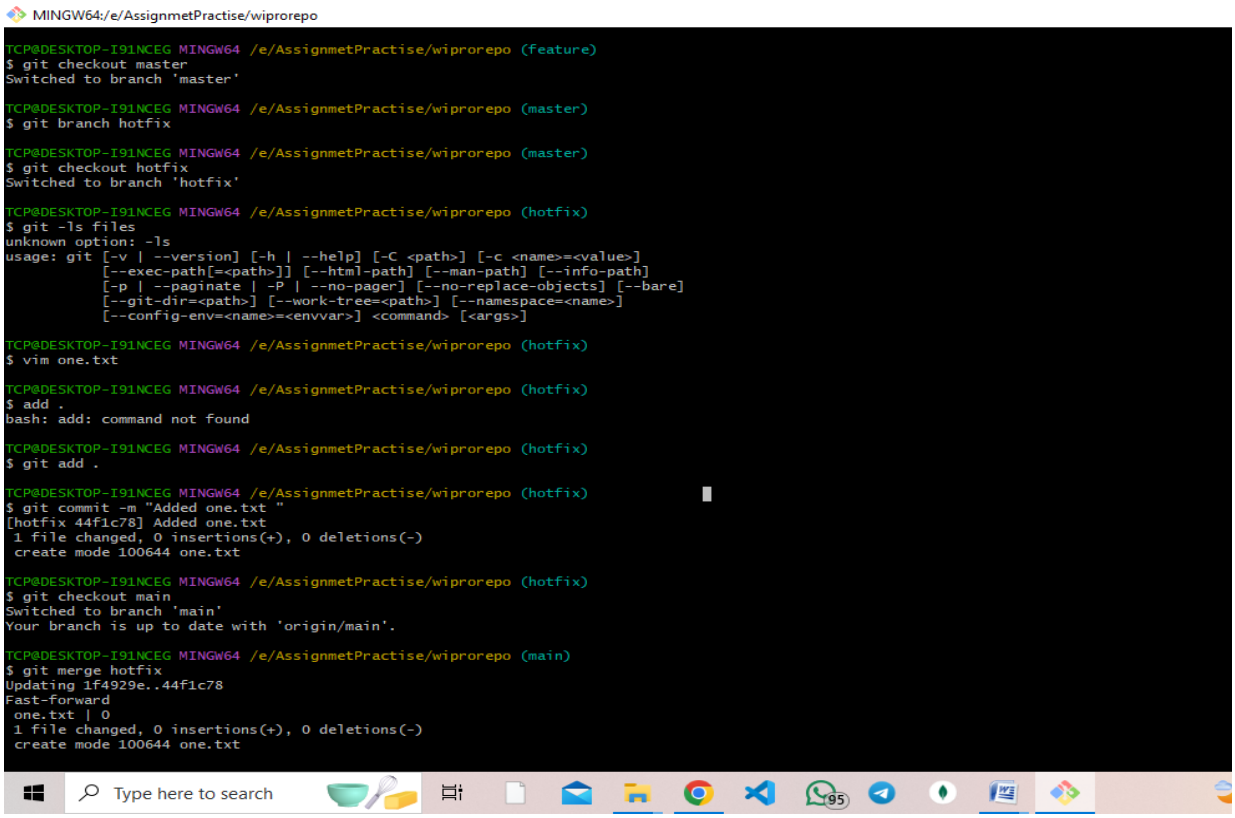
- To see the history of commits in the 'feature' branch, you can use git log:

  Bash

  git log feature

This will show you the commit messages and other details of the commits made in the 'feature' branch.

Assignment 3: Feature Branches and Hotfixes Create a 'hotfix' branch to fix an issue in the main code. Merge the 'hotfix' branch into 'main' ensuring that the issue is resolved.



1. Create a Hotfix Branch:

   Fix the Issue:

   Make the necessary changes to fix the issue in your code.

2. Commit the Changes:

This stages and commits the changes you made to the 'hotfix' branch.

3. Switch back to the Main Branch:

4. Merge Hotfix into Main:

This command merges the changes from the hotfix' branch into the 'main' branch. If there are no conflicts, Git will automatically perform a fast-forward merge.

5. Resolve Conflicts (if any):

If there are conflicts during the merge, you'll need to resolve them manually. Git will prompt you to resolve conflicts and commit the changes.

6. Push Changes:

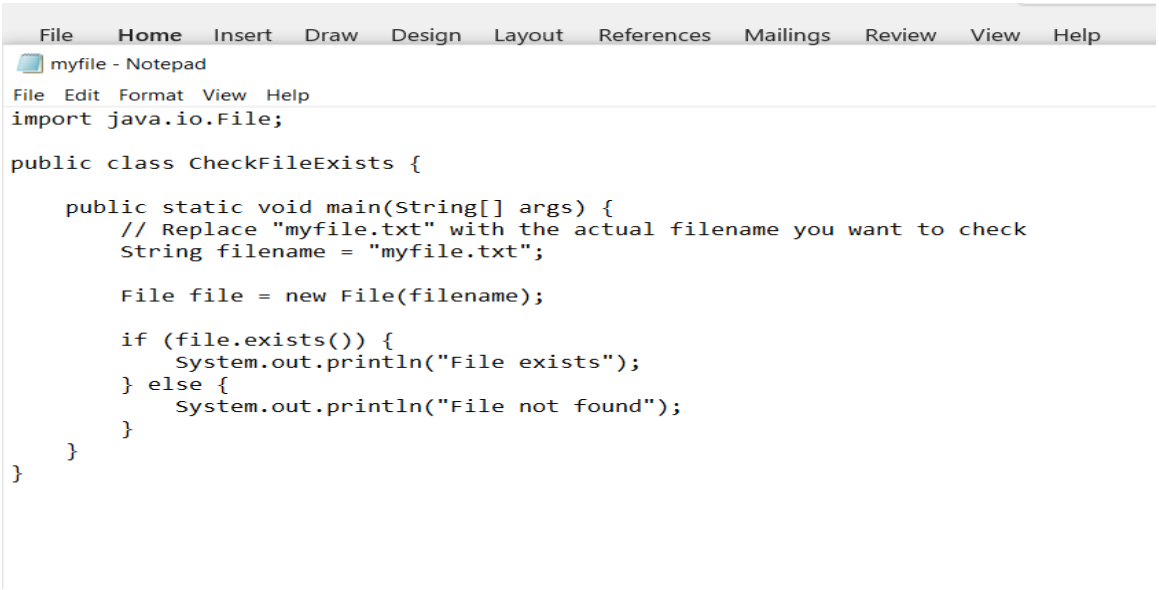
git push origin main

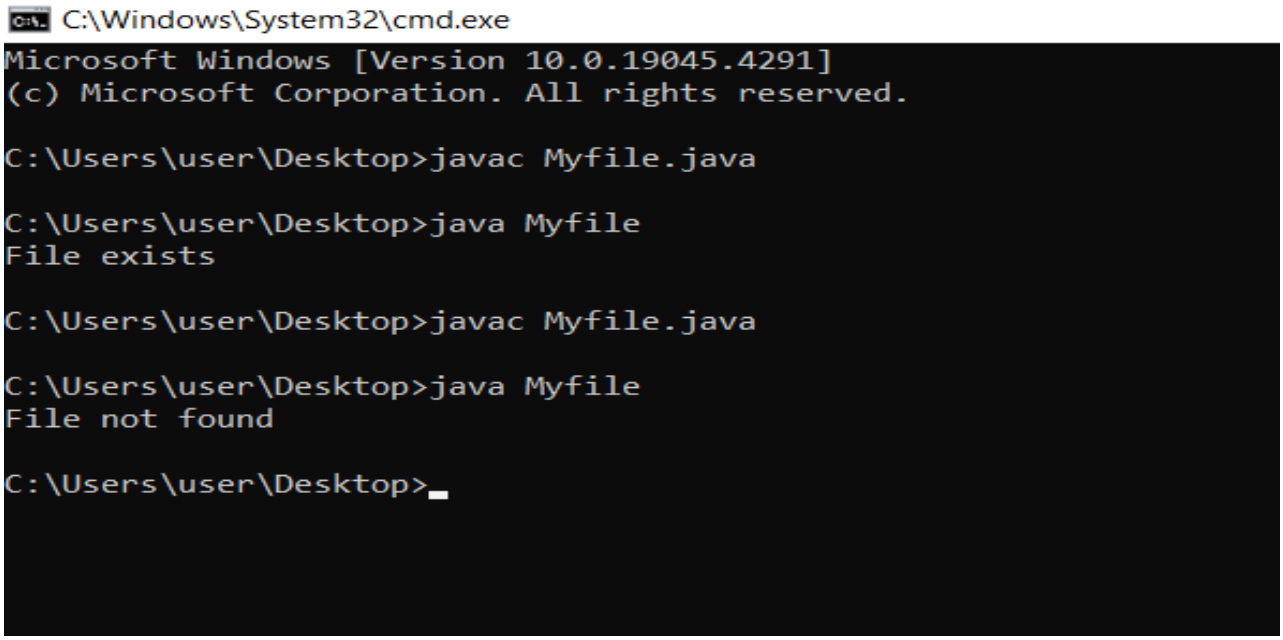
This command pushes the changes in the 'main' branch (including the hotfix) to the remote repository.

Assignment 1: Ensure the script checks if a specific file (e.g., myfile.txt) exists in the current directory. If it exists, print "File exists", otherwise print "File not found".

Java code that checks for the existence of a specific file in the current directory and prints appropriate messages:

```
import java.io.File;
public class Myfile
{
   public static void main(String[] args)
 {
      // Replace "myfile.txt" with the actual filename you want to check
      String filename = "myfile.txt";
      File file = new File(filename);
      if (file.exists())
 {
System.out.println("File exists");
      }
 else {
System.out.println("File not found");
      }
   }
}
```





Explanation:

1. Import: We import the java.io.File class, which provides functionalities for working with files and directories.
2. Filename: We declare a string variable filename and assign the name of the file you want to check (e.g., "myfile.txt"). Make sure to replace this with the actual filename.
3. Create File Object: We create a File object named file using the filename. This object represents the file on the system.
4. Check Existence: We call the exists() method on the file object. This method returns true if the file exists at the specified location, false otherwise.
5. Print Message:
    - If exists() returns true, we print "File exists" using System.out.println().
    - If exists() returns false, we print "File not found" using System.out.println().

Assignment 2: Write a script that reads numbers from the user until they enter '0'. The script should also print whether each number is odd or even.

Program:

CheckEvenOdd - Notepad

File Edit Format View Help

```
import java.util.Scanner;

public class CheckEvenOdd {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        while (true) {
            System.out.print("Enter a number (enter 0 to exit): ");
            int num = scanner.nextInt();

            if (num == 0) {
                System.out.println("Exiting the program...");
                break;
            } else {
                String result = (num % 2 == 0) ? "even" : "odd";
                System.out.println("The number " + num + " is " + result + ".");
            }
        }

        scanner.close();
    }
}
```

Output:

C:\Windows\System32\cmd.exe

```
Microsoft Windows [Version 10.0.19045.4291]
(c) Microsoft Corporation. All rights reserved.

C:\Users\user\Desktop>javac CheckEvenOdd.java

C:\Users\user\Desktop>java CheckEvenOdd
Enter a number (enter 0 to exit): 4
The number 4 is even.
Enter a number (enter 0 to exit): 5
The number 5 is odd.
Enter a number (enter 0 to exit): 6
The number 6 is even.
Enter a number (enter 0 to exit): 13
The number 13 is odd.
Enter a number (enter 0 to exit): 15
The number 15 is odd.
Enter a number (enter 0 to exit): 8
The number 8 is even.
Enter a number (enter 0 to exit): 9
The number 9 is odd.
Enter a number (enter 0 to exit): 56
The number 56 is even.
Enter a number (enter 0 to exit): 77
The number 77 is odd.
Enter a number (enter 0 to exit): 89
The number 89 is odd.
Enter a number (enter 0 to exit): 0
Exiting the program...

C:\Users\user\Desktop>
```

Explanation:

1.Import the Scanner class: This is used to read input from the user.

2.Create a Scanner object: This object (scanner) is used to capture user input.

3. Loop until the user enters '0':

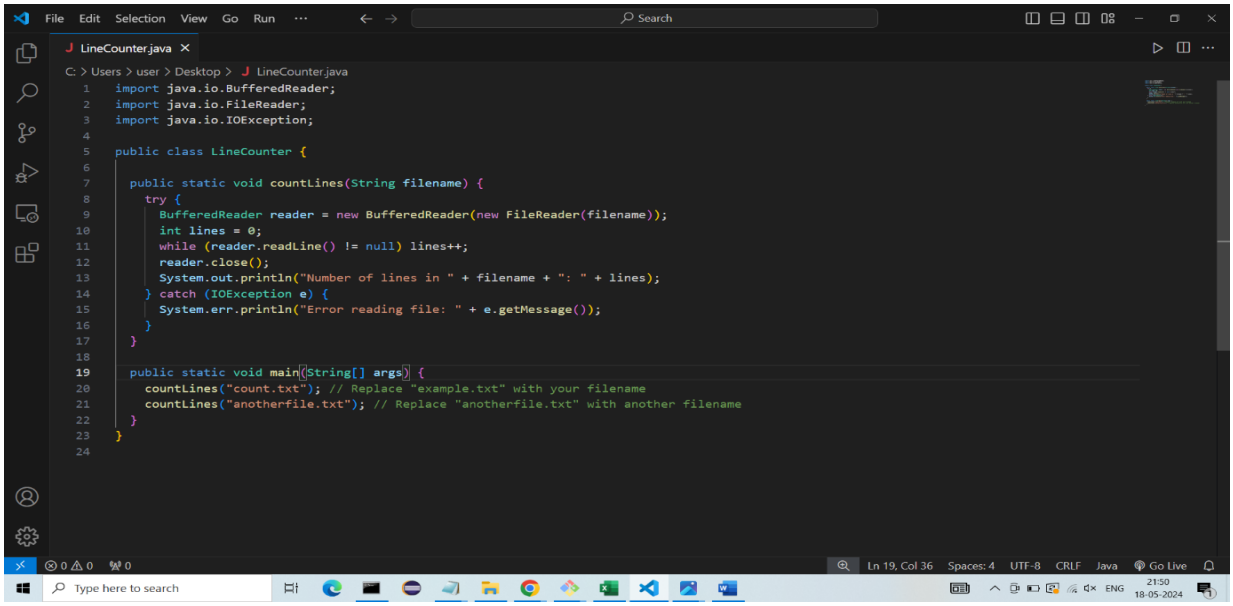
- Prompt the user: Ask the user to enter a number.
- Read the number: Use scanner.nextInt() to read the number entered by the user.
- Check if the number is '0': If the user enters '0', print "Exiting..." and break out of the loop.
- Determine if the number is odd or even:
    - If the number is divisible by 2 (number % 2 == 0), it is even.
    - Otherwise, it is odd.

4.Close the Scanner: It's a good practice to close the scanner to free up resources.

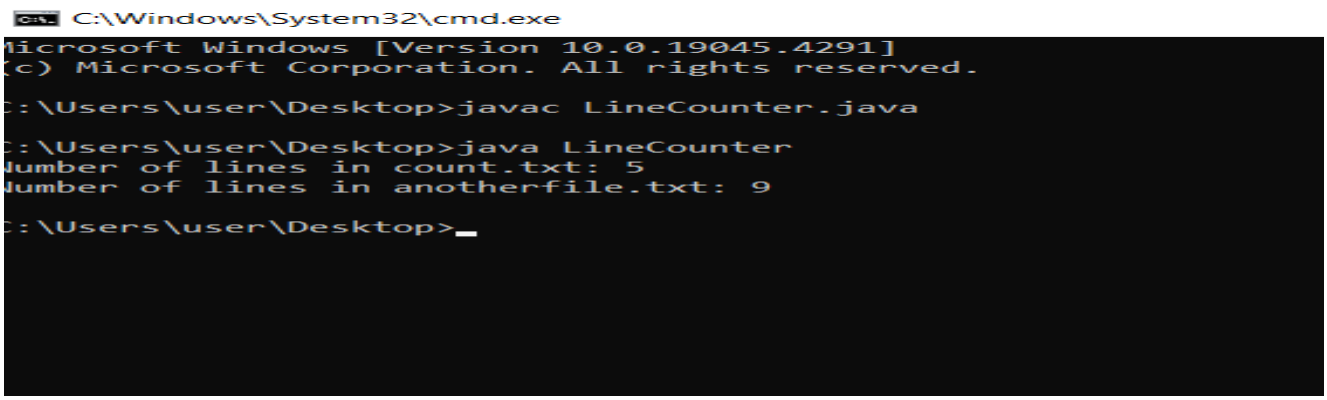
This program will continue to prompt the user for numbers and print whether each number is odd or even until '0' is entered.

Assignment 3: Create a function that takes a filename as an argument and prints the number of lines in the file. Call this function from your script with different filenames.

Function:



Output:



Explanation:

1.countLinesInFile Function:

- This function takes a filename string as input.
- It throws an IOException to indicate potential exceptions during file operations.

- Inside a try-with-resources block:
  - A Scanner object is created to read the file.
  - A while loop iterates through each line using hasNextLine().
  - Line count (lineCount) is incremented for each line.
- The function returns the total line count.

2.main Method:

- An array filename stores different filenames for testing.
- A for loop iterates through each filename.
- Inside the loop:
  - The countLinesInFile function is called with the current filename.
  - The returned line count is stored in the lineCount variable.
  - An informative message is printed displaying the filename and line count.
- An outer try-catch block handles potential IOException thrown by countLinesInFile. If an exception occurs, an error message is printed for that specific filename.

Assignment 4: Write a script that creates a directory named TestDir and inside it, creates ten files named File1.txt, File2.txt, ... File10.txt. Each file should contain its filename as its content (e.g., File1.txt contains "File1.txt").

CreateFiles - Notepad

File Edit Format View Help

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class CreateFiles {
    public static void main(String[] args) {
        // Create directory
        String directoryName = "TestDir";
        File directory = new File(directoryName);
        if (!directory.exists()) {
            directory.mkdir();
            System.out.println("Directory created: " + directoryName);
        } else {
            System.out.println("Directory already exists.");
        }

        // Create and write to files
        for (int i = 1; i <= 10; i++) {
            String fileName = "File" + i + ".txt";
            File file = new File(directory, fileName);
            try {
                FileWriter writer = new FileWriter(file);
                writer.write(fileName);
                writer.close();
                System.out.println("File created: " + fileName);
            } catch (IOException e) {
                System.out.println("An error occurred while creating " + fileName);
                e.printStackTrace();
            }
        }
    }
}
```

C:\Windows\System32\cmd.exe

```
Microsoft Windows [Version 10.0.19045.4291]
(c) Microsoft Corporation. All rights reserved.

C:\Users\user\Desktop>javac LineCounter.java

C:\Users\user\Desktop>java LineCounter
Number of lines in count.txt: 5
Number of lines in anotherfile.txt: 9

C:\Users\user\Desktop>javac CreateFiles.java

C:\Users\user\Desktop>java CreateFiles
Directory created: TestDir
File created: File1.txt
File created: File2.txt
File created: File3.txt
File created: File4.txt
File created: File5.txt
File created: File6.txt
File created: File7.txt
File created: File8.txt
File created: File9.txt
File created: File10.txt

C:\Users\user\Desktop>
```

Explanation:

1. Imports: We import necessary classes:

- File: Represents file and directory paths.
- FileWriter: Used to write text data to files.
- IOException: Handles potential exceptions during file operations.

2. Variables:
    - directoryName: Stores the name of the directory to be created ("TestDir").
    - numFiles: Defines the number of files to create (10 in this case).
3. Create Directory:
    - A File object is created for the directory.
    - An if statement checks if the directory already exists.
    - If it doesn't exist, the mkdir() method is used to create it.
    - If creation fails, an error message is printed.
4. Create Files:
    - A for loop iterates numFiles times.
    - Inside the loop:
        - The filename is constructed dynamically using string concatenation.
        - A File object is created for the specific file path within the directory.
        - A try-with-resources block is used to manage the FileWriter object:
            - Inside the block, a FileWriter is created for the file.
            - The filename string is written to the file using write().
            - A success message is printed with the absolute path of the created file.
        - The catch block handles potential IOException during file creation and prints an error message.

Assignment 5: Modify the script to handle errors, such as the directory already existing or lacking permissions to create files. Add a debugging mode that prints additional information when enabled.

CreateFilesImproved - Notepad

File Edit Format View Help

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class CreateFilesImproved {

    public static void main(String[] args) {
        String directoryName = "TestDir";
        int numFiles = 10;
        boolean debugMode = true;  // Set to true for additional logging

        // Create directory
        createDirectory(directoryName, debugMode);

        // Create files
        for (int i = 1; i <= numFiles; i++) {
            String fileName = "File" + i + ".txt";
            createFile(directoryName, fileName, debugMode);
        }
    }

    private static void createDirectory(String directoryName, boolean debugMode) {
        File directory = new File(directoryName);
        if (!directory.exists()) {
            if (debugMode) {
                System.out.println("Creating directory: " + directoryName);
            }
            if (!directory.mkdir()) {
                System.out.println("Error creating directory: " + directoryName);
            }
        } else if (debugMode) {
            System.out.println("Directory already exists: " + directoryName);
        }
    }

    private static void createFile(String directoryName, String fileName, boolean debugMode) {
        File file = new File(directoryName, fileName);
        try (FileWriter writer = new FileWriter(file)) {
            writer.write(fileName);
            if (debugMode) {
                System.out.println("Created file: " + file.getAbsolutePath());
            }
        } catch (IOException e) {
            System.out.println("Error creating file: " + fileName + " (" + e.getMessage() + ")");
        }
    }
}
```

C:\Windows\System32\cmd.exe

```
Microsoft Windows [Version 10.0.19045.4291]
(c) Microsoft Corporation. All rights reserved.

C:\Users\user\Desktop>javac CreateFiles.java

C:\Users\user\Desktop>java CreateFiles
Directory already exists.
File created: File1.txt
File created: File2.txt
File created: File3.txt
File created: File4.txt
File created: File5.txt
File created: File6.txt
File created: File7.txt
File created: File8.txt
File created: File9.txt
File created: File10.txt

C:\Users\user\Desktop>
```

Explanation:

1. debugMode Flag: We introduce a debugModeboolean flag set to true by default.
2. Separate Methods: The logic for creating the directory and files is encapsulated in separate methods:
    - createDirectory(String directoryName, booleandebugMode):
        - Handles directory creation with additional logging based on debugMode.
    - createFile(String directoryName, String fileName, booleandebugMode):
        - Creates individual files with error handling and logging based on debugMode.
3. Directory Creation:
    - The createDirectory method checks if the directory exists.
    - If it doesn't exist, it attempts to create it using mkdir().
        - In debugMode, it prints a message indicating directory creation.
        - If creation fails, an error message is printed.
    - If the directory already exists (and debugMode is enabled), a message is printed informing the user.
4. File Creation:
    - The createFile method takes the directory name, filename, and debugMode as arguments.
    - A try-with-resources block is used to manage the FileWriter.
    - Inside the block:
        - The file is created.
        - The filename is written to the file.
        - If debugMode is enabled, a message is printed with the absolute path of the created file.
    - The catch block handles potential IOException during file creation and prints an error message with the exception details.

Assignment 6: Given a sample log file, write a script using grep to extract all lines containing "ERROR". Use awk to print the date, time, and error message of each extracted line. Data Processing with sed .

Explanation:

1. **grep:**
   - The script starts with grep "ERROR" log_file.txt.
   - This command searches the log_file.txt for lines containing the string "ERROR" (case-sensitive).
   - The output of grep will be a list of lines containing "ERROR".
2. **awk:**
   - The pipe (|) symbol directs the output of grep to awk.
   - The awk command used here is: {print $1, $2, $3, $4}.
   - This command iterates through each line received from grep (lines containing "ERROR").
   - Inside the awk block:
     - $1, $2, $3, and $4 represent the first four fields separated by whitespace in each line.
     - The print statement outputs these four fields separated by spaces, effectively extracting the date, time, and the beginning of the error message (assuming the format has the date and time in the first four fields).

Assignment 7: Create a script that takes a text file and replaces all occurrences of "old_text" with "new_text". Use sed to perform this operation and output the result to a new file.

Shebang Line (Optional): The first line #!/bin/bash specifies the interpreter to use (bash) when running the script.

Error Handling:

The script checks if three arguments are provided using if [$# -lt3].

$# represents the number of arguments passed to the script.

If less than three arguments are provided, an error message with usage instructions is printed, and the script exits with an error code (exit 1).

Variable Assignment:

The script retrieves the arguments and assigns them to variables:

$1: Input file name

$2: Text to be replaced (old text)

$3: Replacement text (new text)

Input File Check:

The script checks if the input file exists using [! -f "$input_file"].

If the file doesn't exist, an error message is printed, and the script exits.

Output File Name:

The script constructs the output file name:

It removes the .txt extension (if present) using parameter expansion ("${input_file%.txt}").

It appends ".replaced.txt" to create a new filename with a clear indication of the modification.

Replacement with sed:

The core functionality relies on sed:

sed "s/$old_text/$new_text/g" "$input_file" > "$output_file"

s/: search and replace command in sed.

$old_text: pattern to be replaced.

$new_text: replacement text.

/g: global flag to replace all occurrences (not just the first).

"$input_file": specifies the input file.

> "$output_file": redirects the output of sed (modified content) to the new file.