IJSE

Graduate Diploma in Software Engineering(GDSE)

Module Code : 1114

Advanced API Development

Serialization and Deserialization

Charaka V Mihiranga

2301682070

GDSE 68

SUBMISSION DATE (18/07/2024)

Table Of Contents

Glossary Of Terms

Serialization: The process of converting an object’s state into a byte stream, which can be saved to a file, sent over a network, or stored in a database.

Deserialization: The reverse process of serialization, converting a byte stream back into an object.

Serializable: An interface in Java that a class implements to indicate that its objects can be serialized.

serialVersionUID: A unique identifier for Serializable classes, used for version control of the serialized objects.

FileOutputStream: A class in Java used to create a file output stream to write data to a file.

ObjectOutputStream: A class in Java that serializes objects and writes them to an output stream.

FileInputStream: A class in Java used to create a file input stream to read data from a file.

ObjectInputStream: A class in Java that deserializes objects from an input stream.

Byte Stream: A sequence of bytes that represents the state of an object.

Metadata: Data that provides information about other data, such as the type and structure of an object during serialization.

Resource Leak: A situation where system resources are not released after use, potentially causing system performance issues.

Platform-independent: Describes a byte stream or format that can be used across different computing environments without modification.

Encryption: The process of converting data into a coded format to prevent unauthorized access during transmission or storage.

Serialization and Deserialization

3.1 Introduction

An object has three primary characteristics: identify, state and behavior.The state represents the value or data of the object**.**

Serialization is the process of converting an object's state to a byte stream. This byte stream can be save to a file,send over a network or stored in a database.The byte stream represents the object's state,which can later be reconstructed to create a new copy of the object.

Serialization allows us to save the data associated with an object and recreate the object in a new location.

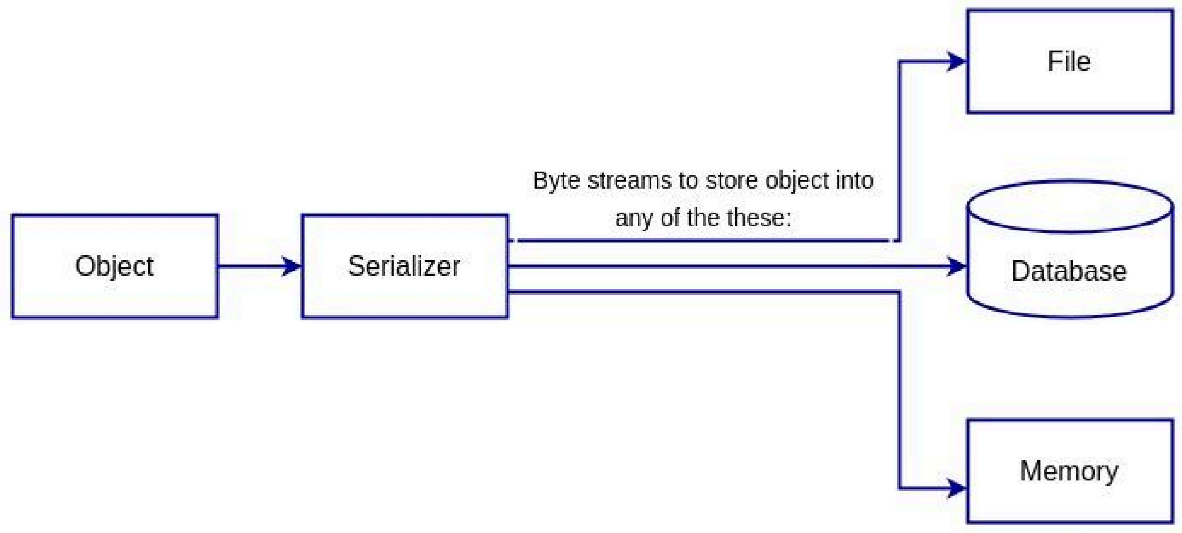


Figure 1 : Serialization

To serialize an object,the programmer must first decide on a format and then use the appropriate tools to convert the object to that format.

Deserialization is the reverse process of serialization.It involves taking a byte stream and converting it back into an object.This is done using the appropriate tool to parse the byte stream and create a new object.In java the *ObjectInputStream* class can be used to deserialize a binary format.

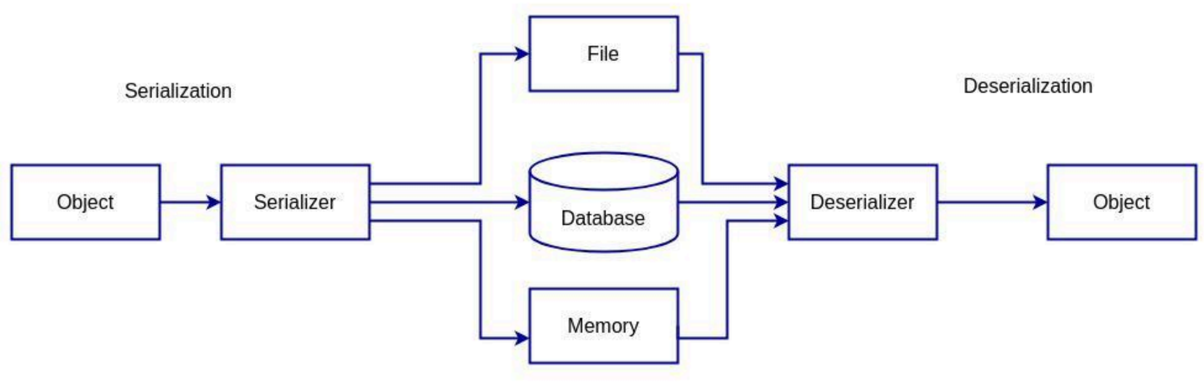


Figure 2 : Deserialization

3.2 Benefits of Serialization and Deserialization

Persistence :

Serialization allows object to be saved to dis or stored in a database,so their state can be retained between between different runs of an application

Ex: In a video game,the player's progress can be saved using serialization.when the player exits the game,the game state is saved.(player's position,score…etc).when the player resumes the game,the state is loaded back and they can continue from where they left off.

Data Transfer :

Serialization enables objects to be easily transmitted over a network

Ex: In a messaging app, when a user sends a message,it is serialized into a byte stream and -transmitted over a network.The recipient's app deserializes the byte stream back into a message object,displaying the message to the user.

Interoperability :

Serialized data can be exchanged between applications written in different programming languages.

Ex: A Java service can serialize data to JSON format and send it to a Python service. The Python service deserializes the JSON data into a Python object, processes it, and sends a response back in a similar serialized format.

Security :

Serialized data can be encrypted to protect it during transmission or storage

Ex: In an online banking system, user credentials can be serialized and encrypted before being sent over the network, ensuring that sensitive information remains protected from unauthorized access.

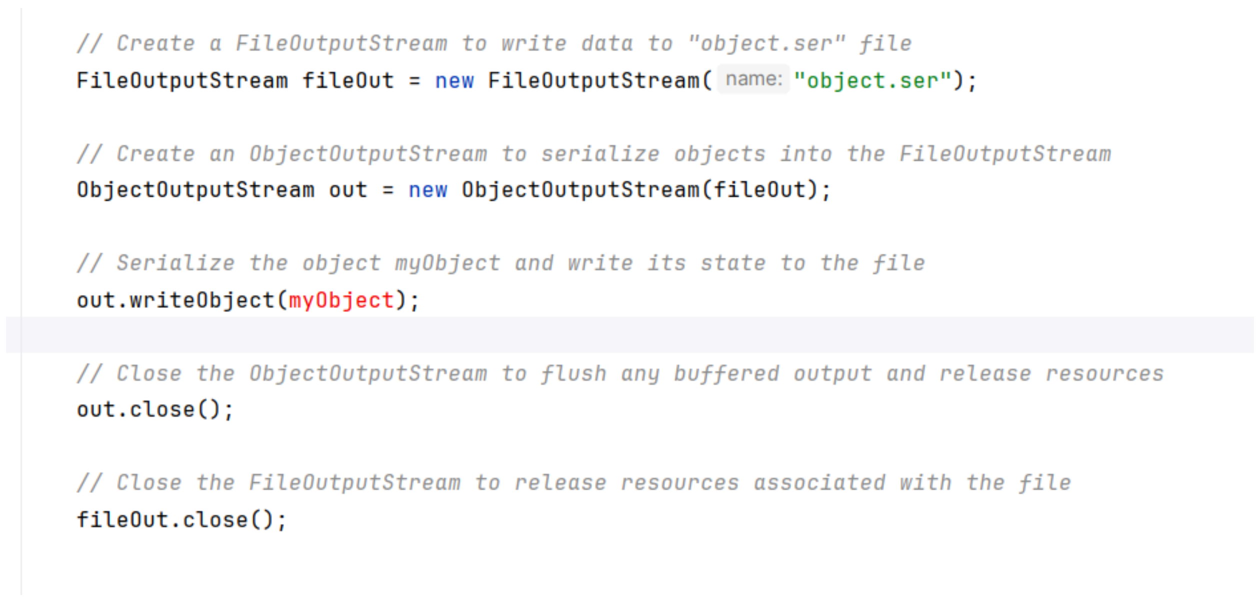
Efficiency :

Converting a byte stream to an object takes way less time than creating an actual object from a class.

Ex : In applications where frequent data exchange or message passing occurs, such as in network communications or distributed systems, using serialization for object creation can significantly improve performance and reduce processing overhead.

3.3 Mechanism of Serialization and Deserialization

Serialization



```java
// Create a FileOutputStream to write data to "object.ser" file
FileOutputStream fileOut = new FileOutputStream( name: "object.ser");

// Create an ObjectOutputStream to serialize objects into the FileOutputStream
ObjectOutputStream out = new ObjectOutputStream(fileOut);

// Serialize the object myObject and write its state to the file
out.writeObject(myObject);

// Close the ObjectOutputStream to flush any buffered output and release resources
out.close();

// Close the FileOutputStream to release resources associated with the file
fileOut.close();
```

* **FileOutputStream**: This class creates a file output stream to write data to a file named "object.ser". It establishes a connection between your program and the file system to enable the writing of byte streams. When you instantiate `FileOutputStream`, it prepares a file (creating one if it doesn't exist) to receive the serialized data.

* **ObjectOutputStream**: This class wraps around the `FileOutputStream` and is responsible for converting the Java object (`myObject`) into a sequence of bytes. This process, known as serialization, involves writing the object's state (its attributes and values) to the `FileOutputStream`. It includes metadata about the object's type and its internal structure. This byte stream is platform-independent, meaning it can be transferred across different environments.

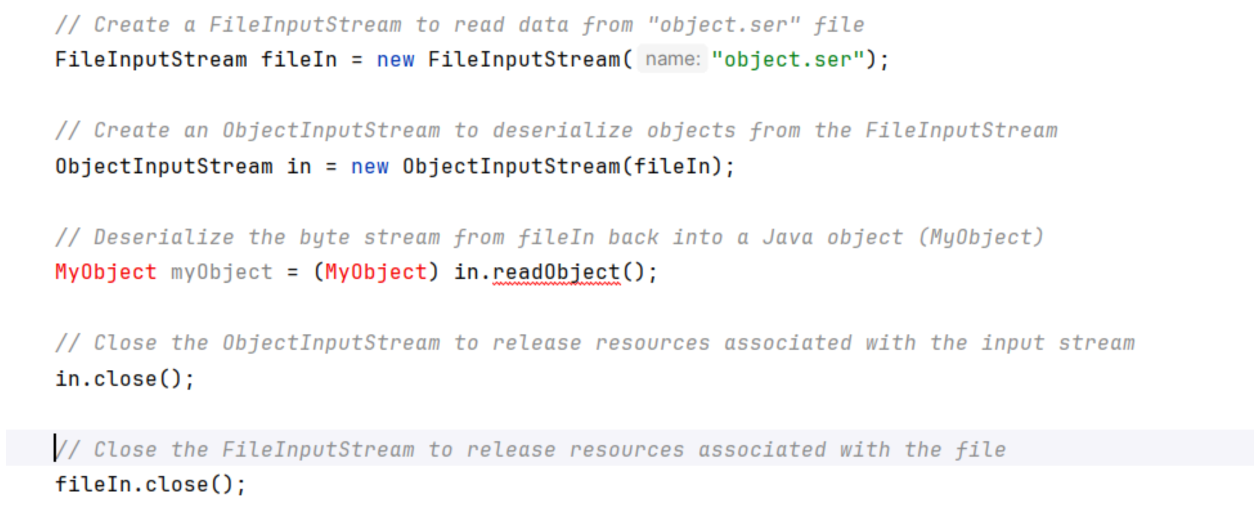
* **out.writeObject(myObject)**: The `writeObject` method of `ObjectOutputStream` serializes `myObject` and writes the resulting byte stream to the underlying file output stream (`fileOut`). During this process, the object's class metadata, fields, and references are all converted into a format suitable for storage or transmission.

* **out.close()**: Closing the `ObjectOutputStream` is essential as it ensures that all data is

properly flushed from the buffer to the underlying file stream (`fileOut`). This operation guarantees that the entire byte stream has been written to the file, and no part of the object's data is left unwritten. Additionally, it releases any resources associated with the stream, such as memory buffers.

* **fileOut.close()**: Finally, closing the `FileOutputStream` releases the file system resources associated with the file "object.ser". It is a good practice to close file streams to prevent resource leaks, which can occur when file handles remain open unnecessarily.

Deserialization



```
// Create a FileInputStream to read data from "object.ser" file
FileInputStream fileIn = new FileInputStream("object.ser");

// Create an ObjectInputStream to deserialize objects from the FileInputStream
ObjectInputStream in = new ObjectInputStream(fileIn);

// Deserialize the byte stream from fileIn back into a Java object (MyObject)
MyObject myObject = (MyObject) in.readObject();

// Close the ObjectInputStream to release resources associated with the input stream
in.close();

// Close the FileInputStream to release resources associated with the file
fileIn.close();
```

* **FileInputStream**: This class creates a file input stream to read data from a file named "object.ser". It establishes a connection between your program and the file system to enable reading of byte streams. When you instantiate `FileInputStream`, it prepares to read from the specified file.

* **ObjectInputStream**: This class wraps around the `FileInputStream` and is responsible for converting the byte stream back into a Java object. This process, known as deserialization, involves reading the object's state (its attributes and values) from the `FileInputStream`. The `ObjectInputStream` reads the metadata about the object's type and structure, allowing it to reconstruct the object accurately.

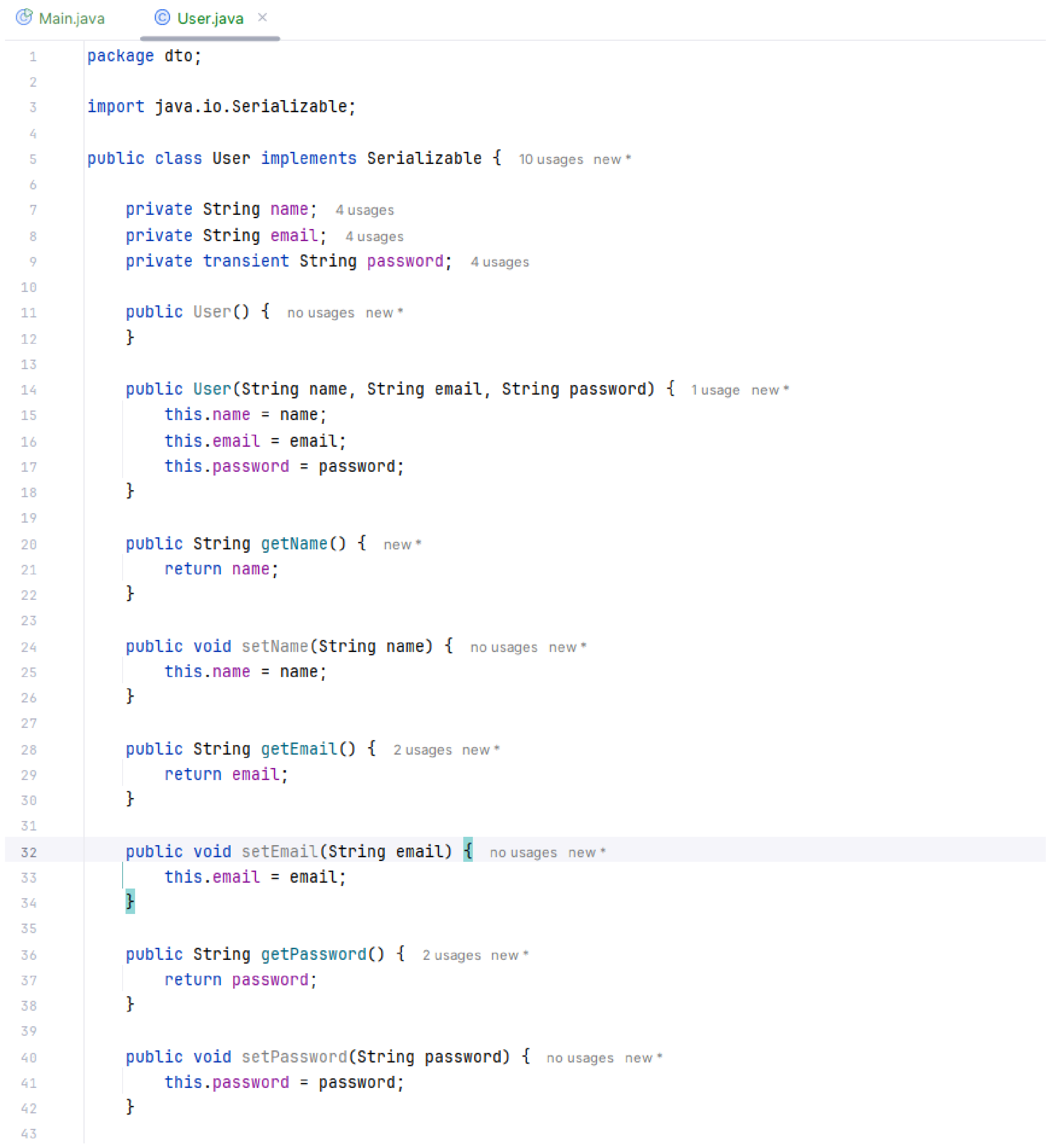
* **MyObject myObject = (MyObject) in.readObject()**: The `readObject` method of `ObjectInputStream` reads the byte stream from `fileIn` and interprets it as an object of type `MyObject`. This process reconstructs the object in memory, with all its attributes and values restored. The resulting object is cast to `MyObject` because `readObject` returns a generic `Object`.

* **in.close()**: Closing the `ObjectInputStream` releases resources associated with the input stream. This operation ensures that all data has been read from the file and the stream is properly closed, preventing resource leaks.

* **fileIn.close()**: Finally, closing the `FileInputStream` releases the file system resources associated with the file "object.ser". It is a good practice to close file streams to prevent resource leaks, which can occur when file handles remain open unnecessarily.

3.4 Example Code Snippet in Java

Imagine you are developing a user management system where you need to save the state of a user object (including their name, email, and password) to a file when the system shuts down and retrieve it when the system restarts.



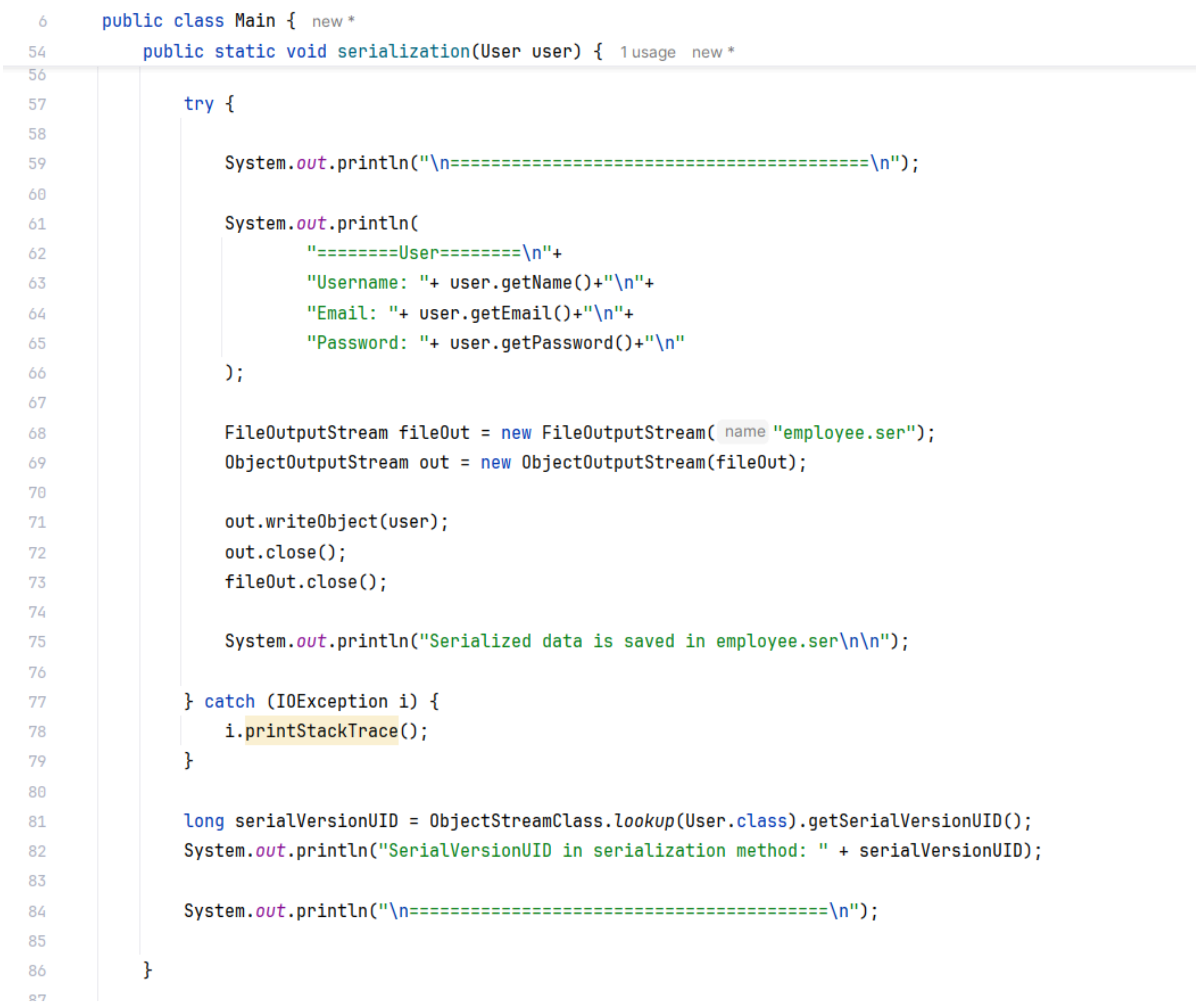
```java
package dto;

import java.io.Serializable;

public class User implements Serializable {

    private String name;
    private String email;
    private transient String password;

    public User() {
    }

    public User(String name, String email, String password) {
        this.name = name;
        this.email = email;
        this.password = password;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

```

User Class :

* The `User` class implements the `Serializable` interface, which allows its instances to be serialized.

* It has three fields: `name`, `email`, and `password`, along with a constructor and a `toString` method for easy display.

Main Class :



```java
public class Main {  new *
    public static void serialization(User user) {  1 usage  new *

        try {

            System.out.println("\n=========================================\n");

            System.out.println(
                    "========User========\n"+
                    "Username: "+ user.getName()+"\n"+
                    "Email: "+ user.getEmail()+"\n"+
                    "Password: "+ user.getPassword()+"\n"
            );

            FileOutputStream fileOut = new FileOutputStream( name "employee.ser");
            ObjectOutputStream out = new ObjectOutputStream(fileOut);

            out.writeObject(user);
            out.close();
            fileOut.close();

            System.out.println("Serialized data is saved in employee.ser\n\n");

        } catch (IOException i) {
            i.printStackTrace();
        }

        long serialVersionUID = ObjectStreamClass.lookup(User.class).getSerialVersionUID();
        System.out.println("SerialVersionUID in serialization method: " + serialVersionUID);

        System.out.println("\n=========================================\n");

    }
```

serialization Method:

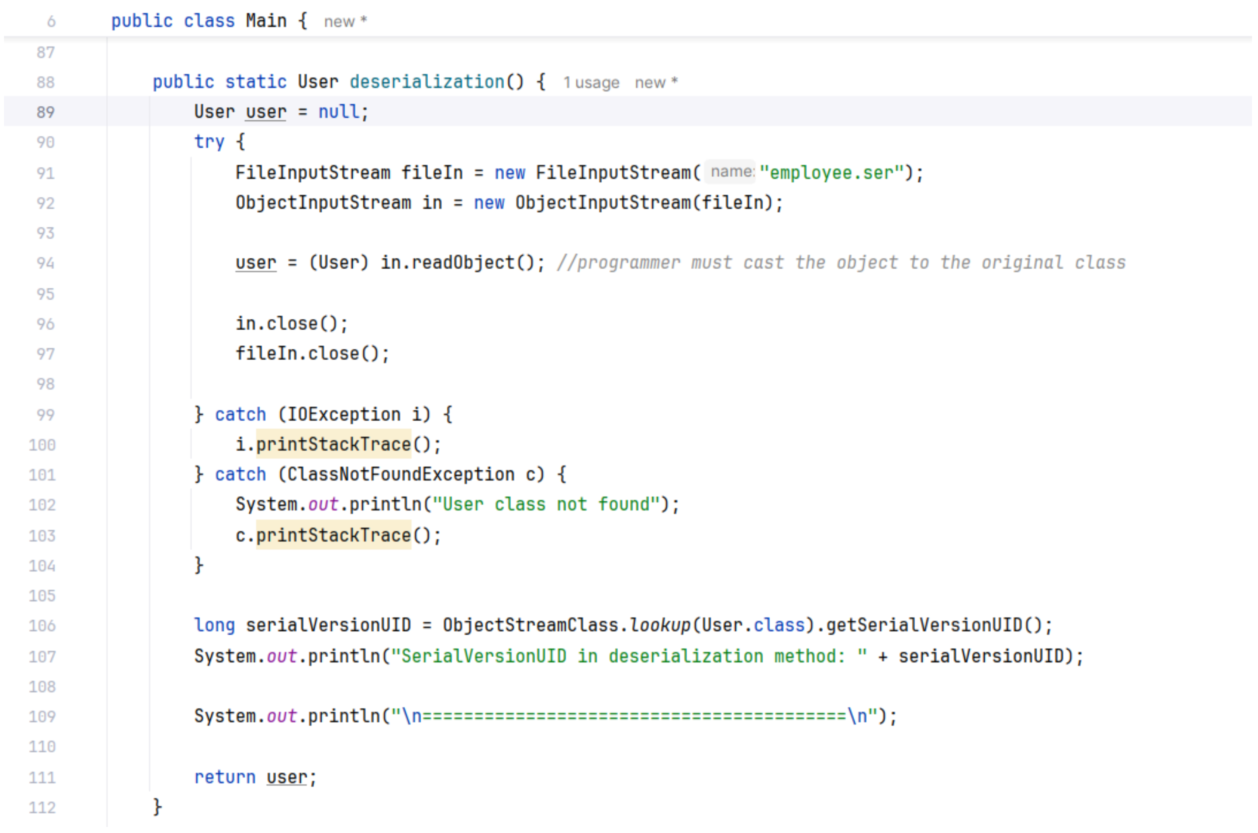
* **FileOutputStream fileOut = new FileOutputStream("employee.ser")**: Creates a stream to write data to a file named "employee.ser".

* **ObjectOutputStream out = new ObjectOutputStream(fileOut)**: Wraps the `FileOutputStream` with `ObjectOutputStream` to serialize the `User` object.

* **out.writeObject(user)**: Serializes the `user` object and writes its state to the file.

* **out.close()** and **fileOut.close()**: Closes the streams to flush the buffer and release resources.

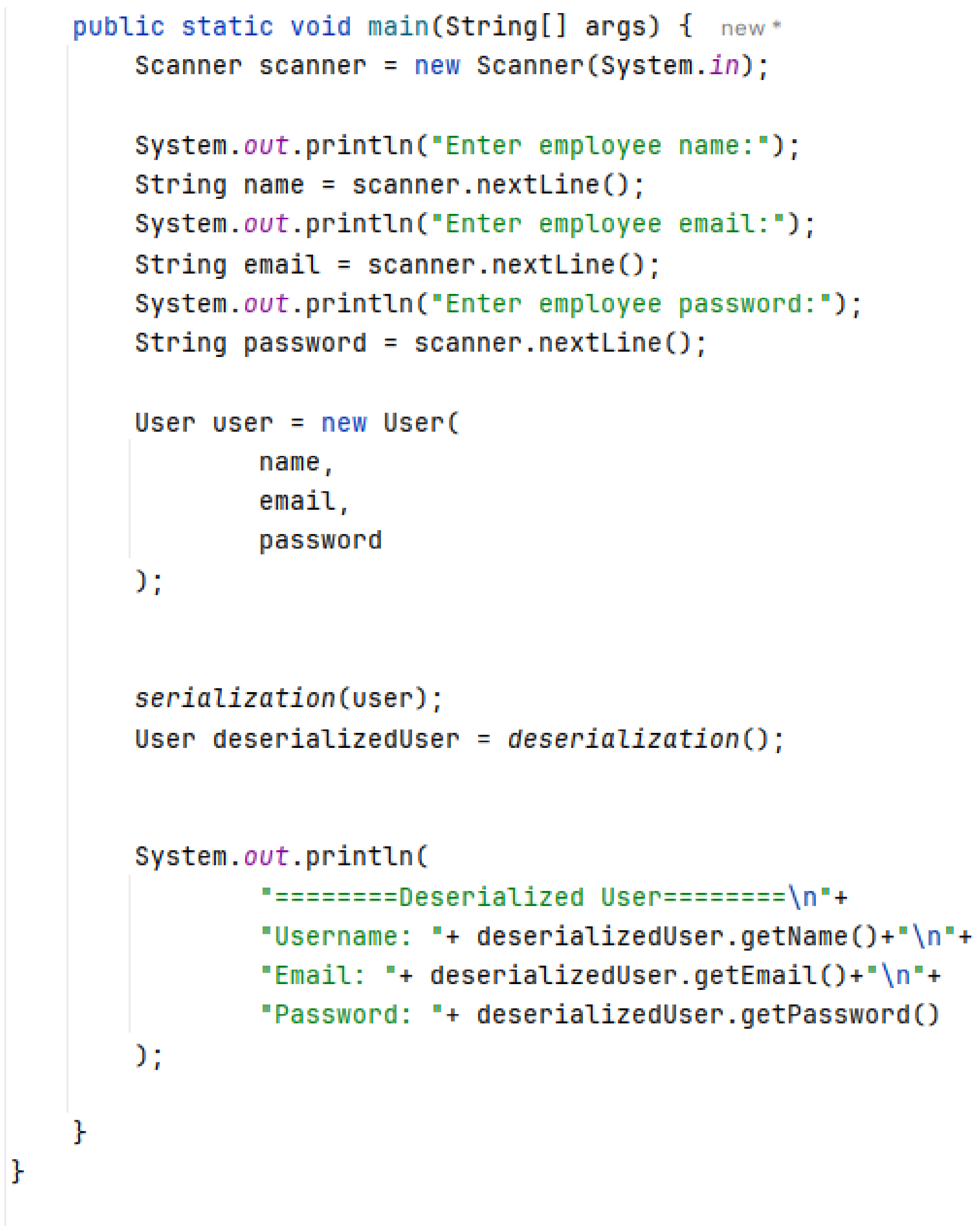
* **ObjectStreamClass.lookup(User.class).getSerialVersionUID()**: Retrieves the serialVersionUID of the `User` class and prints it.



```
public class Main {  new *

    public static User deserialization() {  1 usage  new *
        User user = null;
        try {
            FileInputStream fileIn = new FileInputStream( name: "employee.ser");
            ObjectInputStream in = new ObjectInputStream(fileIn);

            user = (User) in.readObject(); //programmer must cast the object to the original class

            in.close();
            fileIn.close();

        } catch (IOException i) {
            i.printStackTrace();
        } catch (ClassNotFoundException c) {
            System.out.println("User class not found");
            c.printStackTrace();
        }

        long serialVersionUID = ObjectStreamClass.lookup(User.class).getSerialVersionUID();
        System.out.println("SerialVersionUID in deserialization method: " + serialVersionUID);

        System.out.println("\n=========================================\n");

        return user;
    }
```

deserialization Method:

- **FileInputStream fileIn = new FileInputStream("employee.ser")**: Creates a stream to read data from the file "employee.ser".
- **ObjectInputStream in = new ObjectInputStream(fileIn)**: Wraps the `FileInputStream` with `ObjectInputStream` to deserialize the byte stream.
- **user = (User) in.readObject()**: Reads the byte stream from the file and reconstructs the `User` object.
- **in.close()** and **fileIn.close()**: Closes the streams to release resources.
- **ObjectStreamClass.lookup(User.class).getSerialVersionUID()**: Retrieves the serialVersionUID of the `User` class and prints it.



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

        System.out.println("Enter employee name:");
        String name = scanner.nextLine();
        System.out.println("Enter employee email:");
        String email = scanner.nextLine();
        System.out.println("Enter employee password:");
        String password = scanner.nextLine();

        User user = new User(
                name,
                email,
                password
        );


        serialization(user);
        User deserializedUser = deserialization();


        System.out.println(
                "========Deserialized User========\n"+
                "Username: "+ deserializedUser.getName()+"\n"+
                "Email: "+ deserializedUser.getEmail()+"\n"+
                "Password: "+ deserializedUser.getPassword()
        );

    }
}
```

main Method:

- Prompts the user to enter the employee's name, email, and password.
- Creates a `User` object with the entered data.
- Calls `serialization` to serialize the `User` object.
- Calls `deserialization` to deserialize the `User` object.
- Prints the deserialized `User` object to verify that the object's state is correctly restored.

Link to Code : https://github.com/CharakaMihiranga/Serialization-Report.git

Conclusion

In summary, serialization and deserialization are essential techniques in software development for managing the state of objects. Serialization enables the conversion of an object's state into a byte stream, which can be stored, transmitted, or shared across different platforms. Deserialization reverses this process, reconstructing the object from the byte stream. This functionality is crucial for various applications, including data persistence, network communication, and interoperability between different programming languages.

The example provided demonstrates how to implement serialization and deserialization in a user management system using Java. By serializing user objects to a file and deserializing them when needed, the system can effectively save and restore the state of user data. This capability ensures data consistency and integrity across application sessions.

The benefits of serialization and deserialization extend to improving application performance, enhancing security through data encryption, and facilitating efficient data transfer. These techniques are invaluable in scenarios such as saving game progress, transmitting messages in a chat application, and exchanging data between services in different programming languages.

By understanding and implementing serialization and deserialization, developers can create robust applications that handle data efficiently and securely, ensuring a seamless user experience and maintaining data integrity across different environments.

References

- Hazelcast (2024) *What is serialization and how does it work? | Hazelcast*. https://hazelcast.com/glossary/serialization/.
- Science, B. on C. and Science, B. on C. (2024) *What are serialization and deserialization in programming? | Baeldung on Computer Science*. https://www.baeldung.com/cs/serialization-deserialization.
- GeeksforGeeks (2023) *Serialization and Deserialization in Java with Example*. https://www.geeksforgeeks.org/serialization-in-java/.
- Hemant, K.M. (2023) 'Data Serialization and Deserialization: What is it?,' *Medium*, 30 November. https://medium.com/@khemanta/data-serialization-and-deserialization-what-is-it-29b5ca7a756f.
- Jain, S. (2022) 'serialization and deserialization - Scaler Topics,' *Scaler Topics*, 19 August. https://www.scaler.com/topics/java/serialization-and-deserialization/.
- Salvi, P. (2024) 'Serialization and Deserialization explained with examples,' *Medium*, 10 January. https://medium.com/@salvipriya97/serialization-and-deserialization-explained-with-examples-5e2e45af97ee.
- Hazelcast (2023) *What is deserialization and how does it work? | Hazelcast*. https://hazelcast.com/glossary/deserialization/.