Serialization and Deserialization

Serialization in Java is a mechanism of converting the state of an object into a byte stream. It's primarily used in technologies like Hibernate, RMI, JPA, EJB, and JMS. The byte stream can then be stored in a file or sent over a network. This process is platform-independent, meaning the object serialized on one platform can be deserialized on a different platform.

The reverse operation of serialization is called **deserialization**, where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.

To make a Java object serializable, we implement the `java.io.Serializable` interface. The `ObjectOutputStream` class contains the `writeObject()` method for serializing an object, and the `ObjectInputStream` class contains the `readObject()` method for deserializing an object. Here are the method signatures:

```
public final void writeObject (Object obj) throws IOException
public final Object readObject () throws IOException, ClassNotFoundException
```

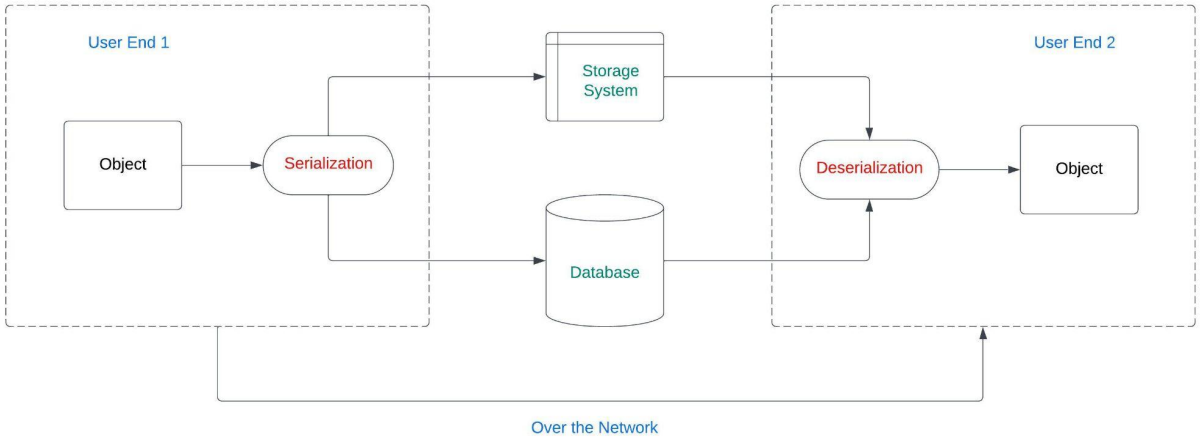
Serialization is mainly used to save/persist the state of an object and to send an object across a network. Only the objects of those classes can be serialized which are implementing `java.io.Serializable` interface. Serializable is a marker interface (has no data member and method). It is used to "mark" Java classes so that objects of these classes may get certain capability.

If a parent class has implemented the Serializable interface, then the child class doesn't need to implement it, but vice-versa is not true. Only non-static data members are saved via the Serialization process. Static data members and transient data members are not saved via the Serialization process. So, if you don't want to save the value of a non-static data member, then make it transient. The constructor of an object is never called when an object is deserialized. Associated objects must be implementing the Serializable interface.

The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object which are compatible with respect to serialization. If the receiver has loaded a class for the object that has a different UID than that of the corresponding sender's class, the Deserialization will result in an InvalidClassException. A Serializable class can declare its own UID explicitly by declaring a field named `serialVersionUID`. It must be static, final, and of type long.

```
public static final long serialVersionUID=42L;
```

If a serializable class doesn't explicitly declare a serialVersionUID, then the serialization runtime will calculate a default one for that class based on various aspects of the class. However, it is strongly recommended that all serializable classes explicitly declare serialVersionUID value, since its computation is highly sensitive to class details that may vary depending on compiler implementations, and any change in the class or using a different id may affect the serialized data..



First, let's define the `Person` class:

```
import java.io.Serializable;

public class Person implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    private int age;

    // Getters and setters
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
}
```

Now, let's create another class that includes the `serialize` method:

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class SerializePerson {
    public void serialize(Object obj, String filename) {
        try {
            FileOutputStream fileOutputStream = new FileOutputStream(filename);
            ObjectOutputStream objectOutputStream = new
ObjectOutputStream(fileOutputStream);
            objectOutputStream.writeObject(obj);
            objectOutputStream.close();
            System.out.println("The object was successfully serialized to " +
filename);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Person person = new Person();
        person.setName("John");
        person.setAge(30);

        SerializePerson sp = new SerializePerson();
        sp.serialize(person, "person.ser");
    }
}
```

this Java code demonstrates the concept of **serialization**. Here's a step-by-step explanation:

1. A `Person` class is defined which implements the `Serializable` interface. This interface is a marker interface (has no methods) used to indicate that a class can be serialized. The `Person` class has two private fields: `name` and `age`, along with their respective getter and setter methods.
2. The `SerializePerson` class is defined with a method `serialize(Object obj, String filename)`. This method takes an object and a filename as parameters. It then writes the object to a file using `ObjectOutputStream`. This process is known as **serialization**. If there's an `IOException`, it will print the stack trace.
3. In the `main` method of the `SerializePerson` class, an instance of `Person` is created and its `name` and `age` fields are set. This `Person` object is then serialized to a file named "person.ser" using the `serialize` method.

So, in essence, this code is creating a `Person` object, setting its fields, and then writing this object to a file through serialization. The serialized object can later be deserialized (read back into an object from the file). This is useful for saving objects between runs of a program, or for sending objects over a network.

Now, let's create another class that includes the `deserialize` method:

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class PersonDeserializer {
    public Object deserialize(String filename) {
        Object obj = null;
        try {
            FileInputStream fileInputStream = new FileInputStream(filename);
            ObjectInputStream objectInputStream = new
ObjectInputStream(fileInputStream);
            obj = objectInputStream.readObject();
            objectInputStream.close();
            System.out.println("The object was successfully deserialized from " +
filename);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
        return obj;
    }

    public static void main(String[] args) {
        PersonDeserializer pd = new PersonDeserializer();
        Person person = (Person) pd.deserialize("person.ser");
        System.out.println("Deserialized Person: " + person.getName() + ", " +
person.getAge());
    }
}
```

this Java code is demonstrating the concept of **deserialization**. Here's a step-by-step explanation:

1. A `PersonDeserializer` class is defined with a method `deserialize(String filename)`. This method takes a filename as a parameter. It then reads an object from a file using `ObjectInputStream`. This process is known as **deserialization**. If there's an `IOException` or `ClassNotFoundException`, it will print the stack trace.

2. In the `main` method of the `PersonDeserializer` class, an instance of `PersonDeserializer` is created. The `deserialize` method is then called with the filename “person.ser”. The returned object is cast to `Person`.
3. The `name` and `age` fields of the deserialized `Person` object are then printed to the console.

So, in essence, this code is reading a `Person` object from a file through deserialization. The deserialized object is then used in the program. This is useful for loading objects saved from a previous run of a program, or for receiving objects over a network. The `Person` class must implement the `Serializable` interface for this to work, as shown in your previous code snippet.