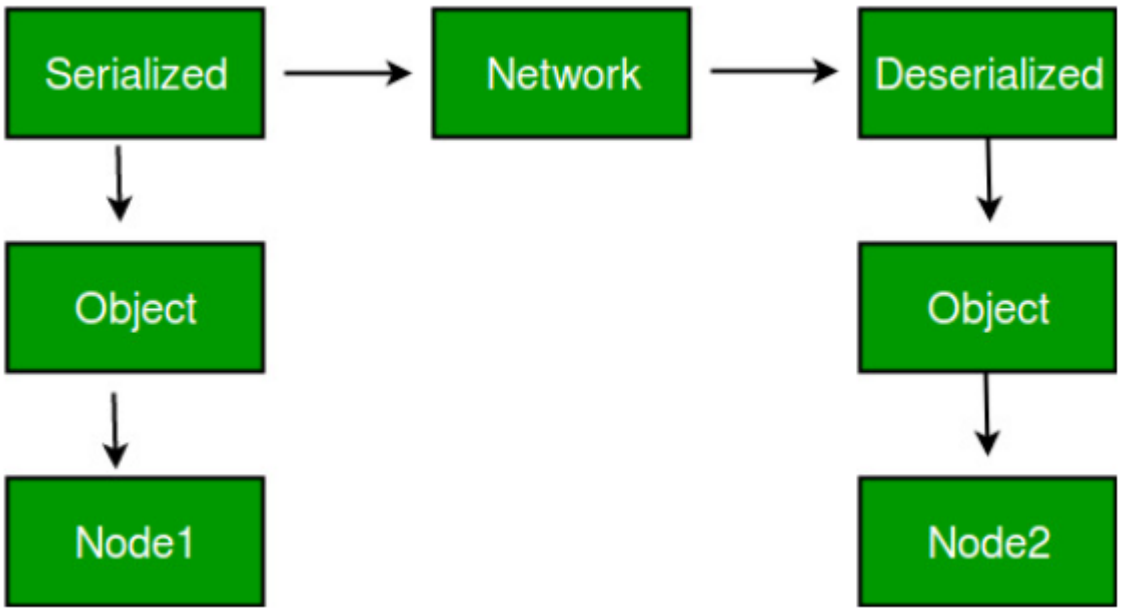
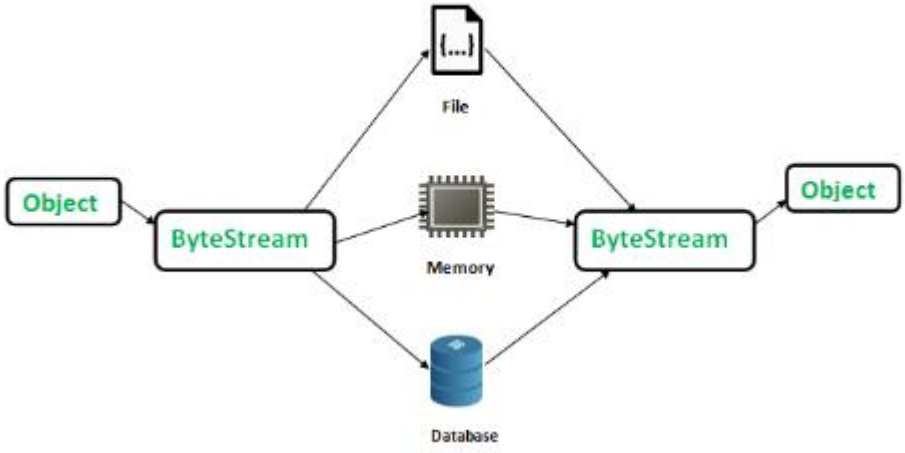
Serialization



Serialization **De-Serialization**



Reference Docs:
https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html
https://docs.oracle.com/javase/8/docs/technotes/guides/serialization/index.html
https://www.baeldung.com/java-serialization

Important points

1. Class must have implemented Serializable interface for serialization process, else NotSerializableException will be thrown.
2. All family members (Object Graph) should have implanted Serializable interface.
3. Serializable is a marker interface.
4. transient keyword is used to avoid serialization, can be used with variables.
5. Callback method can be used by serialization process.

Read: Serializable (Java Platform SE 8) (oracle.com)

```
    private void writeObject(java.io.ObjectOutputStream out) throws IOExcep-
tion;

    private void readObject(java.io.ObjectInputStream in)
        throws IOException, ClassNotFoundException;

    private void readObjectNoData()
        throws ObjectStreamException;
```

6. The *serialVersionUID* attribute is an identifier that is used to serialize/deserialize an object of a Serializable class.

```
import java.io.Serializable;


//if Student does not implements Serializable then NotSerializableException will be thrown at runtime
public class Student implements Serializable {
    int rollNumber;
    String name;

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

    public int getRollNumber() {
        return rollNumber;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "Student{" +
                "rollNumber=" + rollNumber +
                ", name='" + name + '\'' +
                '}';
    }
}


import java.io.*;
```

```
public class Serialization {

    public static void main(String[] args) throws IOException, ClassNotFoundException {

        Student student = new Student (1, "John");
        serializeStudent(student);


        Student student1 = deSerializeStudent();
        System.out.println(student1);

    }

    public static void serializeStudent(Student student) throws IOException {
        FileOutputStream fos = new FileOutputStream("john.serialized");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(student);
        oos.flush();
        oos.close();//Closes the stream. This method must be called to release any resources associated with
the stream.
    }

    public static Student deSerializeStudent() throws IOException, ClassNotFoundException {
        FileInputStream fos = new FileInputStream("john.serialized");
        ObjectInputStream oos = new ObjectInputStream(fos);
        Student john = (Student1) oos.readObject();
        oos.close();//Closes the stream. This method must be called to release any resources associated with
the stream.
        return john;
    }
}
```

```
//all family members should have implemented Serializable for serialization process.

import java.io.Serializable;

class City implements Serializable{
    String name;

    public City(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "City{" +
                "name='" + name + '\'' +
                '}';
    }
}

//Address must be Serializable
class Address implements Serializable {

    String lane1Address;
    String lane2Address;

    City city;


    public Address(String lane1Address, String lane2Address, City city) {
        this.lane1Address = lane1Address;
        this.lane2Address = lane2Address;
        this.city = city;
    }

    @Override
    public String toString() {
        return "Address{" +
                "lane1Address='" + lane1Address + '\'' +
```

```
                ", lane2Address='" + lane2Address + '\'' +
                ", city=" + city +
                '}';
    }
}

public class Student1 implements Serializable {

    int rollNumber;
    String name;

    Address address;

    public Student1(int rollNumber, String name, Address address) {
        this.rollNumber = rollNumber;
        this.name = name;
        this.address = address;
    }

    public int getRollNumber() {
        return rollNumber;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "Student1{" +
                "rollNumber=" + rollNumber +
                ", name='" + name + '\'' +
                ", address=" + address +
                '}';
    }
}



import java.io.*;

public class Serialization {

    public static void main(String[] args) throws IOException, ClassNotFoundException {

        Student1 student = new Student1(1, "John", new Address("line-1", "line-2", new City("Pune")));
        serializeStudent(student);


        Student1 student1 = deSerializeStudent();
        System.out.println(student1);

    }

    public static void serializeStudent(Student1 student) throws IOException {
        FileOutputStream fos = new FileOutputStream("john.serialized");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(student);
        oos.flush();
        oos.close();//Closes the stream. This method must be called to release any resources associated with
the stream.
    }

    public static Student1 deSerializeStudent() throws IOException, ClassNotFoundException {
        FileInputStream fos = new FileInputStream("john.serialized");
        ObjectInputStream oos = new ObjectInputStream(fos);
        Student1 john = (Student1) oos.readObject();
        oos.close();//Closes the stream. This method must be called to release any resources associated with
the stream.
        return john;
```

```
    }
}
```

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;


//from a jar, cannot chnage
class Coller {
    int collerId;

    public Coller(int collerId) {
        this.collerId = collerId;
    }

    public int getCollerId() {
        return collerId;
    }

    @Override
    public String toString() {
        return "Coller{" +
                "collerId='" + collerId + '\'' +
                '}';
    }
}
public class Dog implements Serializable {

    String name;
    transient Coller coller;

    static int dogId = 10;

    public Dog(String name, Coller coller) {
        this.name = name;
        this.coller = coller;
    }

    public String getName() {
        return name;
    }

    public Coller getColler() {
        return coller;
    }


    //callback method,  hook, this will be called by Seriazlier before writing
    private void writeObject(ObjectOutputStream oos) throws IOException {
        oos.defaultWriteObject();
        oos.writeInt(this.coller.collerId);
    }

    //callback method,  hook, this will be called by Seriazlier before reading
    private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {
        ois.defaultReadObject();
        int collerId = ois.readInt();
        this.coller = new Coller(collerId);
    }

    @Override
    public String toString() {
        return "Dog{" +
                "name='" + name + '\'' +
                ", coller=" + coller +
                '}';
```

 Contact: 8087883669

```
    }
}


import java.io.*;

public class DogSerializableTest {

    public static void main(String[] args) throws IOException, ClassNotFoundException {

        Dog hachiko = new Dog("hachiko", new Coller(100));
        serializeDog(hachiko);

        Dog hachiko1 = deSerializeDog();
        System.out.println(hachiko1);
        //System.out.println(hachiko1.dogId);

    }

    public static void serializeDog(Dog dog) throws IOException {
        FileOutputStream fos = new FileOutputStream("dog.serialized");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(dog);
        oos.flush();
        oos.close();//Closes the stream. This method must be called to release any resources associated with
the stream.
    }

    public static Dog deSerializeDog() throws IOException, ClassNotFoundException {
        FileInputStream fos = new FileInputStream("dog.serialized");
        ObjectInputStream oos = new ObjectInputStream(fos);
        Dog dog = (Dog) oos.readObject();
        oos.close();//Closes the stream. This method must be called to release any resources associated with
the stream.
        return dog;
    }
}
```

Transient variables – The values of the transient variables are never considered (they are excluded from the serialization process). i.e. When we declare a variable transient, after de-serialization its value will always be null, false, or, zero (default value).

Static variables – The values of static variables will not be preserved during the de-serialization process. In-fact static variables are also not serialized but since these belongs to the class. After de-serialization they get their current values from the class.

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;


//from a jar, cannot chnage
class Coller {
   int collerId;

   public Coller(int collerId) {
      this.collerId = collerId;
   }

   public int getCollerId() {
      return collerId;
```

```
    }

    @Override
    public String toString() {
        return "Coller{" +
                "collerId='" + collerId + '\'' +
                '}';
    }
}
public class Dog implements Serializable {

    String name;
    transient Coller coller;

    static int dogId;

    public static void setDogId(int id){
        dogId = id;
    }

    public Dog(String name, Coller coller) {
        this.name = name;
        this.coller = coller;
    }

    public String getName() {
        return name;
    }

    public Coller getColler() {
        return coller;
    }


    //callback method,  hook, this will be called by Seriazlier before writing
    private void writeObject(ObjectOutputStream oos) throws IOException {
        oos.defaultWriteObject();
        oos.writeInt(this.coller.collerId);
    }

    //callback method,  hook, this will be called by Seriazlier before reading
    private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {
        ois.defaultReadObject();
        int collerId = ois.readInt();
        this.coller = new Coller(collerId);
    }

    @Override
    public String toString() {
        return "Dog{" +
                "name='" + name + '\'' +
                ", coller=" + coller +
                '}';
    }
}
package com.hdfc.serialization;

import java.io.*;

public class DogSerializableTest {

    public static void main(String[] args) throws IOException, ClassNotFoundException {

        Dog hachiko = new Dog("hachiko", new Coller(100));
        Dog.setDogId(10);
        serializeDog(hachiko);

        Dog.setDogId(100);
```

```
    Dog hachiko1 = deSerializeDog();
    System.out.println(hachiko1);
    System.out.println(hachiko1.dogId);

  }

  public static void serializeDog(Dog dog) throws IOException {
    FileOutputStream fos = new FileOutputStream("dog.serialized");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(dog);
    oos.flush();
    oos.close();//Closes the stream. This method must be called to release any resources associated with the stream.
  }

  public static Dog deSerializeDog() throws IOException, ClassNotFoundException {
    FileInputStream fos = new FileInputStream("dog.serialized");
    ObjectInputStream oos = new ObjectInputStream(fos);
    Dog dog = (Dog) oos.readObject();
    oos.close();//Closes the stream. This method must be called to release any resources associated with the stream.
    return dog;
  }
}
```

Java Serialization is a mechanism provided by the Java programming language to convert Java objects into a byte stream, which can be saved to a file or transmitted over a network, and then reconstructed back into an object when needed. It is mainly used for object persistence, where objects need to be stored and retrieved at a later time.

To make a Java object serializable, it must implement the **java.io.Serializable** interface. This interface acts as a marker, indicating that the object can be serialized. It doesn't have any methods that need to be implemented.

The serialization process is straightforward. To serialize an object, you need to follow these steps:

Create an instance of the java.io.FileOutputStream class to write the byte stream to a file or any other destination.
Create an instance of the java.io.ObjectOutputStream class and pass the FileOutputStream instance to its constructor.
Call the writeObject() method of the ObjectOutputStream class, passing the object you want to serialize as an argument.
Close the streams to release the resources.
Here's an example that demonstrates the serialization process:

```
java
import java.io.*;

public class SerializationExample {
    public static void main(String[] args) {
        // Object to be serialized
        Person person = new Person("John Doe", 25);

        // Serialization
        try {
            FileOutputStream fileOut = new FileOutputStream("person.ser");
            ObjectOutputStream out = new ObjectOutputStream(fileOut);
            out.writeObject(person);
            out.close();
            fileOut.close();
            System.out.println("Object serialized and saved to person.ser");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

class Person implements Serializable {
    private String name;
    private int age;
```

```
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getters and setters (omitted for brevity)
}
```

In the above example, the Person class implements Serializable, indicating that objects of this class can be serialized. The Person object is serialized and saved to a file called "person.ser".

Deserialization is the process of reconstructing an object from the serialized byte stream. Here's an example:

```
java
import java.io.*;

public class DeserializationExample {
    public static void main(String[] args) {
        // Deserialization
        try {
            FileInputStream fileIn = new FileInputStream("person.ser");
            ObjectInputStream in = new ObjectInputStream(fileIn);
            Person person = (Person) in.readObject();
            in.close();
            fileIn.close();
            System.out.println("Object deserialized: " + person.getName() + ", " + person.getAge());
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

In the above example, the Person object is deserialized from the file "person.ser", and its state is printed out.

It's worth noting that Java Serialization has certain security concerns and may not be suitable for all use cases. It's important to consider the implications and alternatives, such as using JSON or other serialization libraries, depending on your specific requirements.

If a superclass is Serializable, then according to normal Java interface rules, all subclasses of that class automatically implement Serializable implicitly. In other words, a subclass of a class marked Serializable passes the IS-A test for Serializable, and thus can be saved without having to explicitly mark the subclass as Serializable. You simply cannot tell whether a class is or is not Serializable UNLESS you can see the class inheritance tree to see if any other superclasses implement Serializable. If the class does not explicitly extend any other class, and does not implement Serializable, then you know for CERTAIN that the class is not Serializable, because class Object does NOT implement Serializable.

class Foo implements Serializable { int num = 3; void changeNum() { num = 10; } }

class Bar implements Serializable { transient int x = 42; }

```
import java.io.*;
class SuperNotSerial {
 public static void main(String [] args) {
 Dog d = new Dog(35, "Fido");
 System.out.println("before: " + d.name + " "
 + d.weight);
 try {
 FileOutputStream fs = new FileOutputStream("testSer.ser");
```

```
 ObjectOutputStream os = new ObjectOutputStream(fs);
 os.writeObject(d);
 os.close();
 } catch (Exception e) { e.printStackTrace(); }
 try {
 FileInputStream fis = new FileInputStream("testSer.ser");
 ObjectInputStream ois = new ObjectInputStream(fis);
 d = (Dog) ois.readObject();
 ois.close();
 } catch (Exception e) { e.printStackTrace(); }
 System.out.println("after: " + d.name + " "
 + d.weight);
 }
}
class Dog extends Animal implements Serializable {
 String name;
 Dog(int w, String n) {
 weight = w; // inherited
 name = n; // not inherited
 }
}
class Animal { // not serializable !
 int weight = 42;
}
which produces the output:
before: Fido 35
after: Fido 42
```

```
package com.hdfc.serialization;

import java.io.*;

public class Animal {
    int animalId = 10;

    @Override
    public String toString() {
        return "Animal{" +
                "animalId=" + animalId +
                '}';
    }
}


//Cow is-a Animal
//is-a
//has-a
class Cow extends Animal implements Serializable {
    private String name;

    public Cow(String name) {
        super();
        this.name = name;
    }


    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "Cow{" +
                "name='" + name + '\'' +
                ", animalId=" + animalId +
                '}';
    }
}
```

```
class TestCow {
    public static void main(String[] args) throws IOException, ClassNotFoundException {


        //Runtime is singleton class
        //Runtime object1 = Runtime.getRuntime();
        //Runtime object2 = Runtime.getRuntime();
        //Runtime object3 = Runtime.getRuntime();


        /**
         * steps: first serialize the code, comment deserialze code
         * comment code Cow cowFromFile = deSerializeCow(); and System.out.println("after Serialization: " +
cowFromFile);
         *
         * step 2: comment serialzie code and uncomment deserialze code and set animalId=0 and check the re-
sult
         * comment:  Cow cow = new Cow("marry");,System.out.println("before Serialization: " + cow); and seri-
alizeCow(cow);
         *
         */

        //before Serilization: Cow{name='marry', animalId=10}
        //Cow cow = new Cow("marry");
        // System.out.println("before Serialization: " + cow);
        // serializeCow(cow);


        //after Serilization: Cow{name='marry', animalId=0}
        Cow cowFromFile = deSerializeCow();
        System.out.println("after Serialization: " + cowFromFile);

    }

    public static void serializeCow(Cow cow) throws IOException {
        FileOutputStream fos = new FileOutputStream("cow.serialized");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(cow);
        oos.flush();
        oos.close();//Closes the stream. This method must be called to release any resources associated with
the stream.
    }

    public static Cow deSerializeCow() throws IOException, ClassNotFoundException {
        FileInputStream fos = new FileInputStream("cow.serialized");
        ObjectInputStream oos = new ObjectInputStream(fos);
        Cow cow = (Cow) oos.readObject();
        oos.close();//Closes the stream. This method must be called to release any resources associated with
the stream.
        return cow;
    }
}
```

serialVersionUID

reference : SerialVersionUID in Java - GeeksforGeeks

- The **SerialVersionUID** must be declared as a **private static final long** variable in Java. This number is calculated by the compiler based on the state of the class and

the class attributes. This is the number that will help the JVM to identify the state of an object when it reads the state of the object from a file.

- The **SerialVersionUID** can be used during **deserialization** to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible w.r.t **serialization**. If the deserialization object is different than serialization, then it can throw an **InvalidClassException**.
- If the **serialVersionUID** is not specified then the runtime will calculate a **default serialVersionUID value** for that class based on various aspects of the class.

The serialization at runtime associates with each serializable class a version number 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 that are compatible with respect to serialization. Geek, now you must be wondering why do we use SerialVersionUID?

It is because SerialVersionUID is used to ensure that during deserialization the same class (that was used during serialize process) is loaded. Consider ran the illustration given below to get a fairer understanding of Serialization & Deserialization.

Illustration:

- Suppose a person who is in the UK and another person who is in India, are going to perform serialization and deserialization respectively. In this case, to authenticate that the receiver who is in India is the authenticated person, JVM creates a unique ID which is known as **SerialVersionUID**.
- In most cases, serialization and deserialization both activities are done by a single person with the same system and same location. But in serialization, sender and receiver are not the same people that is the persons may be different, machine or system may be different and location must be different then SerialVersionUID comes into the picture. In serialization, both sender and receiver should have .class file at the time of beginning only i.e. the person who is going to do serialization and the person who is ready for deserialization should contain the same .class file at the beginning time only.

Serialization at the time of serialization, with every object sender side JVM will save a **Unique Identifier**. JVM is responsible to generate that unique ID based on the corresponding .class file which is present in the sender system. **Deserialization** at the time of deserialization, receiver side JVM will compare the unique ID associated with the Object with local class Unique ID i.e. JVM will also

create a Unique ID based on the corresponding .class file which is present in the receiver system. If both unique ID matched then only deserialization will be performed. Otherwise, we will get Runtime Exception saying InvalidClassException. This unique Identifier is nothing but **SerialVersionUID**.

There are also certain problem associations depending on the default SerialVersionUID generated by JVM as listed below:

1. Both sender and receiver should use the same JVM with respect to platform and version also. Otherwise, the receiver is unable to deserialize because of different SerialVersionUID.
2. Both sender and receiver should use the same '.class' file version. After serialization, if there is any change in the '.class' file at the receiver side then the receiver is unable to deserialize.
3. To generate SerialVersionUID internally JVM may use complex algorithms which may create performance problems.

Implementation:

We can solve the above problem by configuring our own SerialVersionUID. We can configure our own SerialVersionUID for which we need 3 classes as follows:

- Random class which contains two variables which are going to Serialize, let it be 'Geeks'
- Class for sender side which is going to Serialize an object
- Class for receiver side which is going to deserialize

Syntax:

```
private static final long SerialVersionUID=10l;
```

```
package com.hdfc.serialization;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;


//from a jar, cannot change
class Coller {
    int collerId;

    public Coller(int collerId) {
        this.collerId = collerId;
    }

    public int getCollerId() {
        return collerId;
    }

    @Override
    public String toString() {
        return "Coller{" +
```

```
                "collerId='" + collerId + '\'' +
                '}';
    }
}

public class Dog implements Serializable {

    public static final long serialVersionUID = 12312424353456465L;

    String dogName;
    String DogColor;

    transient Coller coller; //ignore while serialiation

    static int dogId; // static fields are ignored while serialiation

    public static void setDogId(int id) {
        dogId = id;
    }

    public Dog(Coller coller, String DogColor, String dogName) {
        this.coller = coller;
        this.DogColor = DogColor;
        this.dogName=dogName;
    }

    public String getDogColor() {
        return DogColor;
    }

    public String getDogName() {
        return this.dogName;
    }

    public Coller getColler() {
        return coller;
    }


    //callback method,  hook, this will be called by Seriazlier before writing
    private void writeObject(ObjectOutputStream oos) throws IOException {
        oos.defaultWriteObject();
        oos.writeInt(this.coller.collerId);
    }

    //callback method,  hook, this will be called by Seriazlier before reading
    private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {
        ois.defaultReadObject();
        int collerId = ois.readInt();
        this.coller = new Coller(collerId);
    }

    @Override
    public String toString() {
        return "Dog{" +
                "DogColor='" + DogColor + '\'' +
                "DogName='" + this.dogName + '\'' +
                ", coller=" + coller +
                '}';
    }
}
package com.hdfc.serialization;

import java.io.*;

public class DogSerializableTest {

    public static void main(String[] args) throws IOException, ClassNotFoundException {

       // Dog hachiko = new Dog(new Coller(100), "Black");
        //Dog.setDogId(10);
       // serializeDog(hachiko);
```

```
        Dog hachiko1 = deSerializeDog();
        System.out.println(hachiko1);
        System.out.println("dogId: " + hachiko1.dogId);

    }

    public static void serializeDog(Dog dog) throws IOException {
        FileOutputStream fos = new FileOutputStream("dog.serialized");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(dog);
        oos.flush();
        oos.close();//Closes the stream. This method must be called to release any resources associated with
the stream.
    }

    public static Dog deSerializeDog() throws IOException, ClassNotFoundException {
        FileInputStream fos = new FileInputStream("dog.serialized");
        ObjectInputStream oos = new ObjectInputStream(fos);
        Dog dog = (Dog) oos.readObject();
        oos.close();//Closes the stream. This method must be called to release any resources associated with
the stream.
        return dog;
    }
}
```

Serialization (Objective 3.3)

❑ The classes you need to understand are all in the java.io package; they include: ObjectOutputStream and ObjectInputStream primarily, and FileOutputStream and FileInputStream because you will use them to create the low-level streams that the ObjectXxxStream classes will use.

❑ A class must implement Serializable before its objects can be serialized.

❑ The ObjectOutputStream.writeObject() method serializes objects, and the ObjectInputStream.readObject() method deserializes objects.

❑ If you mark an instance variable transient, it will not be serialized even thought the rest of the object's state will be.

❑ You can supplement a class's automatic serialization process by implementing the writeObject() and readObject() methods. If you do this, embedding calls to defaultWriteObject() and defaultReadObject(), respectively, will handle the part of serialization that happens normally.

❑ If a superclass implements Serializable, then its subclasses do automatically.

❑ If a superclass doesn't implement Serializable, then when a subclass object is deserialized, the superclass constructor will be invoked, along with its superconstructor(s).