


Imagine someone who has a long commute by train to get to work. To pass the time, they watch movies on their mobile phone every day. Of course, they don't want to have to set the volume and playback speed every time they start to play the movie. As part of the team developing a new video player, you decide to use Java serialization as a practical way to preserve these settings across sessions in this interactive media environment. This lab gives you hands-on experience handling fundamental serialization scenarios.

## Goal

You'll develop a Java program where users can adjust their preferred volume level and playback speed for a video player. These settings should be saved when the application closes and restored upon reopening.

 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.

Some work has already been done on this program, so your first practical step is to familiarize yourself with what is already there.

Open your Java project named *VideoSettingsApp* in IntelliJ IDEA. This project includes two Java files:

- *VideoSettings.java*: This class manages the settings for video playback, specifically the volume level and playback speed. This class is serializable, meaning its state can be saved to a file and later restored.
- *VideoSettingsTest.java*: This class includes the *main* method, which orchestrates the serialization and deserialization processes to test the functionality of the *VideoSettings* class.

Open the *VideoSettings.java* file from the *src* folder and observe its contents. Identify the *serializeSettings* method. This is a static method that serializes a *VideoSettings* object to a specified file path, which means saving the state of a *VideoSettings* object to a file.

 It's time to get coding!

- TODO 1: Write the *VideoSettings* object to the *ObjectOutputStream*.

The primary purpose of serialization is to persist the state of an object so that it can be stored and later reconstructed. The first step to achieve this is to write the *VideoSettings* object to the *ObjectOutputStream*.

By writing the *VideoSettings* object to the *ObjectOutputStream* you convert the object into a byte stream that can be written to a file. This allows the object's state (including its data) to be saved in a persistent storage medium.

- TODO 2: If an *IOException* occurs, print an error message with the exception's message.

If an error occurs during the file-saving operation the user needs to be made aware of it so that they can understand why the operation failed. If an *IOException* occurs, handle it by asking Java to print an error message with the exception's message.

In the same *VideoSettings* class find the *deserializeSettings* method that deserializes a *VideoSettings* object from a specified file path.

- TODO 3: Read the *VideoSettings* object from the *ObjectInputStream* and cast it.

Deserialization is an essential step because it involves converting a byte stream back into a copy of the original object.

By reading the *VideoSettings* object from the *ObjectInputStream*, you can retrieve the serialized data and use it to reconstruct an instance of the *VideoSettings* class with the same state it had when it was serialized.

The data read from the *ObjectInputStream* includes all the fields of the *VideoSettings* object, such as *volumeLevel* and *playbackSpeed*. Casting it to *VideoSettings* ensures that the byte stream is correctly interpreted as an instance of the *VideoSettings* class, thereby restoring its state.

Okay, you've now saved the user's settings, ensuring they can be restored in future sessions through deserialization. Consider the next step: What happens if an error occurs during deserialization? Handling such scenarios in a user-friendly manner is crucial for your application.

- TODO 4: Return default settings in case of an error.

Returning to default settings in cases of error is an important aspect of making your program robust and user-friendly. Certain factors, such as a missing or corrupted file, can cause deserialization to fail. In that situation, returning default settings ensures that the program can continue to run.

Default settings act as a fallback mechanism to ensure the application has valid configuration values. This is especially important in scenarios where configuration settings are critical for the application's operation.

Complete TODO 3 and TODO 4 in the *deserializeSettings* method within the *VideoSettings* class.

Excellent work! After setting up your class and implementing the serialization and deserialization methods, your application should now use deserialization, which will provide the commuter on the train with the best possible user experience. The next crucial step is to test the *VideoSettings* class to ensure everything works as expected.

## Run your code using the IDE

To complete your testing, you can run the *VideoSettingsTest* class. The main method in *VideoSettingsTest* will orchestrate the serialization and deserialization processes.

1. Right-click on the file *VideoSettingsTest.java* in your IntelliJ IDEA.
2. Select "Run *VideoSettingsTest.main()*" to execute the test.
3. The console should print the serialized and deserialized settings, allowing you to verify if the serialization process preserved the object state accurately.

What result should you expect? Check for the print statements confirming successful serialization and deserialization.

1

```
Test passed: The loaded settings match the initial settings.
```

- If you get an *IOException*, check the file path, name, and extension provided to the stream objects in the try-with-resources blocks in the *VideoSettings* class.
- After deserializing the settings object, ensure that the correct class name is written for casting the settings object; otherwise, a *ClassCastException* will be thrown.

Great job! By completing this activity, you learned how to develop a Java program where users can adjust their preferred settings. Those settings are saved when the application closes and restored upon reopening.

Let's recap the steps you completed to achieve this. First, you successfully implemented and tested file operations in Java, specifically focused on copying files and handling serialization and deserialization of objects. You also gained practical experience in managing file I/O and object persistence by developing methods to convert String file paths to *Path* objects, copy files with error handling, and serialize and deserialize *VideoSettings* objects. Finally, this lab enhanced your understanding of Java's file-handling capabilities and reinforced the importance of robust error management. You were able to ensure that your application could gracefully handle unexpected issues and maintain consistent operation.

Well done! In a real-world project, your team leader would be happy with your contribution to the application. And the commuter watching videos on their way to work will have a much smoother and less stressful experience.