Practices for Lesson 10

Practices Overview

In the first practice, you will use the JDK 7 NIO.2 API to write an application to create custom letters by merging a template letter with a list of names, utilizing `Files` and `Path` methods. In the second practice, you will use the `walkFileTree` method to copy all the files and directories from one folder to another on the disk. In the final optional practice, you will use the same method to write an application to recursively find and delete all the files that match a supplied pattern.

Practice 10-1: Summary Level: Writing a File Merge Application

Overview

In this practice, you will use the `Files.readAllLines` method to read the entire contents of two files: a form template, and a list of names to send form letters to. After creating a form letter with a name from the name list, you will use the `Files.write` method to create the custom letter. You will also use the `Pattern` and `Matcher` classes that you saw in the "String Processing" lesson.

Assumptions

You participated in the lecture for this lesson. Note there are Netbeans projects in the example directory to help you understand how to use the `Files` class `readAllLines` and `write` methods.

Tasks

1. Open the file `FormTemplate` in the `resources` directory.

   Note that this is a form letter with a string `<NAME>` that will be replaced by a name from the name list file.

2. Open the file `NamesList.txt` in the `resources` directory.
   a. This file contains the names to send the form letters to.
   b. Add your name to the end of the list.
   c. Save the file.
3. Open the project `FormLetterWriter` in the `practices` directory.
4. Expand the `FormLetterWriter` class. Notice that this class contains the main method, and that the application requires two parameters: One is the path to the form letter template, and the second is the path to the file containing the list of names to substitute in the form letter.
   a. After checking for a valid number of arguments, the main method then checks to see whether the `Path` objects point to valid files.
   b. The main method creates an instance of the `FileMerge` class with the form letter `Path` object and the list of names `Path` object.
   c. In a try block, the main method calls the `writeMergedForm` method of the `FileMerge` class. This is the method that you will write in this practice.
5. Expand the `FileMerge` class.
   a. Note the `writeMergedForms` method is empty. This is the method that you will write in this practice.
   b. The `mergeName` method uses the `Pattern` object defined in the field declarations to replace the string from the form template (first argument) with a name from the name list (second argument). It returns a String. For example, it replaces "`Dear <NAME>,`" with "`Dear Wilson Ball,`".
   c. The `hasToken` method returns a `boolean` to indicate whether the string passed in contains the token. This is useful to identify which string has the token to replace with the name from the name list.

6. Code the `writeMergedForms` method. The overall plan for this method is to read in the entire form letter, line by line, and then read in the entire list of names and merge the names with the form letter, replacing the placeholder in the template with a name from the list and then writing that out as a file. The net result should be that if you have ten names in the name list, you should end up with ten custom letter files addressed to the names from the name list. These ten files will be written to the resources directory.

   a. Read in all of the lines of the form letter into the `formLetter` field, and all of the lines from the name list into the `nameList` field.

      Note: Because `writeMergedForms` throws `IOException`, you do not need to put these statements into a try block. The caller of this method is responsible for handling any exceptions thrown.

   b. Create a `for` loop to iterate through the list of names (`nameList`) strings.

   c. Inside this for loop, create a new List object to hold the strings of the form letter. You need this new `List` to hold the modified form template strings to write out.

   d. Still inside the `for` loop, you will need to create a name for the custom letter. One easy way to do this is to use the name from the name list. You should replace any spaces in the name with underscores for readability of the file name. Create a new `Path` object relative to the form template path.

   e. Create another `for` loop, nested in the first loop, to iterate through the lines of the form template and look for the token string ("<NAME>") to replace with the `String` name from the `nameList`. Use the `hasToken` method to look for the String that contains the token string and replace that string with one containing the name from the `nameList`. Use the `mergeName` method to create the new String. Add the modified String and all of the other Strings from the `formLetter` to the new `customLetter List`.

   f. Still inside the first `for` loop, write the modified `List` of `Strings` that represents the customized form letter to the file system by using the `Files.write` method. Print a message that the file write was successful and close the outer for loop.

   g. Save the `FileMerge` class.

7. Modify the `FormLetterWriter` project to pass the form letter file and the name list file to the main method.

   a. Right-click the project and select Properties.

   b. Select Run.

   c. In the Arguments text field, enter: `D:\labs\resources\FormTemplate.txt D:\labs\resources\NamesList.txt (or your other folder)`

   d. Click OK.

8. Run the project. You should see new files created with the names from the name list. Each file should be customized with the name from the name list. For example, the `Tom_McGinn.txt` file should contain:

```
Dear Tom McGinn,

    It has come to our attention that you would like to prove
your Java Skills. May we recommend that you consider
certification from Oracle?  Oracle has globally recognized
Certification exams that will test your Java knowledge and
skills.

    Start with the Oracle Certified Java Associate exam, and
then continue to the Oracle Certified Java Programmer
Professional for a complete certification profile.

    Good Luck!

Oracle University
```

Practice 10-1: Detail Level: Writing a File Merge Application

Overview

In this practice, you will use the `Files.readAllLines` method to read the entire contents of two files: a form template, and a list of names to send form letters to. After creating a form letter with a name from the name list, you will use the `Files.write` method to create the custom letter. You will also use the `Pattern` and `Matcher` classes that you saw in the "String Processing" lesson.

Assumptions

You participated in the lecture for this lesson. Note there are Netbeans projects in the example directory to help you understand how to use the `Files` class `readAllLines` and `write` methods.

Tasks

1. Open the file `FormTemplate` in the `resources` directory.
   a. Select File > Open File
   b. Navigate to the `resources` directory in `D:\labs (or your other folder)`
   c. Select the file `FormTemplate.txt` and click the Open button.

   Note that this is a form letter with a string placeholder token `<NAME>` that will be replaced by a name from the name list file.
2. Open the file `NamesList.txt` in the `resources` directory.
   a. This file contains the names to send the form letters to.
   b. Add your name to the end of the list.
   c. Save the file.
3. Open the project `FormLetterWriter` in the `practices` directory.
   a. Select File > Open Project.
   b. Browse to `D:\labs\10\practices. (or your other folder)`
   c. Select `FormLetterWriter`.
   d. Select the "Open as Main Project" check box.
   e. Click the Open Project button.
4. Expand the `FormLetterWriter` class. Notice that this class contains the main method, and that the application requires two parameters: One is the path to the form letter template, and the second is the path to the file containing the list of names to substitute in the form letter.
   a. After checking for a valid number of arguments, the main method then checks to see whether the `Path` objects point to valid files.
   b. The main method creates an instance of the `FileMerge` class with the form letter `Path` object and the list of names `Path` object.
   c. In a try block, the main method calls the `writeMergedForm` method of the `FileMerge` class. This is the method that you will write in this practice.

5. Expand the `FileMerge` class.

   a. Note the `writeMergedForms` method is empty. This is the method that you will write in this practice.

   b. The `mergeName` method uses the `Pattern` object defined in the field declarations to replace the string from the form template (first argument) with a name from the name list (second argument). It returns a String. For example, it replaces "`Dear <NAME>,`" with "`Dear Wilson Ball,`".

   c. The `hasToken` method returns a `boolean` to indicate whether the string passed in contains the token. This is useful to identify which string has the token to replace with the name from the name list.

6. Code the `writeMergedForms` method. The overall plan for this method is to read in the entire form letter, line by line, and then read in the entire list of names and merge the names with the form letter, replacing the placeholder in the template with a name from the list and then writing that out as a file. The net result should be that if you have ten names in the name list, you should end up with ten custom letter files addressed to the names from the name list. These ten files will be written to the resources directory.

   a. Create an instance of the default `Charset`. This argument is required for the `Files.readAllLines` method.

   ```
   Charset cs = Charset.defaultCharset();
   ```

   b. Read in all of the lines of the form letter into the `formLetter` field, and all of the lines from the name list into the `nameList` field.

   Note: Because `writeMergedForms` throws `IOException`, you do not need to put these statements into a try block. The caller of this method is responsible for handling any exceptions thrown.

   ```
   formLetter = Files.readAllLines(form, cs);
   nameList = Files.readAllLines(list, cs);
   ```

   c. Create a `for` loop to iterate through the list of names (`nameList`) strings.

   d. Inside this for loop, create a new List object to hold the strings of the form letter. You will need this new `List` to hold the modified form template strings to write out.

   ```
   for (int j = 0; j < nameList.size(); j++) {
       customLetter = new ArrayList<>();
   ```

   e. Still inside the `for` loop, you need to create a name for the custom letter. One easy way to do this is to use the name from the name list. You should replace any spaces in the name with underscores for readability of the file name. Create a new `Path` object relative to the form template path.

   ```
       String formName = nameList.get(j).replace(' ',
   '_').concat(".txt");
       Path formOut = form.getParent().resolve(formName);
   ```

   f.

Create another `for` loop, nested in the first loop, to iterate through the lines of the form template and look for the token placeholder string ("<NAME>") to replace with the `String` name from the `nameList`. Use the `hasToken` method to look for the String that contains the token string and replace that string with one containing the name from the `nameList`. Use the `mergeName` method to create the new String. Add the modified String and all of the other Strings from the `formLetter` to the new `customLetter List`.

```
    for (int k = 0; k < formLetter.size(); k++) {
        if (hasToken(formLetter.get(k))) {
            customLetter.add(mergeName(formLetter.get(k),
nameList.get(j)));
        } else {
            customLetter.add(formLetter.get(k));
        }
    }
```

g. Finally, still inside the first `for` loop, write the modified `List` of `Strings` that represents the customized form letter to the file system by using the `Files.write` method. Print a message that the file write was successful and close the outer for loop.

```
    Files.write(formOut, customLetter, cs);
    System.out.println ("Wrote form letter to: " +
nameList.get(j));
} // closing brace for the outer for loop
```

h. Reformat the code to ensure that you have everything in the right place. Press the Ctrl-Alt-F key combination or right-click in the editor pane and choose Format.

i. Save the `FileMerge` class.

7. Modify the `FormLetterWriter` project to pass the form letter file and the name list file to the main method.

   a. Right-click the project and select Properties.

   b. Select Run.

   c. In the Arguments text field, enter: `D:\labs\resources\FormTemplate.txt D:\labs\resources\NamesList.txt (or your other folder)`

   d. Click OK.

8. Run the project. You should see new files created with the names from the name list. Each file should be customized with the name from the name list. For example, the `Tom_McGinn.txt` file should contain:

```
Dear Tom McGinn,


    It has come to our attention that you would like to prove
your Java Skills. May we recommend that you consider
certification from Oracle?  Oracle has globally recognized
Certification exams that will test your Java knowledge and
skills.


    Start with the Oracle Certified Java Associate exam, and
then continue to the Oracle Certified Java Programmer
Professional for a complete certification profile.


    Good Luck!


Oracle University
```

Practice 10-2: Summary Level: Recursive Copy

Overview

In this practice, you write Java classes that use the `FileVisitor` class to recursively copy one directory to another.

Assumptions

You participated in the lecture for this lesson.

Tasks

1. Open the project `RecursiveCopyExercise` in the directory `D:\labs\10\practices`. `(or your other folder)`
2. Expand the Source Packages folder and subfolders and look at the `Copy.java` class.
   a. Note that the `Copy.java` class contains the `main` method.
   b. The application takes two arguments, a `source` and `target` paths.
   c. If the target file or directory exists, the user is prompted whether to overwrite.
   d. If the answer is yes (or the letter y), the method continues.
   e. An instance of the `CopyFileTree` class is created with the `source` and `target`.
   f. This instance is then passed to the `walkFileTree` method (with the `source` Path object).

   You will need to provide method bodies for the methods in the `CopyFileTree.java` class.
3. Open the `CopyFileTree.java` class.
   a. This class implements the `FileVisitor` interface. Note that `FileVisitor` is a generic interface, and this interface is boxed with the `Path` class. This allows the interface to define the type of the arguments passed to its methods.
   b. The `CopyFileTree` implements all the methods defined by `FileVisitor`.
   c. Your task is to write method bodies for the `preVisitDirectory` and `visitFile` methods. You will not need the `postVisitDirectory` method, and you have been provided a method body for the `visitFileFailed` method.
4. Write the method body for `preVisitDirectory`. This method is called for the starting node of the tree and every subdirectory. Therefore, you should copy the directory of the source to the target. If the file already exists, you can ignore that exception (because you are doing the copy because the user elected to overwrite.)
   a. Start by creating a new directory that is relative to the target passed in, but is the node name from the source. The method call to do this is:

      ```
      Path newdir = target.resolve(source.relativize(dir));
      ```

   b. In a try block, copy the directory passed to the `preVisitDirectory` method to the `newdir` that you created.

c. You can ignore any `FileAlreadyExistsException` thrown, because you are overwriting any existing folders and files in this copy.

d. Catch any other `IOExceptions`, and use the `SKIP_SUBTREE` return to avoid repeated errors.

5. Write the method body for the `visitFile` method. This method is called when the node reached is a file. The file is passed as an argument to the method.

   a. As with the `preVisitDirectory`, you must rationalize the file reached (source path) with the path that you wanted for the target. Use the same method call as above (only using `file` instead of `dir`):

      ```
      Path newdir = target.resolve(source.relativize(file));
      ```

   b. As in the `preVisitDirectory` method, use the `Files.copy` method in a try block. Make sure that you pass `REPLACE_EXISTING` in as an option to overwrite any existing file in the directory.

   c. Catch any `IOException` thrown and report an error.

   d. Fix any missing imports.

   e. Save your class.

6. Test your application by copying a directory (ideally with subdirectories) to another location on the disk. For example, copy the `D:\labs\10` directory to `D:\Temp`.

   a. Right-click the project and select Properties.

   b. Click Run.

   c. Enter the following as Arguments:

      ```
      D:\labs\10 D:\Temp
      ```

   d. Click OK.

7. Run the project and you should see the following message:

   ```
   Successfully copied D:\labs\10 to D:\Temp
   ```

   a. Run the project again, and you should be prompted:

   ```
   Target directory exists. Overwrite existing files? (yes/no):
   ```

Practice 10-2: Detailed Level: Recursive Copy

Overview

In this practice, you write Java classes that use the FileVisitor class to recursively copy one directory to another.

Assumptions

You participated in the lecture for this lesson.

Tasks

1. Open the project `RecursiveCopyExercise` in the directory `D:\labs\10\practices. (or your other folder)`
   a. Select File > Open Project.
   b. Browse to `D:\labs\10\practices. (or your other folder)`
   c. Select `RecursiveCopyExercise`.
   d. Select the “Open as Main Project” check box.
   e. Click the Open Project button.
2. Expand the Source Packages folder and subfolders and look at the `Copy.java` class.
   a. Note that the `Copy.java` class contains the `main` method.
   b. The application takes two arguments, a `source` and `target` paths.
   c. If the target file or directory exists, the user is prompted whether to overwrite.
   d. If the answer is yes (or the letter y), the method continues.
   e. An instance of the `CopyFileTree` class is created with the `source` and `target`.
   f. This instance is then passed to the `walkFileTree` method (with the `source` Path object).

   You will need to provide method bodies for the methods in the `CopyFileTree.java` class.
3. Open the `CopyFileTree.java` class.
   a. This class implements the `FileVisitor` interface. Note that `FileVisitor` is a generic interface, and this interface is boxed with the `Path` class. This allows the interface to define the type of the arguments passed to its methods.
   b. The `CopyFileTree` implements all the methods defined by `FileVisitor`.
   c. Your task is to write method bodies for the `preVisitDirectory` and `visitFile` methods. You will not need the `postVisitDirectory` method, and you have been provided a method body for the `visitFileFailed` method.

4. Write the method body for `preVisitDirectory`. This method is called for the starting node of the tree and every subdirectory. Therefore, you should copy the directory of the source to the target. If the file already exists, you can ignore that exception (because you are doing the copy because the user elected to overwrite.)

   a. Start by creating a new directory that is relative to the target passed in, and is the node name from the source. The method call to do this is:

      ```
      Path newdir = target.resolve(source.relativize(dir));
      ```

   b. In a try block, copy the directory passed to the `preVisitDirectory` method to the `newdir` that you created.

      ```
      try {
          Files.copy(dir, newdir);
      ```

   c. You can ignore any `FileAlreadyExistsException` thrown, because you are overwriting any existing folders and files in this copy.

      ```
      } catch (FileAlreadyExistsException x) {
          // ignore
      ```

   d. Do catch any other `IOExceptions`, and use the `SKIP_SUBTREE` return to avoid repeated errors.

      ```
      } catch (IOException x) {
          System.err.format("Unable to create: %s: %s%n",
                            newdir, x);
          return SKIP_SUBTREE;
      }
      ```

5. Write the method body for the `visitFile` method. This method is called when the node reached is a file. The file is passed as an argument to the method.

   a. As with the `preVisitDirectory`, you must rationalize the file reached (source path) with the path that you wanted for the target. Use the same method call as above (only using `file` instead of `dir`):

      ```
      Path newdir = target.resolve(source.relativize(file));
      ```

   b. As in the `preVisitDirectory` method, use the `Files.copy` method in a try block. Make sure that you pass `REPLACE_EXISTING` in as an option to overwrite any existing file in the directory.

      ```
      try {
          Files.copy(file, newdir, REPLACE_EXISTING);
      ```

      Note: To use the `REPLACE_EXISTING` enum type, you must import the `java.nio.file.StandardCopyOption` enum class using a static import, like this:

      ```
      import static java.nio.file.StandardCopyOption.*;
      ```

   c. Catch any `IOException` thrown and report an error.

      ```
      } catch (IOException x) {
          System.err.format("Unable to copy: %s: %s%n", source, x);
      }
      ```

d. Fix any missing imports.

e. Save your class.

6. Test your application by copying a directory (ideally with subdirectories) to another location on the disk. For example, copy the `D:\labs\10` directory to `D:\Temp`.

   a. Right-click the project and select Properties.

   b. Click Run.

   c. Enter the following as Arguments:

      ```
      D:\labs\10 D:\Temp
      ```

   d. Click OK.

7. Run the project and you should see the following message:

   ```
   Successfully copied D:\labs\10 to D:\Temp
   ```

   a. Run the project again, and you should be prompted:

      ```
      Target directory exists. Overwrite existing files? (yes/no):
      ```

(Optional) Practice 10-3: Summary Level: Using `PathMatcher` to Recursively Delete

Overview

In this practice, you write a Java main that creates a `PathMatcher` class and uses `FileVisitor` to recursively delete a file or directory pattern.

Assumptions

You have completed the previous practice.

Tasks

1. Open the project `RecursiveDeleteExercise` in the practices directory.
2. Expand the Source Packages folders.
3. Open the `Delete.java` class file. This is the class that contains the `main` method. The main class accepts two arguments: the first is the starting path and the other the pattern to delete.
4. You must code the remainder of this class. Look at the comments for hints as to what to do.
    a. Start by creating a `PathMatcher` object from the search string passed in as the second argument. To obtain a `PathMatcher` instance, you will need to use the `FileSystems` class to get a path matcher instance from the default file system.
    b. Create a `Path` object from the first argument.
    c. If the starting path is a file, check it against the pattern using the `PathMatcher` instance that you created. If there is a match, delete the file, and then terminate the application.
    d. If the starting path is a directory, create an instance of the `DeleteFileTree` with the starting directory and the `PathMatcher` object as initial arguments in the constructor. Pass the starting directory and the file tree to a `Files.walkFileTree` method to recursively look for the pattern to delete.
    e. Fix any missing imports.
    f. Save the Delete class.
5. Open the `DeleteFileTree` class file. This class implements `FileVisitor`. This class recursively looks for instances of files or directories that match the `PathMatcher` object passed into the constructor. This class is complete with the exception of the `delete` method.
    a. The `delete` method is called by the `preVisitDirectory` and `visitFile` methods. You must check whether the file or directory reached by these methods matches the pattern.
    b. We only want to match the path name at the node, so use the `Path.getFileName` method to obtain the file name at the end of the full path.
    c. If the name matches, use the `Files.delete` method to attempt to delete the file pattern and print a result statement, or print an error if an `IOException` is thrown.
    d. Save the `DeleteFileTree` class.

6. Run the `Delete` application using a temporary directory.
    a. For example, if you completed the first practice, you can delete all the Java class files from the `D:\Temp` directory.
    b. Right-click the project and select Properties.
    c. Click Run and enter the following in the Arguments text field:

        ```
        D:\Temp\examples *.class
        ```

    d. Run the project.

(Optional) Practice 10-3: Detailed Level: Using `PathMatcher` to Recursively Delete

Overview

In this practice, you write a Java main that creates a `PathMatcher` class and uses `FileVisitor` to recursively delete a file or directory pattern.

Assumptions

You have completed the previous practice.

Tasks

1. Open the project `RecursiveDeleteExercise` in the practices directory.
   a. Select File > Open Project.
   b. Browse to `D:\labs\10\practices`. `(or your other folder)`
   c. Select `RecursiveDeleteExercise`.
   d. Click the Open Project button.
2. Expand the Source Packages folders.
3. Open the `Delete.java` class file. This is the class that contains the `main` method. The main class accepts two arguments: the first is the starting path and the other the pattern to delete.
4. You must code the remainder of this class. Look at the comments for hints as to what to do.
   a. Start by creating a `PathMatcher` object from the search string passed in as the second argument. To obtain a `PathMatcher` instance, you will need to use the `FileSystems` class to get a path matcher instance from the default file system.

      ```
      PathMatcher matcher =
      FileSystems.getDefault().getPathMatcher("glob:" + args[1]);
      ```

   b. Create a `Path` object from the first argument.

      ```
      Path root = Paths.get(args[0]);
      ```

c. If the starting path is a file, check it against the pattern using the `PathMatcher` instance that you created. If there is a match, delete the file, and then terminate the application.

```
if (!Files.isDirectory(root)) {
    Path name = root.getFileName();
    if (name != null && matcher.matches(name)) {
        try {
            Files.delete(root);
            System.out.println("Deleted :" + root);
            System.exit(0);
        } catch (IOException e) {
            System.err.println("Exception deleting file: " +
                               root);
            System.err.println("Exception: " + e);
            System.exit(-1);
        }
    }
}
```

d. If the starting path is a directory, create an instance of the `DeleteFileTree` with the starting directory and the `PathMatcher` object as initial arguments in the constructor. Pass the starting directory and the file tree to a `Files.walkFileTree` method to recursively look for the pattern to delete.

```
DeleteFileTree deleter = new DeleteFileTree(root, matcher);
try {
    Files.walkFileTree(root, deleter);
} catch (IOException e) {
    System.out.println("Exception: " + e);
}
```

e. Fix any missing imports.

f. Save the Delete class.

5. Open the `DeleteFileTree` class file. This class implements `FileVisitor`. This class recursively looks for instances of files or directories that match the `PathMatcher` object passed into the constructor. This class is complete with the exception of the `delete` method.

   a. The `delete` method is called by the `preVisitDirectory` and `visitFile` methods. You must check whether the file or directory reached by these methods matches the pattern.

   b. We only want to match the path name at the node, so use the `Path.getFileName` method to obtain the file name at the end of the full path.

   ```
   Path name = file.getFileName();
   ```

c. If the name matches, use the `Files.delete` method to attempt to delete the file pattern and print a result statement, or print an error if an `IOException` is thrown.

```
if (matcher.matches(name)) {
    //if (name != null && matcher.matches(name)) {
    try {
         Files.delete(file);
         System.out.println("Deleted: " + file);
    } catch (IOException e) {
         System.err.println("Unable to delete: " + name);
         System.err.println("Exception: " + e);
    }
}
```

d. Save the `DeleteFileTree` class.

6. Run the `Delete` application using a temporary directory.

   a. For example, if you completed the first practice, you can delete all the Java class files from the `D:\Temp` directory.

   b. Right-click the project and select Properties.

   c. Click Run and enter the following in the Arguments text field:

      ```
      D:\Temp\examples *.class
      ```

   d. Run the project.