**Serialization**

Serialization in Java is the process of **converting an object into a byte stream.**

**After Serialization , the object can be :**

1. **Stored in a file or database** for persistence.
2. **Sent over a network** to another system (e.g., for remote communication).
3. **Reconstructed later** (deserialization) to restore the object’s state.

It is useful in **distributed systems**, caching, and logging.

**How to Serialize?**

**1. Modify Your Class to Implement Serializable**

Java provides java.io.Serializable as a marker interface to enable serialization.

import java.io.Serializable;

public class Student implements Serializable {

private static final long serialVersionUID = 1L; // Recommended for version control

private String id;

private String name;

private String email;

private String city;

private int year;

public Student(String id, String name, String email, String city, int year) {

this.id = id;

this.name = name;

this.email = email;

this.city = city;

this.year = year;

}

// Getters & Setters (Optional)

@Override

public String toString() {

return "Student{id='" + id + "', name='" + name + "', email='" + email +

"', city='" + city + "', year=" + year + "}";

}

}

#### ****2. Serialize the Object to a File****

Now, modify your test to serialize the object.

import java.io.FileOutputStream;

import java.io.ObjectOutputStream;

public class SerializeExample {

public static void main(String[] args) {

Student s = new Student("3000", "Amith", "Amith@gmail.com", "Hyderabad", 2021);

try (FileOutputStream fileOut = new FileOutputStream("student.ser");

ObjectOutputStream out = new ObjectOutputStream(fileOut)) {

out.writeObject(s); // Serializing the object

System.out.println("Serialized data is saved in student.ser");

} catch (Exception e) {

e.printStackTrace();

}

}

}

#### ****3. Deserialize the Object****

To **retrieve** the object, use the following code:

import java.io.FileInputStream;

import java.io.ObjectInputStream;

public class DeserializeExample {

public static void main(String[] args) {

Student s = null;

try (FileInputStream fileIn = new FileInputStream("student.ser");

ObjectInputStream in = new ObjectInputStream(fileIn)) {

s = (Student) in.readObject(); // Deserializing the object

System.out.println("Deserialized Student: " + s);

} catch (Exception e) {

e.printStackTrace();

}

}

}

### **Why Use Serialization?**

In your case, serialization can be used for:

* **Saving student records in a file** for backup.
* **Sending objects over a network** if needed.
* **Storing objects in a cache** for faster retrieval.

### **Who Will Deserialize the Object?**

Deserialization is needed when you want to **retrieve and use** the serialized object later. The **consumer** of the serialized data is responsible for deserializing it.

Here are **three common scenarios** where deserialization is used:

1. **When Reading from a File** (e.g., restoring saved data)
2. **When Sending Over a Network** (e.g., Remote Method Invocation - RMI)
3. **When Caching Objects** (e.g., Redis, Ehcache)

**Serialization and Its Relation to Servlets**

Serialization is crucial in servlets for handling **session management, data transfer, and inter-process communication**. Below are **three key areas** where serialization is used in servlets:

1. Session Management (Storing Objects in HTTP Session)
2. **Sending Java Objects in Servlet Response (Serialization in API Responses)**
3. Storing Servlet Data in Files (Serialization for Persistence)

**Sending Java Objects in Servlet Response (Serialization in API Responses):**

In **servlets and REST APIs**, Java objects cannot be sent directly in HTTP responses because **HTTP is a text-based protocol** that does not understand Java objects. Therefore, **serialization** is required to convert Java objects into a format that can be transferred over the network, such as **JSON or XML**.

## **HTTP Only Understands Text and Binary Formats**

* HTTP cannot **directly send Java objects**; it only supports text-based data formats like **JSON, XML, HTML, plain text**, or binary formats like images and PDFs.

## **Interoperability Between Different Systems**

* If your servlet is part of a **microservices architecture** or is being accessed by a **frontend app (React, Angular, Vue, etc.)**, serialization ensures compatibility.
* Java objects must be **converted to JSON/XML** so that they can be **understood by non-Java systems** (JavaScript, Python, .NET, etc.).

## **REST API Example: Sending JSON Response from Servlet**

In a **servlet-based REST API**, we can serialize Java objects into JSON and send them as responses.

## **If You Don't Serialize, You Get Errors!**

If you try to send a **Java object directly** in an HTTP response **without serialization**, the servlet will throw an **error** because HTTP **cannot understand Java objects**.

### **Deserialization: When, Who, and How?**

After a Java object is **serialized** into JSON and sent over the network, the **receiving system (client or another server)** must **deserialize** it back into a Java object (or another format like JavaScript, Python object, etc.).

## **When Does Deserialization Happen?**

Deserialization occurs when:

1. **A client (browser, frontend app, mobile app) receives JSON from a Java server**
   * Example: A React, Angular, or Android app gets JSON from a Java servlet.
2. **A Java-based consumer service receives JSON from another microservice**
   * Example: In a microservices architecture, Service A sends JSON to Service B, and Service B deserializes it.
3. **Another backend (Python, Node.js, .NET) processes JSON from Java**
   * Example: A Python or .NET application consumes a JSON API from a Java backend.