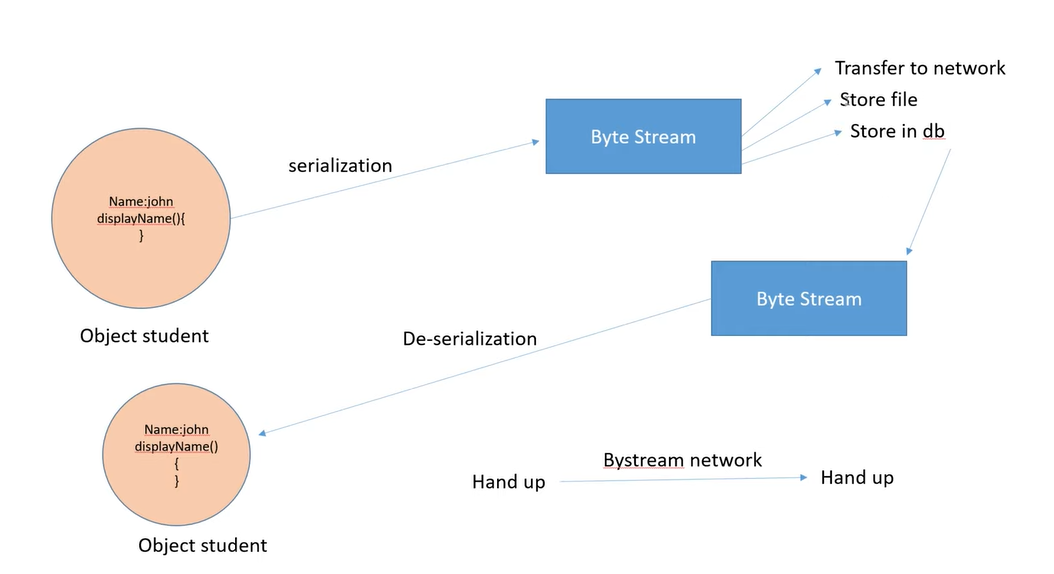
**Serialiation and Deserialization:[JavaConcepts]**



In Java, the **Serializable** interface is used to mark a class so that its objects can be **converted into a byte stream**. This process is known as **serialization**. When a class implements the Serializable interface, it indicates that the objects of that class can be serialized and deserialized, which is helpful for saving objects to a file, sending them over a network, or caching them.

**Key Concepts of Serializable**

1. **Serialization**: The process of converting an object into a byte stream, which can then be stored in a file, transferred over a network, or saved in memory for future use.
2. **Deserialization**: The reverse process of converting a byte stream back into a copy of the original object, reconstructing the object's state.

**How Does Serializable Work?**

* The Serializable interface is a **marker interface**, meaning it doesn’t have any methods. It serves as a signal to the Java Virtual Machine (JVM) and ObjectOutputStream that the class can be serialized.
* When an object is serialized, its state (the values of its fields) is converted into a byte stream.
* When the object is deserialized, the byte stream is used to recreate the object in memory, retaining its original state.

**Why Use Serialization?**

Serialization is useful for a variety of scenarios, such as:

1. **Persistence**: Saving the state of an object to a file or database so that it can be restored later.
2. **Communication**: Transferring objects over a network, for example, in Remote Method Invocation (RMI) or socket programming.
3. **Caching**: Storing the state of objects in memory for performance reasons.
4. **Deep Cloning**: Creating deep copies of objects by serializing and deserializing them.

**How to Make a Class Serializable**

To make a class serializable, the class must implement the Serializable interface.

**Explanation of the Example:**

1. **Employee Class**: The Employee class implements the Serializable interface, making its objects eligible for serialization.
2. **Serialization**:
   * The ObjectOutputStream is used to write the employee object to a file named employee.txt.
   * This process converts the object into a byte stream and saves it to the file.
3. **Deserialization**:
   * The ObjectInputStream reads the employee.txt file and converts the byte stream back into an Employee object.
   * The deserialized object retains the same state as the original.

**Important Considerations for Serialization**

1. **serialVersionUID**:
   * It’s recommended to define a serialVersionUID field in a serializable class.
   * This is used to ensure version compatibility during the deserialization process. If the class definition changes and no matching serialVersionUID is found, a InvalidClassException is thrown.
   * Example:

private static final long serialVersionUID = 1L;

1. **Transient Fields**:
   * Fields marked with the transient keyword are **not serialized**.
   * Use transient for sensitive information (like passwords) or fields that can be recalculated.
   * Example:

private transient String password; // This field will not be saved during serialization

1. **Static Fields**:
   * Static fields belong to the class, not to an instance, and hence are **not serialized**.
   * Only the instance fields of an object are serialized.
2. **Inheritance**:
   * If a superclass implements Serializable, then all its subclasses are automatically serializable.
   * If a class doesn’t implement Serializable, any attempt to serialize its object will result in a NotSerializableException.
3. **Customization**:
   * You can customize the serialization and deserialization processes by defining special methods in your class:
     + private void writeObject(ObjectOutputStream oos) throws IOException
     + private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException

**Use Cases of Serialization**

* **Storing User Sessions**: Save and retrieve the state of user sessions in web applications.
* **Sending Objects Over a Network**: Transfer objects in distributed applications (e.g., RMI, message queues).
* **Deep Cloning**: Create deep copies of objects by serializing and deserializing them.
* **Caching**: Store and retrieve cached objects in memory.

**Limitations of Serialization**

* **Performance Overhead**: Serialization can introduce overhead in terms of time and memory usage.
* **Security Risks**: Deserializing untrusted data can lead to security vulnerabilities (e.g., arbitrary code execution).
* **Versioning Issues**: Changes to class definitions can cause InvalidClassException if serialVersionUID is not managed properly.

**Summary**

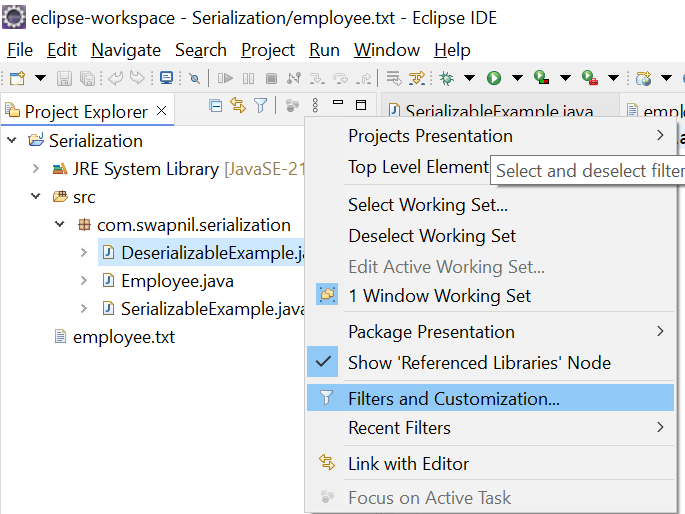
* The Serializable interface allows an object’s state to be converted into a byte stream and vice versa.
* Serialization is useful for saving objects to files, sending objects over a network, or deep cloning.
* Implementing Serializable is straightforward but should be used cautiously due to performance, security, and version compatibility concerns.
* If we change serialVersionUID after executing SerializableExample.java and then run DeserializableExample.java then it will give exception(runtime)

java.io.InvalidClassException: com.swapnil.serialization.Employee; local class incompatible: stream classdesc serialVersionUID = 2, local class serialVersionUID = 1

* If we remove Serializable from class then it will give exception (runtime)

java.io.NotSerializableException: com.swapnil.serialization.Employee

* If we remove ClassNotFoundException then it will give exception



Uncheck .\* resources so that files are visible in eclipse project explorer.

**POJO**

A **POJO** (Plain Old Java Object) is a simple Java object that does not follow any special restrictions other than those forced by the Java language specification. It’s used to create lightweight Java objects that primarily encapsulate data, without adding complex functionality or behavior. The term was coined as a way to emphasize that these objects are simple and not bound to any framework-specific conventions.

**Characteristics of a POJO**

1. **No Special Inheritance or Annotations**: A POJO does not implement any special interfaces (like Serializable or Remote) or extend any framework-specific base classes.
2. **Contains Only Fields and Methods**: A POJO usually contains private fields and provides public getter and setter methods to access or modify these fields.
3. **No Business Logic**: Typically, a POJO only holds data and does not contain any business logic.
4. **Easily Readable and Writable**: POJOs are easy to create, read, and maintain due to their simplicity.

**When to Use a POJO?**

POJOs are primarily used to represent data or as model objects in applications. They are often used in frameworks like Hibernate or Spring, where they serve as data carriers or entities that map to database tables or external data sources.

**POJO Example**

Let’s create a simple Employee POJO in Java:

// Employee.java - A simple POJO class

public class Employee {

// Private fields (attributes)

private int id;

private String name;

private String department;

// Default constructor (no-argument constructor)

public Employee() {}

// Parameterized constructor

public Employee(int id, String name, String department) {

this.id = id;

this.name = name;

this.department = department;

}

// Getter and Setter methods for each field

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getDepartment() {

return department;

}

public void setDepartment(String department) {

this.department = department;

}

// Optional: Override the toString() method for better readability

@Override

public String toString() {

return "Employee [id=" + id + ", name=" + name + ", department=" + department + "]";

}

}

**Explanation of the POJO Example:**

1. **Private Fields**: The Employee class has three private fields: id, name, and department. This encapsulation ensures that the data is protected from direct modification.
2. **Constructors**: The class has two constructors: a default no-argument constructor and a parameterized constructor to initialize the fields.
3. **Getter and Setter Methods**: Public getter and setter methods are provided for each field, allowing controlled access and modification.
4. **toString() Method**: The toString() method is overridden to give a readable representation of the Employee object.

**When is it Not a POJO?**

An object is not considered a POJO if it:

1. Implements framework-specific interfaces (e.g., Serializable, Cloneable).
2. Extends classes or uses annotations that are required by a particular framework (e.g., @Entity or @Component annotations in Spring or Hibernate).
3. Contains additional behavior or business logic that is not just related to the data it holds.

**Use Cases for POJOs**

* **Data Transfer Objects (DTOs)**: POJOs are often used as DTOs to carry data between different parts of a program or across a network.
* **Model Objects**: Represent database entities when working with Object-Relational Mapping (ORM) frameworks like Hibernate.
* **Configuration or Setting Objects**: Store application configurations or settings.
* **Serialization/Deserialization**: Hold data to be serialized and deserialized (e.g., converting a POJO to JSON and vice versa).

**POJO vs. Bean vs. Java Object**

* **POJO**: A plain, simple Java object without any special requirements.
* **JavaBean**: A specific type of POJO with additional constraints: a no-argument constructor, properties that are accessible via getters and setters, and being Serializable.
* **Java Object**: Any instance of a class that extends the Object class in Java.

In summary, a POJO is a simple, straightforward Java class that serves as a data container, making it ideal for creating model objects or representing structured data without depending on external libraries or frameworks.

**RUN JAVA PROGRAM(CMD):**

If java -version works without setting the path manually, the path is likely configured automatically by the installer or previously set up in the system.

Suppose we create file Test.java

public class Test1 {

public static void main(String[] args) {

System.out.println("Test1");

}

}

**cmd**

**javac Test.java**

**Test.java:1: error: class Test1 is public, should be declared in a file named Test1.java**

**public class Test1 {**

**^**

**1 error**

[If class is declared public then it should be in file with same name]

Suppose we create file Test1.java

public class Test1 {

public static void main(String[] args) {

System.out.println("Test1");

}

}

class Test2 {

public static void main(String[] args) {

System.out.println("Test2");

}

}

**cmd**

**javac Test1.java**

It will create two class files(Test1.class , Test2.class) in same directory. Then you can execute any one of them

**cmd**

**java Test1 // java Test2**

[here Test1 is public and in same file name]

Suppose we create file Test.java

class Test1 {

public static void main(String[] args) {

System.out.println("Test1");

}

}

class Test2 {

public static void main(String[] args) {

System.out.println("Test2");

}

}

**cmd**

**javac Test.java**

It will create two class files(Test1.class , Test2.class) in same directory. Then you can execute any one of them

**cmd**

**java Test1 // java Test2**

[here Test1 and Test2 are not public and so we are able to create file with diff name Test]