TECHNICAL DESCRIPTION BASED ON RESEARCH & DEVELOPMENT GDSE 67

Serialization and Deserialization in Java

JULY 18, 2024

GDSE-67 | ITS1114 - ADVANCED API DEVELOPMENT

H.R.P.L.C.K. Siyambalapitiya | 2301671114

Contents

1. Introduction 2
2. Understanding Serialization And Deserialization 2
3. Implementing Serializable Interface 2
4. The Serialization Process 3
    1. FilesOutPutStream 3
    2. ObjectOutputStream 3
    3. Write Object 3
    4. .ser File 3
    5. Closing Streams 3

5. The Deserialization Process 4
    1. FileInputStream 4
    2. ObjectInputStream 4
    3. Read Object and Casting 4
    4. Closing Streams 4

6. Benefits of Serialization and Deserialization 5
7. Conclusion 6
8. Bibliography 6

1. Introduction

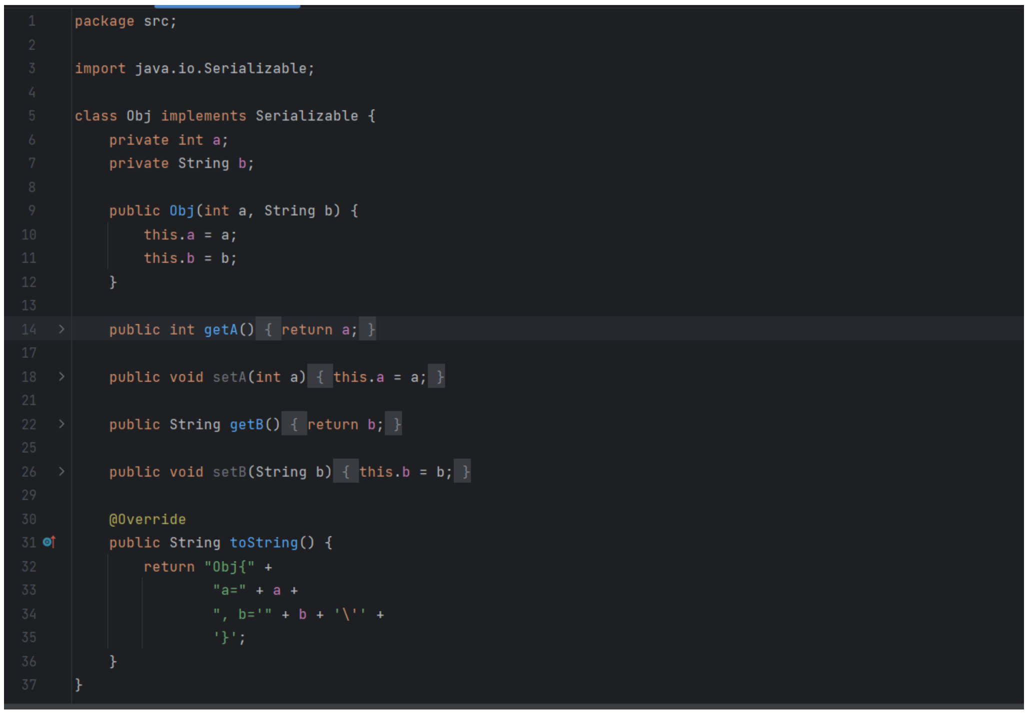
Serialization and deserialization are fundamental processes in Java programming that enable objects to be converted into byte streams for storage or transmission and reconstructed back into objects. These mechanisms are essential for data persistence, network communication, and efficient object management in Java applications.

2. Understanding Serialization And Deserialization

Serialization involves converting an object into a format that can be easily transported or stored, allowing objects to be saved as a sequence of bytes while maintaining their state and structure. Deserialization, on the other hand, reverses this process by using the byte stream to recreate the actual Java object in memory. This mechanism is crucial for persisting objects, enabling them to be stored persistently and reconstructed as needed during runtime.

3. Implementing Serializable Interface

In Java, objects that need to be serialized must implement the Serializable interface. This interface acts as a marker, indicating that the class is safe for serialization and does not contain any methods to implement.



```java
package src;

import java.io.Serializable;

class Obj implements Serializable {
    private int a;
    private String b;

    public Obj(int a, String b) {
        this.a = a;
        this.b = b;
    }

    public int getA() { return a; }

    public void setA(int a) { this.a = a; }

    public String getB() { return b; }

    public void setB(String b) { this.b = b; }

    @Override
    public String toString() {
        return "Obj{" +
                "a=" + a +
                ", b='" + b + '\'' +
                '}';
    }
}
```

4. The Serialization Process

Serialization in Java involves converting an object into a byte stream so that it can be easily transported or stored. This process ensures that the object's state and structure are preserved. Here are the steps involved in the serialization process, along with an example code snippet to illustrate each step,

Steps involved,

1. **FilesOutPutStream**

The FileOutputStream class is used to create a file and write bytes to it. This file will store the serialized object.

2. **ObjectOutputStream**

The ObjectOutputStream class is used to serialize the object. It converts the object into a byte stream and writes it to the FileOutputStream.

3. **Write Object**

The writeObject() method of ObjectOutputStream writes the object to the stream. This method serializes the object and all objects it references, recursively.

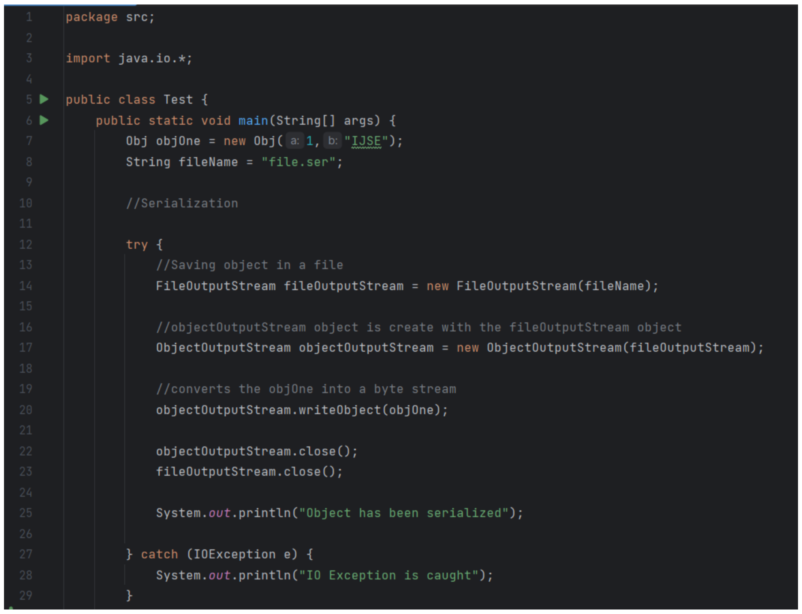
4. **.ser File**

After serialization, a .ser file is created. This file contains the serialized data of the object, which can be stored or transported.

5. **Closing Streams**

It is important to close the streams (ObjectOutputStream and FileOutputStream) after serialization to release system resources and avoid memory leaks.

Code Example:



```java
package src;

import java.io.*;

public class Test {
    public static void main(String[] args) {
        Obj objOne = new Obj( a: 1, b: "IJSE");
        String fileName = "file.ser";

        //Serialization

        try {
            //Saving object in a file
            FileOutputStream fileOutputStream = new FileOutputStream(fileName);

            //objectOutputStream object is create with the fileOutputStream object
            ObjectOutputStream objectOutputStream = new ObjectOutputStream(fileOutputStream);

            //converts the objOne into a byte stream
            objectOutputStream.writeObject(objOne);

            objectOutputStream.close();
            fileOutputStream.close();

            System.out.println("Object has been serialized");

        } catch (IOException e) {
            System.out.println("IO Exception is caught");
        }
```

5. The Deserialization Process

Deserialization is the process of converting a byte stream back into a replica of the original object in memory. This process allows the stored or transmitted object to be reconstructed and used as if it were the original object.

Steps Involved,

1. **FileInputStream**

Reads bytes from the file where the serialized object is stored.

2. **ObjectInputStream**

Deserializes the byte stream into an object by reading the data from the FileInputStream.

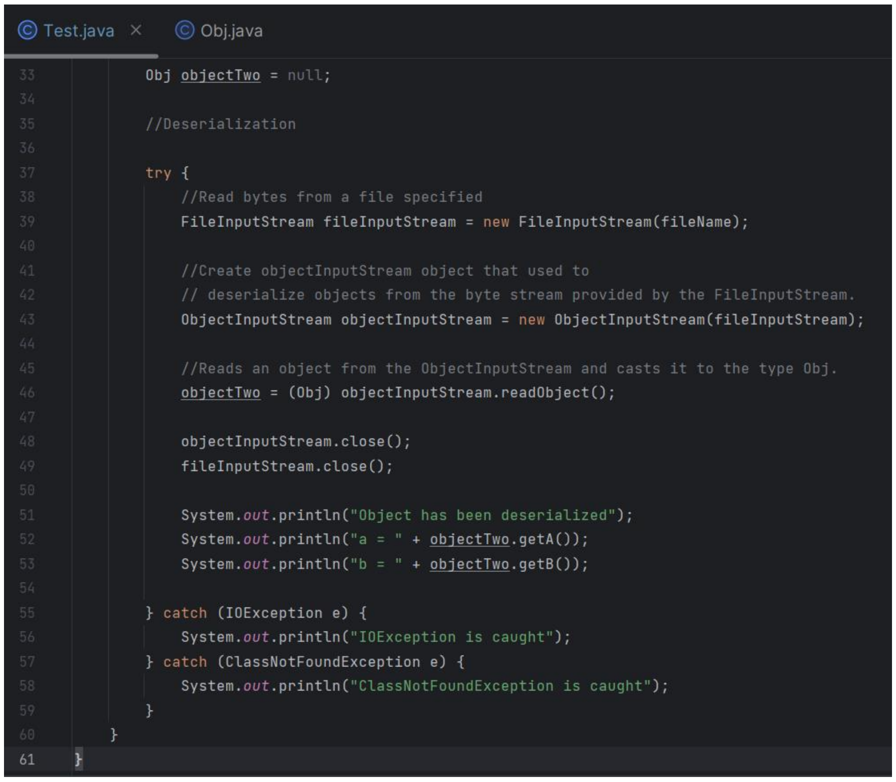
3. **Read Object and Casting**

The readObject() method of ObjectInputStream reads the object from the stream, and casting is necessary to convert the generic Object type to the specific object type.

4. **Closing Streams**

Closing the streams (ObjectInputStream and FileInputStream) after deserialization is crucial to release resources and avoid memory leaks.

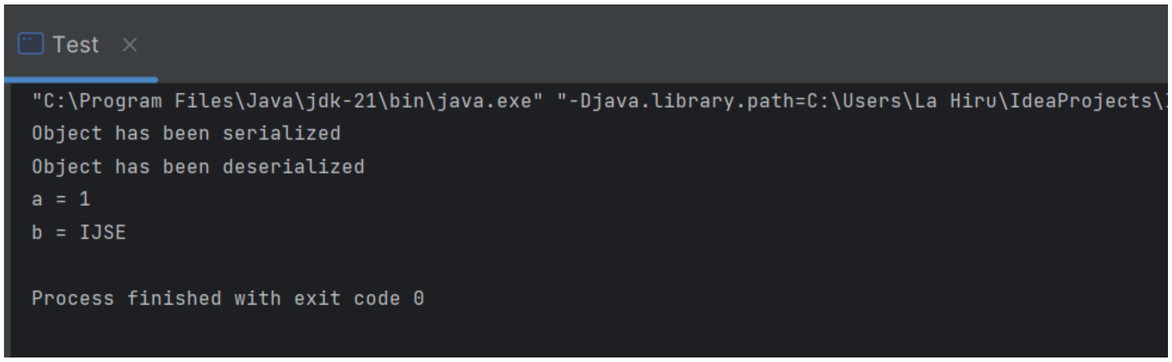
Code Example:



```java
        Obj objectTwo = null;

        //Deserialization

        try {
            //Read bytes from a file specified
            FileInputStream fileInputStream = new FileInputStream(fileName);

            //Create objectInputStream object that used to
            // deserialize objects from the byte stream provided by the FileInputStream.
            ObjectInputStream objectInputStream = new ObjectInputStream(fileInputStream);

            //Reads an object from the ObjectInputStream and casts it to the type Obj.
            objectTwo = (Obj) objectInputStream.readObject();

            objectInputStream.close();
            fileInputStream.close();

            System.out.println("Object has been deserialized");
            System.out.println("a = " + objectTwo.getA());
            System.out.println("b = " + objectTwo.getB());

        } catch (IOException e) {
            System.out.println("IOException is caught");
        } catch (ClassNotFoundException e) {
            System.out.println("ClassNotFoundException is caught");
        }
    }
}
```

Output Explanation:

When the provided code is executed, the following output is produced.



```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-Djava.library.path=C:\Users\La Hiru\IdeaProjects\
Object has been serialized
Object has been deserialized
a = 1
b = IJSE

Process finished with exit code 0
```

When running the provided Java code, the output sequence confirms the successful execution of both serialization and deserialization processes. Initially, upon serializing the objOne object, the program outputs "Object has been serialized", signifying the successful conversion of objOne into a byte stream and its storage in the file.ser file. Subsequently, during deserialization, the program outputs "Object has been deserialized", indicating that the byte stream from file.ser was effectively read and reconstructed into a new Obj object named objectTwo. The subsequent outputs "a = 1" and "b = IJSE" demonstrate that the deserialization process accurately restores the values of a (set to 1) and b (set to "IJSE") from the serialized object objOne. This process showcases the reliability and effectiveness of Java's serialization and deserialization mechanisms in maintaining object state across different stages of data storage and retrieval.

6. Benefits of Serialization and Deserialization

Serialization and deserialization offer several advantages,

- **Persistence** Objects can be saved and retrieved from storage, maintaining their state across program executions.
- **Communication**: Serialized objects can be transmitted between different applications or systems over networks.
- **Caching**: Serialized objects can be stored in memory caches, enhancing application performance by reducing object creation overhead.
- **Platform Independence**: Serialized objects are platform-independent, allowing interoperability across different operating systems and architectures.
- **Security**: Serialized data can be encrypted for secure transmission and storage, ensuring data integrity and confidentiality.

7. Conclusion

Serialization and deserialization are integral processes in Java that enable developers to manage object state effectively, facilitate seamless data exchange between distributed systems, and ensure data persistence and security. By implementing the Serializable interface, Java classes can be serialized and deserialized, allowing for efficient storage, transmission, and recreation of object instances. These mechanisms are foundational for building robust, scalable, and efficient Java applications.

8. Bibliography

https://www.geeksforgeeks.org/. (2023, 10 27). Retrieved from https://www.geeksforgeeks.org/serialization-in-java/: https://www.geeksforgeeks.org/serialization-in-java/