Contents

[Serialization and Deserialization in Java 1](#_Toc163232655)

[Advantages of Java Serialization 1](#_Toc163232656)

[java.io.Serializable interface 4](#_Toc163232657)

[Java Serialization with Inheritance (IS-A Relationship) 5](#_Toc163232658)

[Java Serialization with Aggregation (HAS-A Relationship) 5](#_Toc163232659)

[Java Serialization with array or collection 6](#_Toc163232660)

[serialVersionUID 6](#_Toc163232661)

# Serialization and Deserialization in Java

**Serialization in Java** is a mechanism of *writing the state of an object into a byte-stream*. It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.

The reverse operation of serialization is called *deserialization* where byte-stream is converted into an object. The serialization and deserialization process is platform-independent, it means you can serialize an object in a platform and deserialize in different platform.

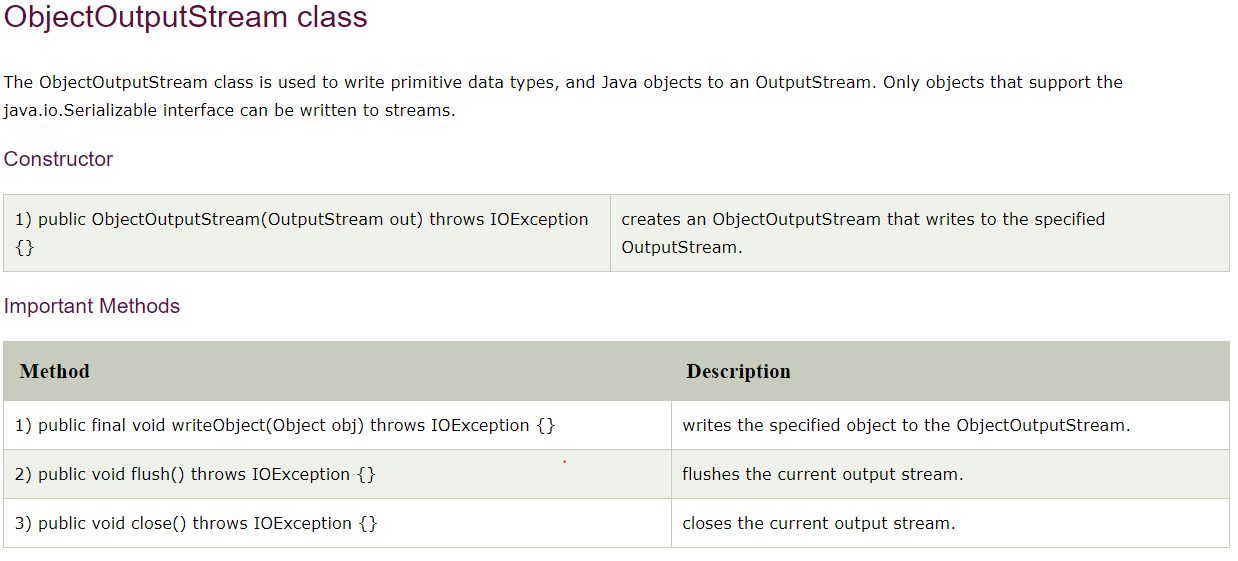
For serializing the object, we call the **writeObject()** method *ObjectOutputStream*, and for deserialization we call the **readObject()** method of *ObjectInputStream* class.

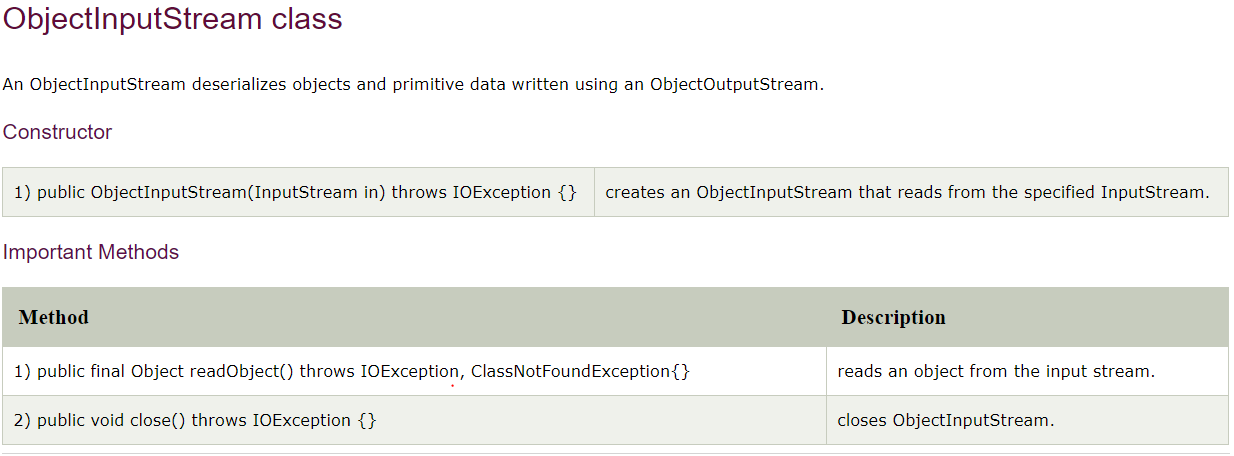
We must have to implement the *Serializable* interface for serializing the object.

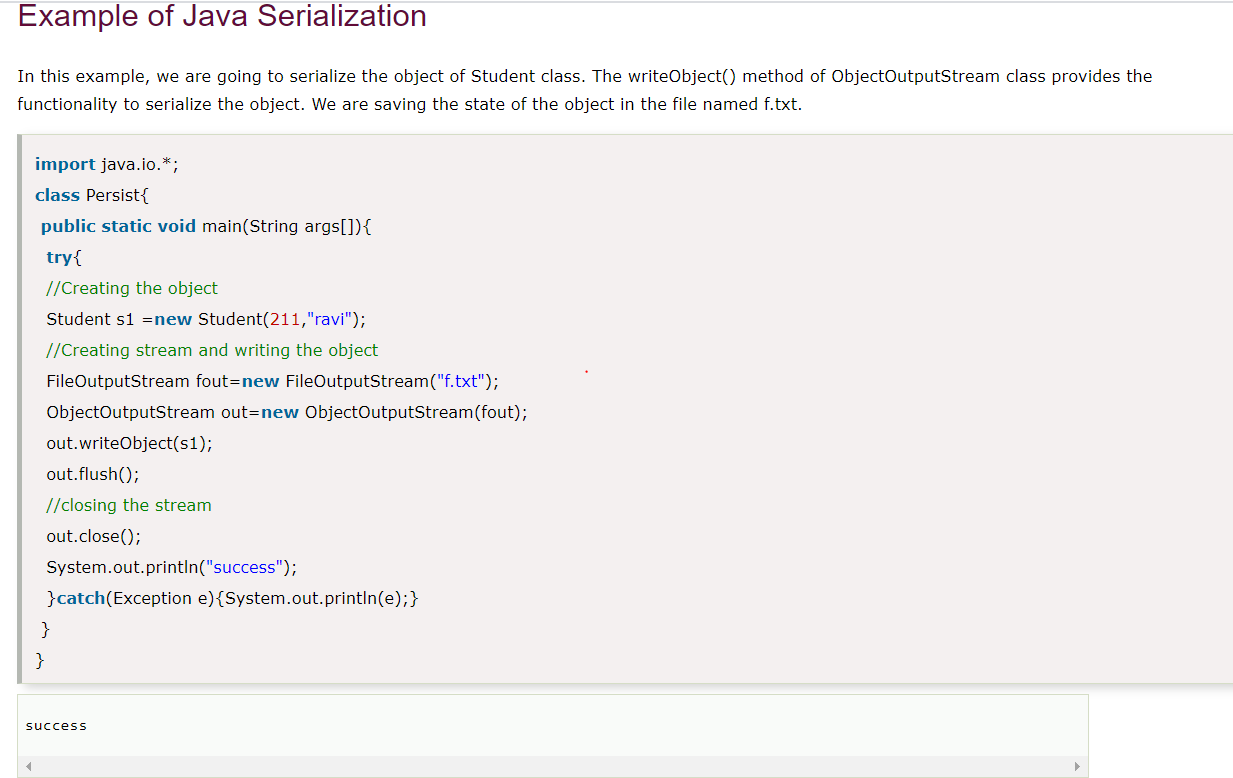
# Advantages of Java Serialization

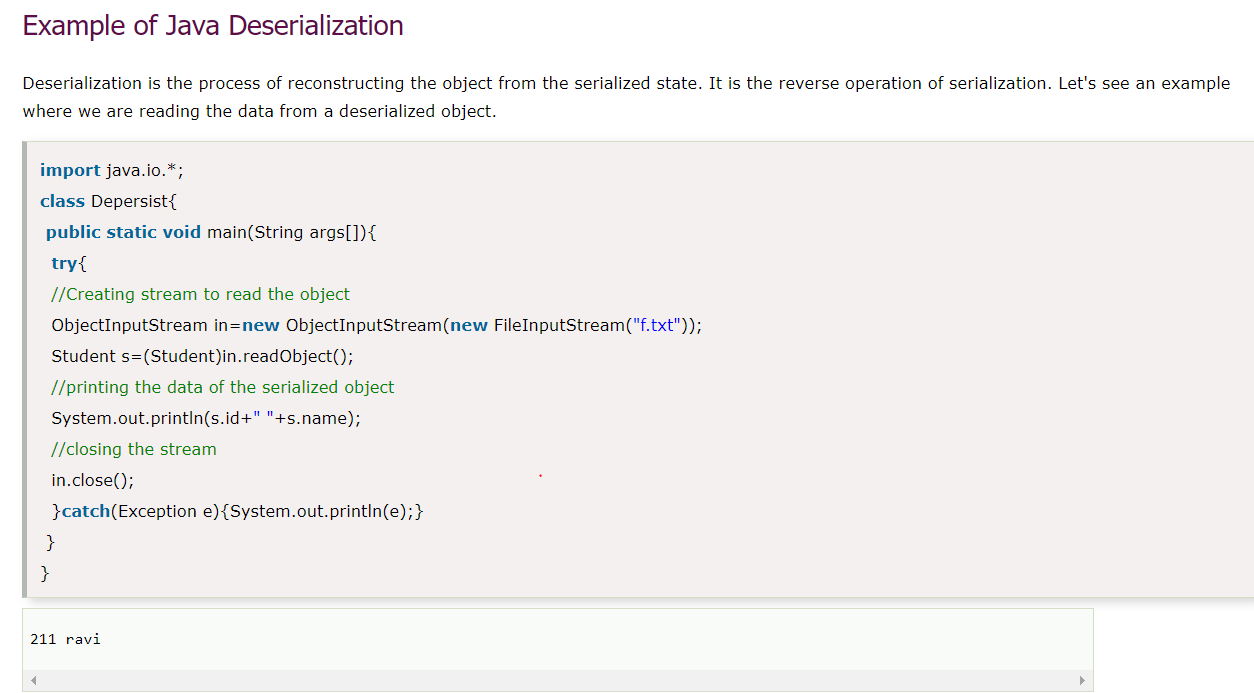
It is mainly used to travel object's state on the network (which is known as marshaling).









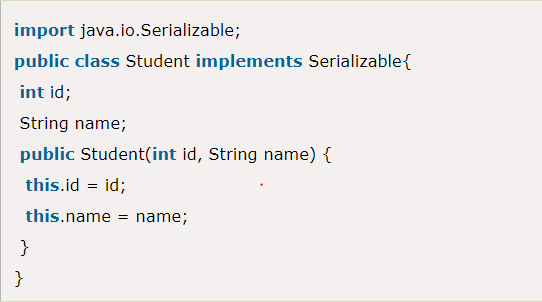


# java.io.Serializable interface

Serializable is a marker interface (has no data member and method). It is used to "mark" Java classes so that the objects of these classes may get a certain capability. The Cloneable and Remote are also marker interfaces.

It must be implemented by the class whose object you want to persist.

The String class and all the wrapper classes implement the *java.io.Serializable* interface by default.



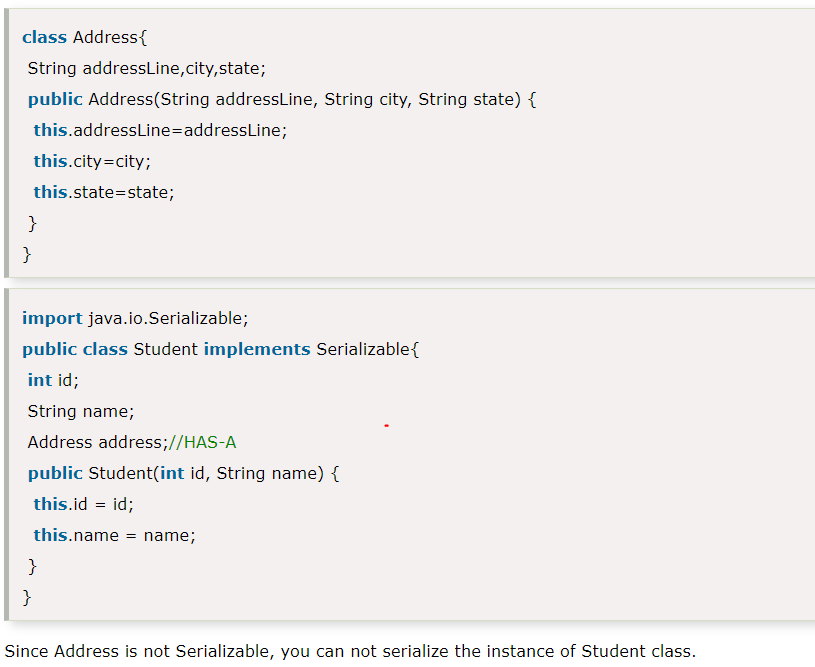
# Java Serialization with Inheritance (IS-A Relationship)

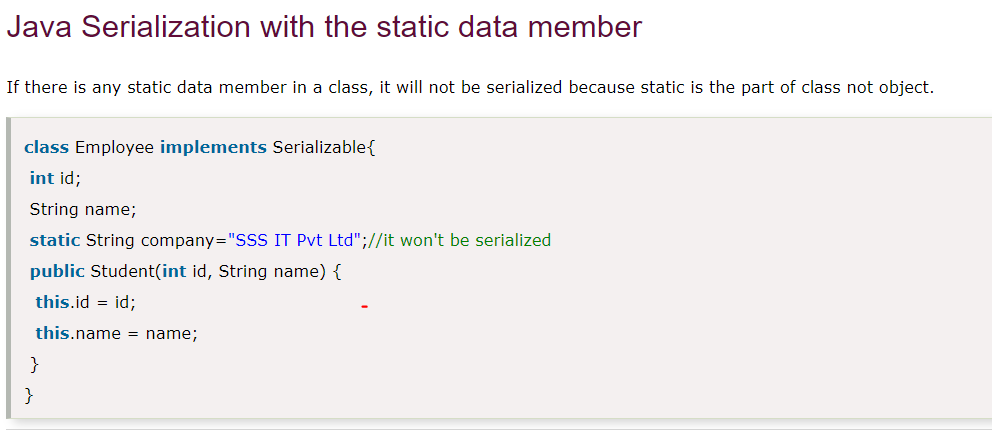
If a class implements serializable then all its sub classes will also be serializable

Now you can serialize the Student class object that extends the Person class which is Serializable. Parent class properties are inherited to subclasses so if parent class is Serializable, subclass would also be.

# Java Serialization with Aggregation (HAS-A Relationship)

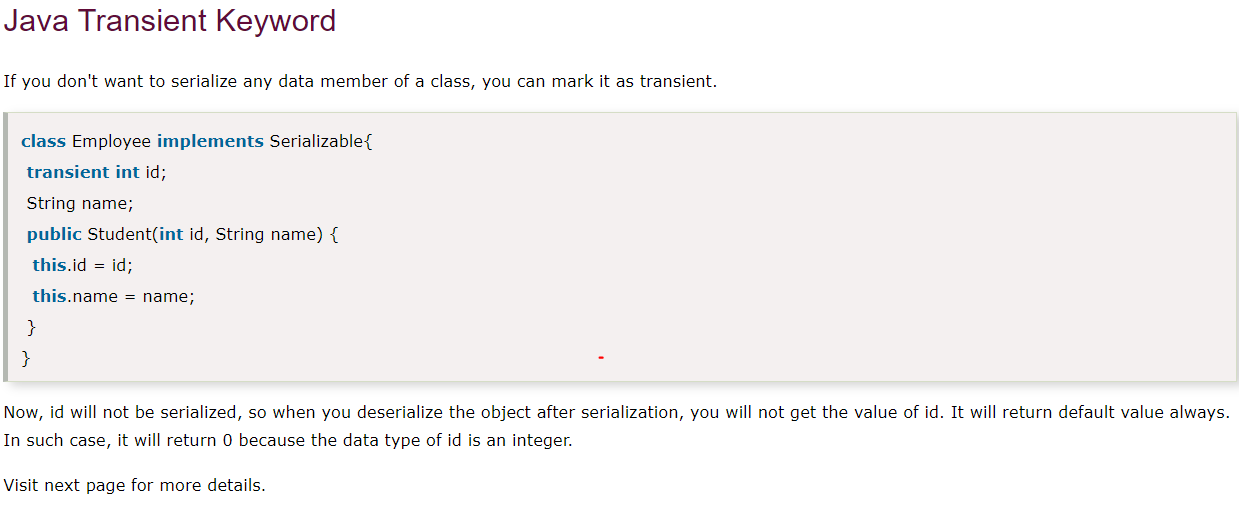
If a class has a reference to another class, all the references must be Serializable otherwise serialization process will not be performed. In such case, NotSerializableException is thrown at runtime.





# Java Serialization with array or collection

Rule: In case of array or collection, all the objects of array or collection must be serializable. If any object is not serialiizable, serialization will be failed.



# serialVersionUID

You're probably familiar with serialization and deserialization. In simple terms, Java objects exist only within the limits of the JVM. When the JVM exits, object values also get destroyed. Serialization means you save the objects as bytes somewhere. Probably in the filesystem. Deserialization means you read these bytes as constructing the Java object again. But how can you make sure the class remains unchanged from serialization to deserialization?

For example, say it is January 1 and we have the Employee class as follows:

public class Employee {

private String id;

private String name;

private int age;

}

With those, you serialized the data to your file system as empdata.dat on the same day.

Now, on January 2, someone changes the class as follows:

public class Employee {

private String id;

private String name;

private Date dateOfBirth;

}

If you are trying to restore (deserialization) empdata.dat to the Employee class, you can see it is not correct. Those are two different formats. Note: Technically, this will deserialize without error, skipping the missing field. But your business logic output may not as expected.

Here is where we need serialVersionUID. It is used during deserialization to verify that the sender ( the person who serializes) and receiver ( the person who deserializes) of a serialized object have loaded classes for that object that are compatible with respect to serialization. In case a receiver has loaded a class for the object that has a different serialVersionUID than that used to serialize, then deserialization will end with InvalidClassException. A serializable class can declare its own serialVersionUID explicitly by declaring a field named “serialVersionUID” that must be static, final, and of type long:.

Here, for ease of understanding, we are using save and read to file system. But serialization can happen in runtime itself through the wire, so the sender and receiver would be in an identical world to save and read.

Even though your IDE insisted you to add seriaVersionUID, you can still compile the program without a compiler error. In such a case, the serialization runtime will calculate a default serialVersionUID value for that class. This calculation will be based on several factors.