Chapter 20: Serialization and Deserialization

Introduction

Serialization and Deserialization are foundational concepts in Java and many modern programming languages, particularly important for persistent storage, remote method invocation, and distributed computing. Serialization is the process of converting an object into a byte stream, while deserialization is the reverse process – reconstructing the object from the byte stream. These processes are essential when an object needs to be transferred over a network or saved to a file or database.

In this chapter, we will explore the internal mechanisms of serialization and deserialization in Java, understand how to make classes serializable, and discuss custom serialization, transient keywords, versioning, and associated design concerns. This knowledge is critical for building scalable, secure, and efficient distributed systems.

20.1 What is Serialization?

Serialization is the process of converting the state of an object into a byte stream so that the byte stream can be reverted back into a copy of the original object.

- **Purpose of Serialization**:
    - Save the state of an object to a storage medium.
    - Send an object over a network using sockets or RMI.
    - Cache objects.
    - Clone or deep copy objects.
- **Typical use cases**:
    - Java RMI (Remote Method Invocation).
    - Hibernate / JPA (object persistence frameworks).
    - Android bundle data passing.
    - Distributed systems and microservices.

20.2 Java Serialization API

Java provides built-in support for serialization via the `java.io.Serializable` interface.

20.2.1 Serializable Interface

```
public interface Serializable {
}
```

- Marker interface (contains no methods).
- Its presence informs the JVM that a class is eligible for serialization.
- All fields of the class must also be serializable (either primitive or also implementing `Serializable`).

20.3 Basic Serialization Example

```
import java.io.*;

class Student implements Serializable {
    int id;
    String name;

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
}

public class SerializeDemo {
    public static void main(String[] args) throws Exception {
        Student s1 = new Student(1, "Rahul");

        FileOutputStream fout = new FileOutputStream("student.ser");
        ObjectOutputStream out = new ObjectOutputStream(fout);

        out.writeObject(s1);
        out.close();
        fout.close();

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

20.4 Deserialization

Deserialization is the process of reconstructing the object from its serialized byte stream.

Example

```
import java.io.*;

public class DeserializeDemo {
    public static void main(String[] args) throws Exception {
        FileInputStream fin = new FileInputStream("student.ser");
        ObjectInputStream in = new ObjectInputStream(fin);
```

```
        Student s = (Student) in.readObject();
        in.close();
        fin.close();

        System.out.println("Deserialized Student: " + s.id + ", " + s.name);
    }
}
```

20.5 The `transient` Keyword

If you don't want a field to be serialized, you can declare it as `transient`.

```
class User implements Serializable {
    String username;
    transient String password;  // will not be serialized
}
```

- Useful for sensitive information (passwords, security tokens).
- Fields marked `transient` are initialized to default values during deserialization.

20.6 serialVersionUID

The `serialVersionUID` is a unique version identifier for a class. It ensures that a loaded class corresponds exactly to a serialized object.

```
private static final long serialVersionUID = 1L;
```

- Required to avoid `InvalidClassException` during deserialization.
- Helps in version control if the class definition changes.

20.7 Custom Serialization with `Externalizable`

`Externalizable` interface allows control over serialization logic.

```
public interface Externalizable extends Serializable {
    void writeExternal(ObjectOutput out) throws IOException;
    void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException;
}
```

Example

```
class Employee implements Externalizable {
    int id;
    String name;
```

```
    public Employee() {}

    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeInt(id);
        out.writeObject(name);
    }

    public void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException {
        id = in.readInt();
        name = (String) in.readObject();
    }
}
```

20.8 Serialization of Object Graphs

When an object contains references to other objects, and they all implement `Serializable`, the entire object graph is serialized automatically.

- Cycles in the graph are handled using a reference table internally.
- Non-serializable objects in the graph will throw `NotSerializableException`.

20.9 Serializing Collections and Arrays

- Most Java collections like `ArrayList`, `HashMap`, etc., are serializable.
- Custom objects stored in collections must also implement `Serializable`.

```
List<Student> list = new ArrayList<>();
list.add(new Student(1, "A"));
list.add(new Student(2, "B"));
```

- Use `ObjectOutputStream` to serialize the entire list.

20.10 Limitations of Java Serialization

- **Platform dependent**: Not ideal for cross-language communication.
- **Security**: Vulnerable to deserialization attacks (e.g., remote code execution).
- **Performance**: Java serialization is slower than alternatives like Protocol Buffers, JSON, etc.
- **Versioning issues**: Changing class structure can break deserialization unless handled carefully.

20.11 Best Practices

- Always declare `serialVersionUID`.
- Avoid serializing sensitive fields (use `transient`).
- Prefer custom serialization (`Externalizable`) for performance and control.
- Use alternatives (JSON, XML, ProtoBuf) in microservices or public APIs.
- Validate deserialized objects to prevent injection attacks.

20.12 Alternatives to Java Serialization

Format	Use Case	Language Agnostic	Speed	Schema
JSON	Web APIs, config files	✓	Moderate	✗
XML	Configurations, legacy systems	✓	Slow	✗
Protocol Buffers	Efficient binary serialization (Google)	✓	Very fast	✓
Kryo	High-performance Java serialization	✗	Very fast	✗
Avro	Big Data (Apache Hadoop)	✓	Fast	✓

Summary

Serialization and Deserialization play a vital role in persistent data storage and distributed systems. Java provides a robust and extensible API for object serialization through the `Serializable` and `Externalizable` interfaces. Understanding how serialization works, managing class evolution, and securing serialized data are essential skills for advanced developers.

With modern requirements demanding security and performance, Java's default serialization might not always be the best fit, and alternatives like JSON, Protocol Buffers, or custom formats are often preferred.
