

Amelia runs a small local bookstore. Lucas, on the other hand, has set up an online business selling first-edition copies of classic fiction titles. For both of them, tracking the books that they have in stock is vital to their business success. Amelia and Lucas are possible future customers for an application you are helping to develop called My Books Management System.

The My Books Management System currently comprises a directory named *myBooks* that contains a collection of sample files representing individual books. In this lab, you will demonstrate your ability to handle various file and directory operations in Java by managing file systems within the My Books Management System. The tasks you are asked to do, such as creating directories, renaming, copying or deleting files, and implementing methods to serialize and deserialize a book object, are common file handling and directory tasks you might encounter in real-world software development.

Goal

Your goal is to implement efficient file system management in the My Books Management System. You will manage directories in Java by performing a series of file and directory operations each of which is critical for efficient file system management.

Requirements

- Read all files available in the directory and list them in the console. This will be used to print all book files.
- Create a new directory in the system. This directory will store backup copies of your book files.
- Rename files to correct any naming errors or to change the directory or file name.
- Copy files from one directory to another. This task will test your ability to duplicate files effectively, including handling file copying operations and potential errors.
- Delete specific files from a directory. This shows your ability to manage file lifecycle, including removing unnecessary or obsolete files, which is crucial for maintaining an efficient and clean file system.
- Implement methods to serialize a Book object to a file and deserialize it back into an object. Object persistence is a key concept for saving and restoring object states, which is essential for data storage and retrieval in real-world applications.



Note: When you encounter this icon, it's time to get into your IDE and start coding!

In your lab environment, open IntelliJ by double-clicking on the icon.

Because he runs an online business, Lucas stores the books that he sells in a separate storage location. Therefore, being able to access a list of what is in stock is a critical requirement for him, which the My Book Management System needs to fulfill. Your first coding task is to make sure that this is possible.

You're aware that work has already been done on the *MyBooksManagement* project files. Before you begin this coding task, you sensibly take a moment to review the existing project file so that you can feel confident when you begin to code.

Open your Java project named *MyBooksManagementSystem* in IntelliJ IDEA. This project includes four Java files and one folder with three text files. Take a moment or two to identify the following items.


1. *myBooks* folder: This is used to store text files that you will work on within this assignment.
2. *FileCreator* class: This is used to create text files (books) in the *myBooks* directory.
3. *DirectoryManager* class: Used to create and manage files and directories in the program.
4. *Book* class. This class defines the book object with *title*, *author*, and *isbn* fields and implements `Serializable` to allow the object to be serialized.
5. *Main* class: This class is used to test the application.

Now that you've got a comprehensive view of the existing content, your first requirement is to create a new method to list all files in the "*myBooks*" folder. This will read all files in the directory and print them in the console.

Remember

If you've had any difficulty, you can refer to the video [Working with directories](#) to refresh your knowledge.

•

-  **TODO 1:** Inside the *DirectoryManager* class, create a new method.
 - **TODO1a:** Define the method signature following these guidelines:
 - Method name: *listFilesAndDirectories*.
 - Method parameters: one parameter for the directory path name. An example of this could be *directoryPath*.
 - No return value as its primary purpose is to print the names of the files.
 - The method must be static so that it can be called conveniently with the class name using the dot operator.
 - **TODO1b:** Create a *New File* object for the *directoryPath*.

- TODO1c: Use the *listFiles()* method from the *File* class to get an array of *File* objects.
- TODO 1d: Check if the array is null or empty.
 - If it is empty or null, print an appropriate message to the console indicating that the directory is empty or does not exist.
 - If not, loop through the array and print each file's name.

Here is the sample code for the method with TODOs to help you:

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

```
// TODO 1a: Define a static void method called listFilesAndDirectories with a  
single parameter directoryPath of type String
```

```

public static void listFilesAndDirectories(String directoryPath) {

    // TODO 1b: Create a new File object for the directoryPath


    // TODO 1c: Use the listFiles() method from the File class to get an array of
    File objects


    // TODO 1d: Check if the array is null or empty

    if (files == null || files.length == 0) {

        // If it is empty or null, print an appropriate message indicating the
        same

    }

    else {

        // If not, loop through the array and print each file's name

    }

}

```

The method has already been called in the *Main* class. You only need to uncomment the code from line 10 to line 25.

Pause and check

Okay, it's time to check that what you've done so far works as expected. Run your program, and a list of book titles should be printed in the console.

List of books available in ./myBooks directory:

Title: 1984.txt

Title: The Great Gatsby.txt

Title: To Kill a Mockingbird.txt

So what's the next step? Well, the My Book Management System application should allow for the fact that the list of books may need to be copied to a different directory, perhaps for backup purposes. Of course, the directory needs to be created before this can happen.

Your next task is to define a method for creating a new directory. In this case, it will be a backup directory named *myBooksBackup*.

Remember

Refer to the videos [Standard file I/O operations](#) and [Working with directories](#) if you need to refresh your knowledge.

-



TODO 2: Create a new method inside the *DirectoryManager* class.

- TODO 2a: Define the method signature using the following guidelines:
 - Method name: *createDirectory*.
 - This method will take one parameter *directoryPath* name.
 - Return Type: The method returns no value (void). Its primary purpose is to create a directory if it does not already exist.
 - The method must be static so that it can be called conveniently with the class name using the dot operator.
- TODO 2b: Create a *New File* object for the directory path.
- TODO 2c: Check if the directory exists using the *exists()* method from the *File* class
 - If the directory does not exist, create the directory using the *makedirs()* method and print a message indicating that the directory was created.
 - If the directory already exists, print a message indicating the same.

What is your plan for achieving this? Here is the sample code for the method with TODOs to help you:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

```
// TODO 2a: define a static void method called createDirectory with a single
parameter directoryPath of type String
```

```
public static void createDirectory(String directoryPath) {
```

```
    // TODO 2b: create a new File object for the directoryPath
```

```
    // TODO 2c: check if the directory exists using the exists() method from the
File class
```

```

    if (!directory.exists()) {

        // If the directory does not exist, create the directory using the mkdirs()
        method from the File class

        // Print a message indicating that the directory was created

    }

else {

    // If the directory already exists, print a message indicating the same

}

}

```

The method has already been called in the *Main* class. You only need to uncomment the code from line 28 to line 31.

Pause and check

It's time to rerun your program to ensure everything is on track and confirm that the application can now create new directories. Run the program to determine that a message confirming the directory creation is printed as follows:

1

```
Directory created successfully: ./myBookBackup
```

If your program printed that message, well done! But perhaps your program printed this message instead?

1

```
Failed to create directory.
```

This message indicates that your program has failed to create the required directory. What might have caused this? It could be the result of insufficient permissions or an

invalid path. Perhaps files with the same name already exist? The issue could also be disk space limitations or concurrency conflicts. These could prevent the `makedirs()` method from successfully creating the directory.


If you encounter these issues, follow these steps to troubleshoot and resolve the problem:

1. Verify *Path* and permissions: Ensure the *directoryPath* is correct and does not contain invalid characters or spaces. Check that your program has the necessary permissions to create a directory in the specified location.
2. Check for existing files: Ensure that a file or directory with the same name does not already exist. If a file exists with the same name, the program will fail to create a directory.
3. Disk space: Confirm there is enough available disk space on the drive where you are trying to create the directory.
4. Concurrency issues: If multiple processes or threads are accessing the file system simultaneously, ensure proper synchronization to prevent conflicts.
5. Review error messages: Check any error messages or exceptions for specific information on why directory creation failed. Adjust the code based on these insights.

By addressing these key areas, you can resolve issues related to directory creation and ensure your program functions correctly.

Well done. A named directory now exists. But what if a user needs to change the name assigned to that directory during the creation process? Perhaps when Amelia created a directory it reflected conditions within her business. But if conditions changed over time, a new name might be more appropriate? The My Books Management System has to allow her to rename it.

To provide this functionality, you must create a new method to rename a directory in the system. In this case, you will rename the '*myBooksBackupDirectory*' directory to '*BooksBackupDirectory*'.

-  TODO 3: Create a new method to rename directories in the *DirectoryManager* class.
 - TODO 3a: Define the method signature using the following guidelines:
 - Method name: *renameDirectory*.
 - This method will take two parameters, *currentDirectory* and *newDirectory* path names.
 - Return type: The method returns no value (void); and its primary purpose is to rename a directory.
 - The method must be static so that it can be called conveniently with the class name using the dot operator.
 - TODO 3b: Create *File* objects for both the current and new directory names.

- TODO 3c: Check if the new directory already exists using the *exists()* method.
 - If the new directory exists, print an error message and return
- TODO 3d: Use the *renameTo()* method to try renaming the old directory to the new directory.
 - If the rename fails, print an error message.
 - If the rename is successful, print a success message.

The following sample code provides a skeleton of the method if needed:

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19

20

21

```
// TODO 3a: Define a static void method called renameDirectory with two parameters:  
currentDirectory and newDirectory of type String
```

```
public static void renameDirectory(String currentDirectory, String newDirectory) {
```

```
    // TODO 3b: create two File objects for the currentDirectory and newDirectory
```

```
    // TODO 3c: Check if the new directory already exists using the exists() method  
    from the File class
```

```
    if (newDir.exists()) {
```

```
    }
```

```
    // TODO 3d: Use the renameTo() method from the File class to rename the old  
    directory to the new directory
```

```
    if (!oldDir.renameTo(newDir)) {
```

```
        // If the rename fails, print an error message
```

```
    }
```

```
else {
```

```
    // If the rename is successful, print a success message
```

```
    }
```

```
}
```

The method has already been called in the *Main* class. Just uncomment the code from lines 43 to 46.

Pause and check

Run your program again to check that the outcomes are what you expected. You should receive a message saying that the directory has been renamed.


1

```
Directory has been renamed successfully.
```

If you encounter the *"Failed to rename directory."* message:

- Ensure the directory exists and check for sufficient permissions.
- Verify that the new directory path is valid.
- Another application isn't accessing the directory you're trying to rename. For example, suppose a file within the directory is open in another application. In that case, the rename operation may fail because the operating system prevents changes to directories that are actively being used.

If Lucas signs up to use the My Books Management System, he would expect to be able to create copies of files in different directories. For example, he might have "The Great Gatsby" title in a directory called "Early 20th Century Fiction" and also in another directory called "Modern American Fiction." You need to create a new method to copy files from one directory into another directory. Here you can use the example of copying from the *myBooks* directory to the *BooksBackupDirectory*.

-  **TODO 4:** Create a new method to copy files in the *DirectoryManager* class.
 - **TODO 4a:** Define the method signature using the following guidelines:
 - Method name: *copyFiles*.
 - This method will take two String parameters, *sourceDir* and *destDir*.
 - The method must be static so that it can be called conveniently with the class name using the dot operator.
 - **TODO 4b:** Inside the method, create *Path* objects from the String parameters using the *Paths.get()* method.
 - **TODO 4c:** Write a *try-catch* block to handle *IOException* because creating a new directory and copying files can throw an *IOException*.
 - **TODO 4d:** Inside the *try* block, check if the destination directory exists using the *exists()* method.
 - If the destination directory does not exist, create the directory using the *createDirectories()* method.
 - **TODO 4e:** Iterate through the files in the source directory using a loop.

- For each file, create a Path object using the file's name and the *destDir*.
- Use the *copy()* method from the *Files* class to copy the file to the destination directory.
- Print a message indicating that the file was copied.
- Here's a sample code with a list of instructions to guide you through the implementation.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

```
// TODO 4a: Define a static void method called copyFiles with two parameters
sourceDir and destDir of type String
```

```
public static void copyFiles(String sourceDir, String destDir) {
```

```
    // TODO 4b: create Path objects for the sourceDir and destDir using the
    Paths.get() method
```

```
    // TODO 4c: Write a try-catch block to handle IOException because creating a
    new directory and copying files can throw an IOException
```

```
    try {
```

```
        // TODO 4d: Check if the destination directory exists using the exists()
        method from the Files class
```

```
        if (!Files.exists(destPath)) {
```

```
            // If the destination directory does not exist, create the directory
            using the createDirectories() method from the Files class
```

```

    }

    // TODO 4e: Iterate through the files in the source directory using a loop

    File sourceDirectory = new File(sourceDir);

    File[] files = sourceDirectory.listFiles();

    // For each file, create a Path object using the file's name and the
    destDir

    for (File file : files) {

        // Use the copy() method from the Files class to copy the file to the
        destination directory

        // Print a message indicating that the file was copied

    }

}

catch(IOException e){

    System.out.println(e.getMessage());

}

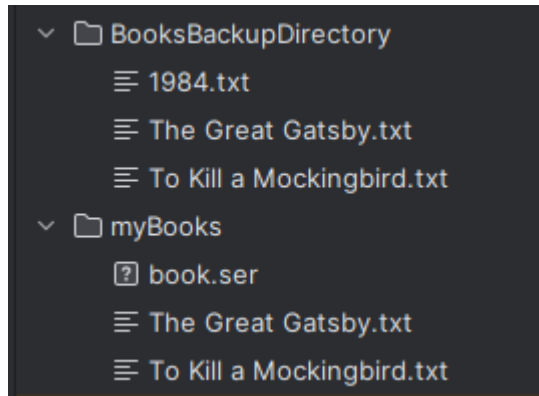
}

```

The method is already called in the *Main* Class. You only need to uncomment the code from lines 43 to 45.

Pause and check


Run your program and check your output. The *BooksBackupDirectory* and the *myBooks* directory should contain similar lists of book titles, that is, *The Great Gatsby.txt* and *To Kill a Mockingbird.txt*.



Output of lists of book titles, including *The Great Gatsby.txt* and *To Kill a Mockingbird.txt*.

Lucas' online business is growing rapidly, and his stock is changing rapidly. Any application he uses must allow him to delete titles when he has sold them and they are no longer in stock.

You will need to add this functionality to the My Books Management System.

-  **TODO 5:** Create a new method to delete files in the *DirectoryManager* class.
 - **TODO 5a:** Define the method signature using the following guidelines:
 - Method name: *deleteFile*.
 - This method takes one String parameter, which is the path name of the file you want to delete.
 - The method must be static so that it can be called conveniently with the class name using the dot operator.
 - **TODO 5b:** Inside the method, create a File object using the provided *fileName*.
 - **TODO 5c:** Attempt to delete the file using the *delete()* method from the *Files* class:
 - If the file is deleted successfully, print a message indicating the same.
 - If the file deletion fails, print an error message.

Refer to this sample code if you need help while implementing the code:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

```
// TODO 5a: Define a static void method called deleteFile with a single parameter  
fileName of type String
```

```
public static void deleteFile(String fileName) {
```

```
// TODO 5b: create a File object using the provided fileName
```

```
// TODO 5c: Attempt to delete the file using the delete() method from the File  
class
```

```
if (file.delete()) {
```

```
// If the file is deleted successfully, print a message indicating the same
```

```
} else {
```

```
// If the file deletion fails, print an error message
```



```
}  
  
}
```

The method is already called the *main* class. Just uncomment the code from lines 50 to 51.

Pause and check

Run your program, and you should get a confirmation message that the file has been deleted.

1

```
./myBooks/1984.txt has been deleted.
```

Lucas will delete a book file when that title is out of stock in his online shop. However, if he gets that book title back into stock and restores the file, he would want the settings information stored in it to also be restored. You must incorporate this functionality into the My Books File Management System.

In this task, you must demonstrate how to serialize an object to a file and then deserialize it back into an object. This is useful for saving the state of an object to a file and later restoring it.

Remember

If you need to refresh your knowledge, consult the video [The process of serialization and deserialization](#).

- - 🖥️ TODO 6: Create the *serializeBook* method in the *Book* class. This method will serialize the *Book* object to a file specified by *filePath*.
 - TODO 6a: Define the *serializeBook* method using the following guidelines:
 - Method name: *serializeBook*.
 - Parameters: *Book book* and *String filePath*.
 - Return type: *void*.
 - TODO 6b: Inside the method, create a *try-with-resources* block.
 - TODO 6c: Inside the *try-with-resources* parenthesis, create a new *FileOutputStream* object with the *filePath* and a new *ObjectOutputStream* object with the *FileOutputStream* object.
 - TODO 6d: Inside the *try* block, use the *writeObject()* method from the *ObjectOutputStream* class to write the book object to the file.

- TODO 6e: Print a message indicating that the book data was serialized.
- TODO 6f: Catch the *IOException* and print the stack trace.

This guide shows steps on how you can implement this code.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

```
// TODO 6a: Define a static void method named serializeBook with two parameters  
book of type Book and filePath of type String
```

```
public static void serializeBook(Book book, String filePath) {
```

```
    // TODO 6b: create a try-with-resources block
```

```

// TODO 6c: create a new FileOutputStream object with the filePath and a new
ObjectOutputStream object with the FileOutputStream object

    try (FileOutputStream fileOut = new FileOutputStream(filePath);

        ObjectOutputStream out = new ObjectOutputStream(fileOut)) {

        // TODO 6d: use the writeObject() method from the ObjectOutputStream class
        to write the book object to the file

        // TODO 6e: print a message indicating that the book data was serialized

    }


catch (IOException e) {

    // TODO 6f: catch the IOException and print the stack trace

}

}

```

-  **TODO 7:** Create the *deserializeBook* method in the *Book* class. This method will deserialize the *Book* object from the file specified by *filePath*.
 - **TODO 7a:** Define the *deserializeBook* method using the following guidelines:
 - Method name: *deserializeBook*.
 - Parameters: String *filePath*.
 - Return Type: *Book*.
 - **TODO 7b:** Inside the method, create a *Book* object and set it to *null*.
 - **TODO 7c:** Create a *try-with-resources* block
 - **TODO 7d:** Inside the *try-with-resources* parenthesis, create a new *FileInputStream* object with the *filePath* and a new *ObjectInputStream* object with the *FileInputStream* object.
 - **TODO 7e:** Inside the *try* block, read the object from the file and cast it to a *Book* object.

- TODO 7f: Print a message indicating that the book data was deserialized.
- TODO 7g: Catch the *IOException* and *ClassNotFoundException* and print the stack trace.
- TODO 7h: Return the *book* object.

Again, this sample code will guide you through implementing the method.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

```
// TODO 7a: Define a static method named deserializeBook with a single parameter
filePath of type String and a return type of Book

public static Book deserializeBook(String filePath) {

    // TODO 7b: create a Book object and set it to null


    // TODO 7c: Create a try-with-resources block


    // TODO 7d: Inside the try-with-resources parenthesis, create a new FileInputStream
    object with the filePath and a new ObjectInputStream object with the
    FileInputStream object

    try (FileInputStream fileIn = new FileInputStream(filePath);

        ObjectInputStream in = new ObjectInputStream(fileIn)) {

        // TODO 7e: read the object from the file and cast it to a Book object


        // TODO 7f: Print a message indicating that the book data was deserialized


    }

    catch (IOException | ClassNotFoundException e) {

        // TODO 7g: Catch the IOException and ClassNotFoundException and print the stack
        trace


    }

    // TODO 7h: Return the book object


    }
}
```

In the *main* method of the *Main* class, the code has already been written for you to test task 6. It includes creating a *Book* object, serializing it to a file, deserializing it back into an object, and printing its details. Just uncomment the code from lines 56 to 60.

Run Your Program:

When you run your program, it should display messages similar to the following:

1

2

3

```
Serialized book data to ./myBooks/book.ser
```

```
Deserialized book data from ./myBooks/book.ser
```

```
Deserialized Book: Book [title=1984, author=George Orwell, isbn=978-0451524935]
```

Troubleshooting

If you didn't get the expected output, try these troubleshooting steps:

1. Make sure the *filePath* is correct, and the directory exists. If not, create the directory or adjust the path.
2. Verify that the *Book* class implements *Serializable*. Without this, the object cannot be serialized.
3. Look for any *IOException* or *ClassNotFoundException*. Check file permissions, path correctness, and that the *Book* class hasn't changed since serialization.
4. Make sure the *try-with-resources* block is used correctly to manage file streams and close them properly.
5. Ensure the program prints the success messages for both serialization and deserialization. If not, check your *try-catch* blocks.

By checking these areas, you can resolve common issues with serialization and deserialization in your program.

After you have tested and confirmed your project functionality, if you are happy with your results you can now submit your assignment for grading and feedback! To submit your assignment, look for and click the blue “Submit Assignment” button in the top right corner of your lab environment. This sends your code off to the grader for evaluation.

When you submit your assignment, the grader is designed to check how well your code performs its intended functions based on the instructions for this assignment. Think of it as a thorough code review!

You can submit as many times as you'd like! Use the grader as a partner to help improve your code with every submission.

Here's the crucial part: the grader needs to compile and run your application in order to work. If it can't get past this stage, your grade will unfortunately be a 0. If you submit your work and the grade is a 0, there is a good chance that your hard work couldn't compile - not necessarily that your logic or functionality is wrong.

So double-check that everything compiles and runs correctly in the IDE before submitting.

Congratulations! By working through this lab, you demonstrated your ability to handle various file and directory operations in Java. You successfully implemented and tested key tasks such as listing and copying files, and creating, deleting, and renaming directories. You also incorporated additional functionality by effectively using serialization and deserialization. All of the tasks you completed successfully in this lab are critical for efficient file system management.

Today you worked with the fictional My Books Management System and enabled functions that would be required by potential purchasers Amelia and Lucas. However, the tasks that you completed here will apply to many other real-world applications. The vast majority of applications need some file-handling functionality. Your knowledge of file handling will therefore be an important asset in your career.

Great job. You've successfully put all you've learned in these lessons into practice but don't stop there. Continue working through the types of tasks you've stepped through in this activity to develop your skills and expertise.

After submitting your project for a grade, take a moment to review the Exemplar Solution. This sample solution can offer insights into different coding techniques and approaches. You can view the Exemplar Solution project in the Solution folder located here:

`/home/coder/coursera/Solution`

- Open the project in IntelliJ in a new window to see your project and the Exemplar Solution at the same time.
- Reflect on what you can learn from the Exemplar Solution to improve your coding skills.
- Remember, multiple solutions can exist for a problem, however the grader for this lab looking for one specific solution. The goal is to learn and grow as a programmer by exploring various approaches.

Use the Exemplar Solution as a learning tool to enhance your understanding and refine your approach to coding challenges.

Megjelölés elvégzettként