Serialization

- **present in java.io package**

- Mechanism of writing the state of an object into a byte-stream.
- Mainly used to travel object's state on the network (known as marshaling)
- For serializing the object, call the **writeObject()** method of **ObjectOutputStream** *class.*
- The class must implement **Serializable** interface in order to serialize the object. Serializable is a **marker interface.**

Eg:-

```
class Student implements Serializable {
        public int rollNo;
        public String name;

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

public class SerializationExample {
        pvsm(..) throws IOException {
                Student s = new Student(1,"Tushar","DIT");
                FileOutputStream fos = new FOS(“abc.text");
                ObjectOutputStream oos =  new OOS(fos);
                oos.writeObject(s);
        }
}
```

- ***If a class implements serializable then all its sub classes will also be serializable.***

- ***If a class has a reference to another class, all the references must be Serializable otherwise serialization process will not be performed. In such case, NotSerializableException is thrown at runtime.***

- ***If there is any static data member in a class, it will not be serialized because static member is the part of class not object.***

– ***Any data member marked with transient keyword will not be serialized. So when this file is deserialized the value for that particular data member will be set to default(0 for integer, null for String and so on).***

SerialVersionUID

- It is a final and static integer value.
- It is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization.
- SerialVersionUID is used to ensure that during deserialization the same class that was used during serialize process, is loaded.
- If both unique ID matches then only deserialization will be performed. Otherwise we will get Runtime Exception saying **InvalidClassException**.
- JVM generates UID, but it is highly sensitive and depends on the class size and JVM. It is recommended to provide your own UID.

Deserialization

- Reverse of serialisation, i.e. byte-stream is converted into object.
- For deserialization call the **readObject()** method of **ObjectInputStream** class.

Eg:-

```
public class DeserializationExample {
        pvsm(..) throws ClassNotFoundException {
                FileInputStream fis = new FileInputStream(“abc.text");
                ObjectInputStream ois = new ObjectInputStream(fis);
                Student s1 = (Student)ois.readObject();
                System.out.println(s1.toString());
        }
}
```

Serialization with Inheritance

- **If superclass is serializable then subclass is automatically serializable**
  – If superclass is serializable then by default every subclass is serializable. Hence even though subclass doesn’t implement

Serializable interface, we can serialize subclass's object.

- **If a superclass is not serializable then subclass can still be serialized**
  - Even though superclass doesn't implements Serializable interface, we can serialize subclass object if subclass itself implements Serializable interface.

  - At the time of serialization, if any instance variable is inheriting from non-serializable superclass, then JVM ignores original value of that instance variable and save default value to the file.

  - At the time of de-serialization, if any non-serializable superclass is present, then JVM will execute instance control flow in the superclass. To execute instance control flow in a class, JVM will always invoke default(no-arg) constructor of that class. So every non-serializable superclass must necessarily contain default constructor, otherwise we will get runtime-exception, **InvalidClassException.**

- **If the superclass is serializable but we don't want the subclass to be serialized**
  - This can be done by implementing the *writeObject()* and *readObject()* methods in subclass and to throw *NotSerializableException* from these methods.
  - These methods are executed during serialization and de-serialization respectively. By overriding these methods, we are just implementing our own custom serialization.

Eg :-

```
public void writeObject(ObjectOutputStream o) throws IOException {
      throw new NotSerializableException();
}

public void readObject(ObjectInputStream o) throws IOException {
      throw new NotSerializableException();
}
```

Externalizable Interface

- Serves the purpose of custom serialization.
- Unlike Serializable interface which will serialize the variables in object, here we have to explicitly mention what fields or variables you want to serialize.
- Consists of two methods which we have to override to write/read object into/from stream which are

        - ***void readExternal(ObjectInput in)***
        - ***void writeExternal(ObjectOutput out)***
- Serializable does not require no arg constructor. But Externalizable requires public no-arg constructor.

Eg:-

```
public class Student implements Externalizable {
        public int roll;
        public int class;
        public int name;

        public Student() {}

        public void writeExternal(ObjectOutputStream o)
                        throws IOException {
                o.writeObject(name);
        }

        public void readExternal(ObjectInputStream i)
                        throws IOException, ClassNotFoundException{
                name = (String)i.readObject(name);
        }
}
```

- When we write Student object to OutputStream, writeExternal() method is called to persist the data.
- When an Externalizable object is reconstructed, an instance is created first using the public no-argument constructor, then the readExternal method is called. So, it is mandatory to provide a no-argument constructor.