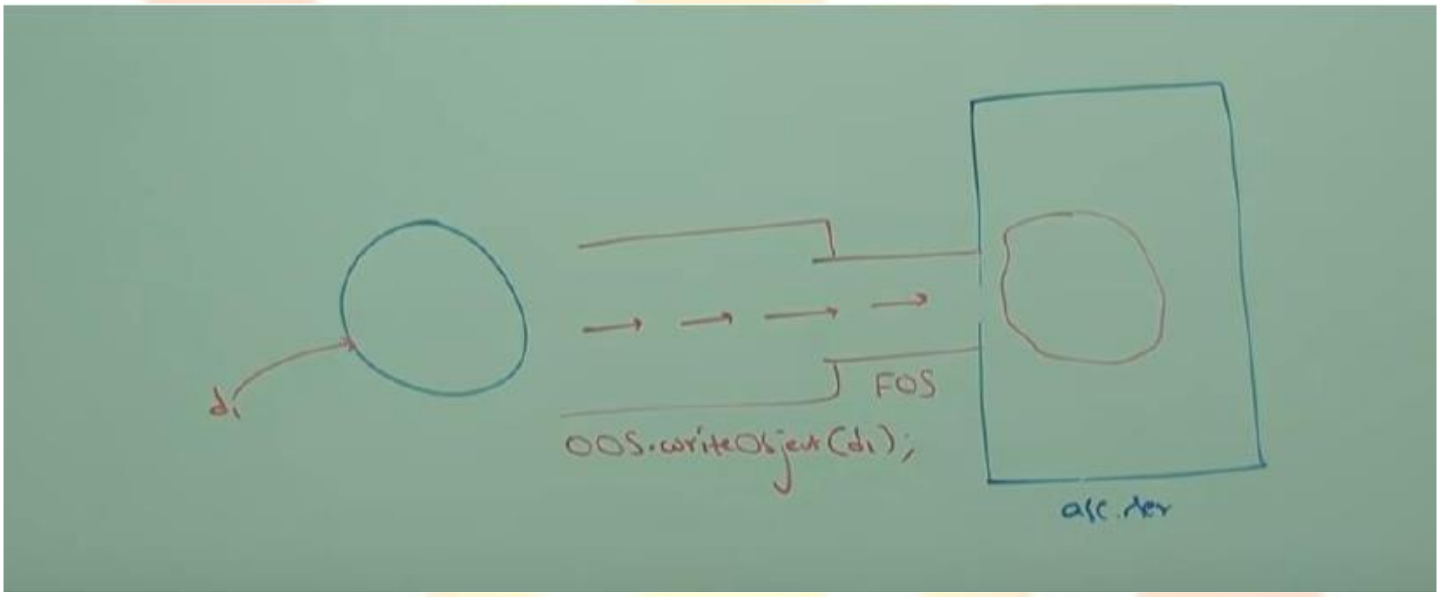
Serialization

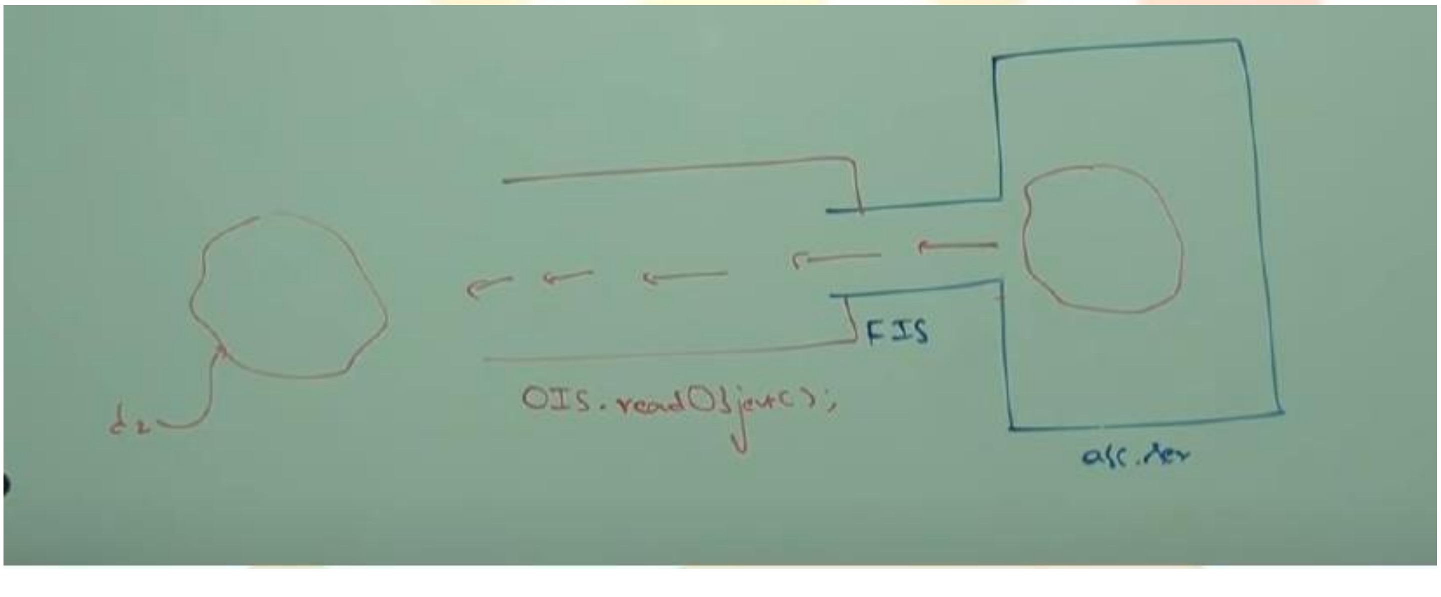
Topics covered in this Chapter:

- Introduction
- transient Keyword
- Object Graphs in Serialization
- Customized Serialization
- Serialization With Respect to Inheritance
- Externalization
- SerialVersionUID

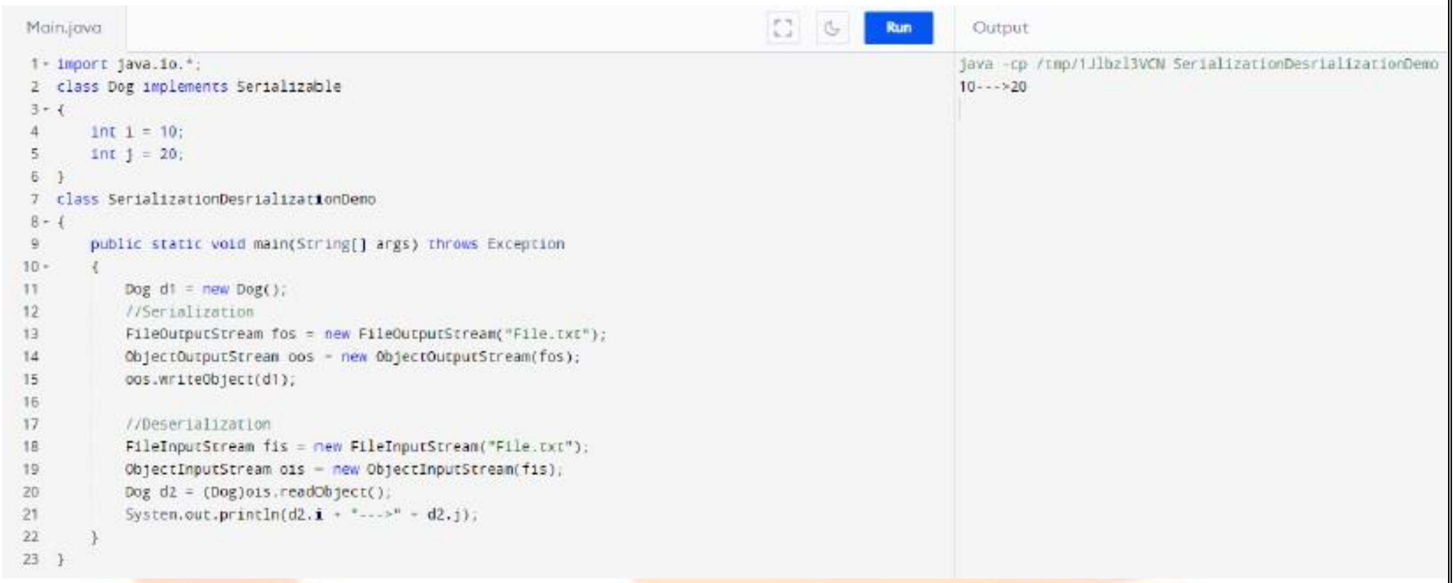
1.. Introduction

- The process of writing state of an object to a file is called **Serialization**. Serialization is the process of converting the regular Java Object into either file supported form or Network Supported form for transporting it.
- The process of reading state of an object from a file is called **Deserialization**. Deserialization is the process of converting either the file supported or network supported form of Object into a Regualar Object.
- Example – Suppose I want to transport Employee Data from System A to System B over the Network. Then System A will first Serialize or convert the Employee Object which our Network can support and then Network will transmit the supported data to System B. Now System B can't use the data directly as it is in a Network Supported form, so System B will now Deserialize or convert the data from Network Supported Form to the Regular Java Employee Object.
- By Using **FileOutputStream** and **ObjectOutputStream** class we can achieve and implements **Serialization** while by Using **FileInputStream** and **ObjectInputStream** class we can achieve and implements **Deserialization**.





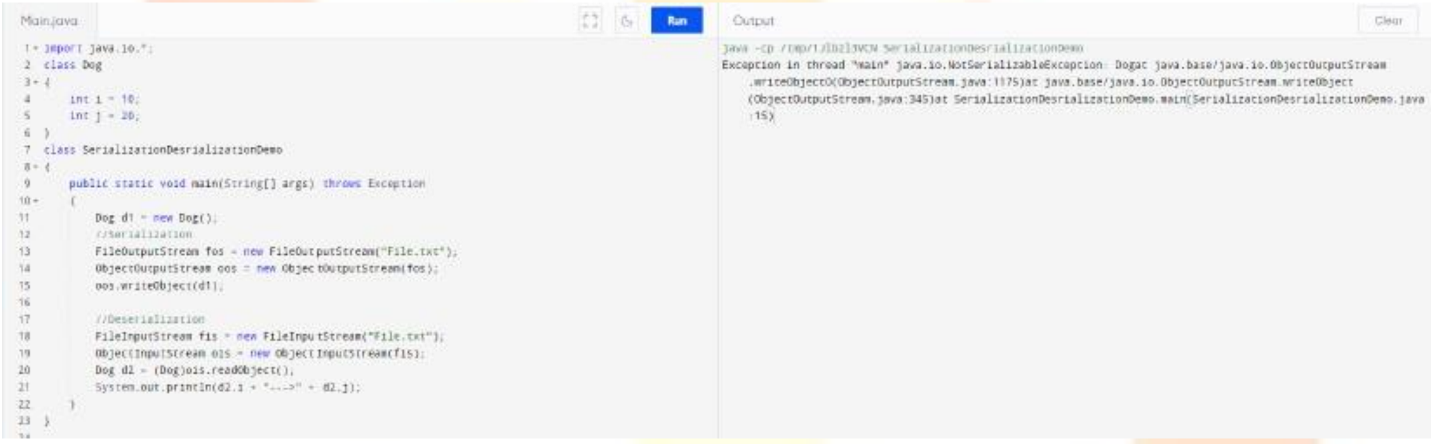
Here abc.ser is a File. File can be anything.



```java
import java.io.*;
class Dog implements Serializable
{
    int i = 10;
    int j = 20;
}
class SerializationDesrializationDemo
{
    public static void main(String[] args) throws Exception
    {
        Dog d1 = new Dog();
        //Serialization
        FileOutputStream fos = new FileOutputStream("File.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d1);

        //Deserialization
        FileInputStream fis = new FileInputStream("File.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Dog d2 = (Dog)ois.readObject();
        System.out.println(d2.i + "--->" + d2.j);
    }
}
```

```
java -cp /tmp/1Jlbzl3VCN SerializationDesrializationDemo
10--->20
```

If class Dog doesn't implement Serializable i.e., if we are trying to serialize a non-serializable Object then we will get Runtime Exception saying **NotSerializableException**.



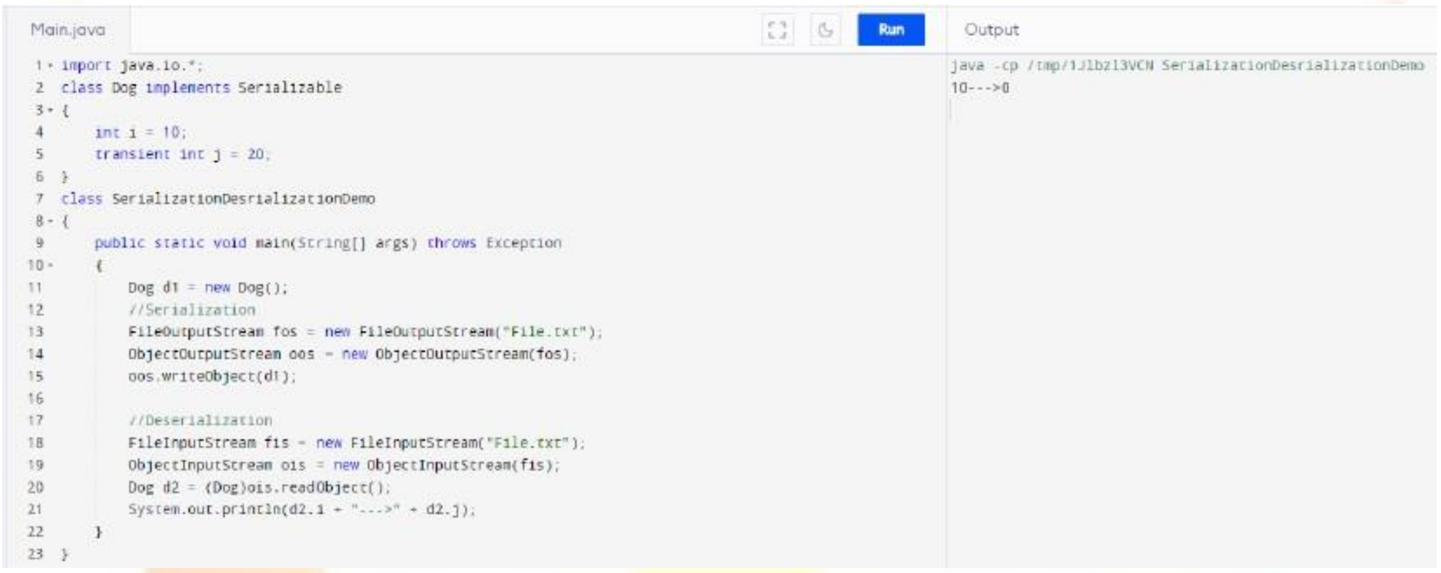
```java
import java.io.*;
class Dog
{
    int i = 10;
    int j = 20;
}
class SerializationDesrializationDemo
{
    public static void main(String[] args) throws Exception
    {
        Dog d1 = new Dog();
        //Serialization
        FileOutputStream fos = new FileOutputStream("File.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d1);

        //Deserialization
        FileInputStream fis = new FileInputStream("File.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Dog d2 = (Dog)ois.readObject();
        System.out.println(d2.i + "--->" + d2.j);
    }
}
```

```
java -cp /tmp/1Jlbzl3VCN SerializationDesrializationDemo
Exception in thread "main" java.io.NotSerializableException: Dogat java.base/java.io.ObjectOutputStream
    .writeObject0(ObjectOutputStream.java:1175)at java.base/java.io.ObjectOutputStream.writeObject
    (ObjectOutputStream.java:345)at SerializationDesrializationDemo.main(SerializationDesrializationDemo.java
    :15)
```

Note: We can serialize and deserialize only serializable Object. To Make an Object Serializable the class of that object should implement Serializable Interface which is present in java.io Package and doesn't contain any methods i.e., it is a Marker Interface.

2.. transient Keyword

- The transient modifier is applicable only for variables but not for methods and classes.
- At the time of Serialization, if we don't want to save the value of a particular variable to meet security constraints then we should declare that variable as transient.
- While performing Serialization, JVM ignores the original value of Transient Variable and saves default value to the file.
- Hence, transient means not to serialize.

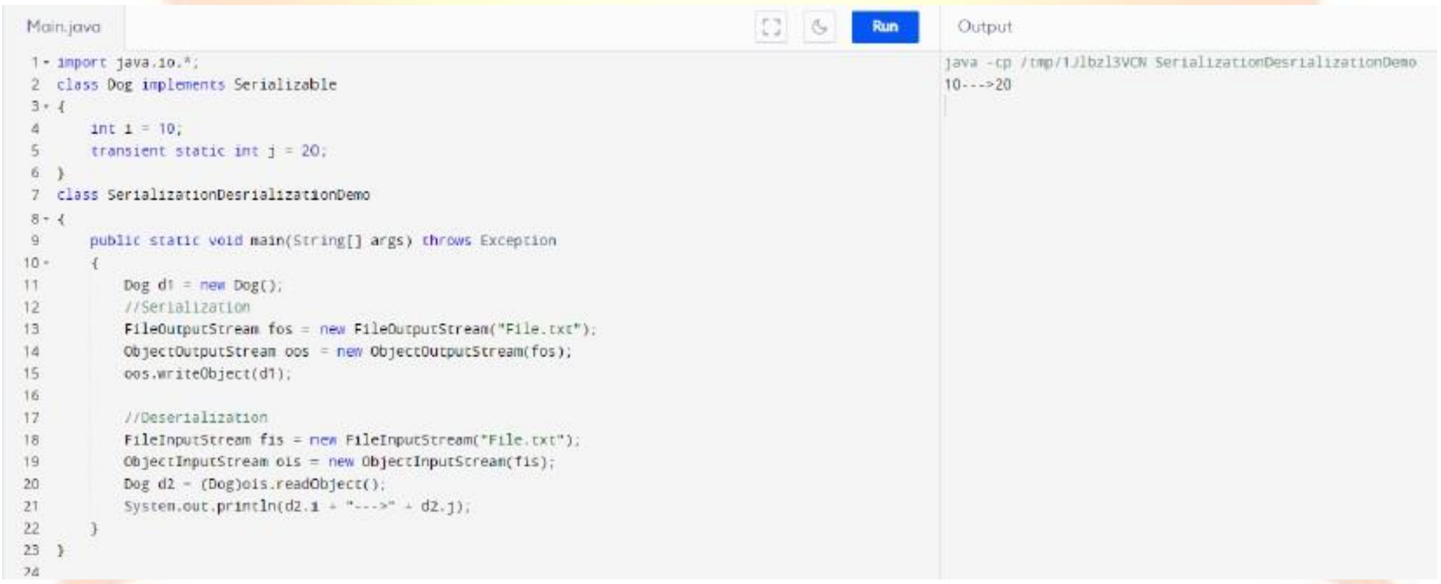


```java
import java.io.*;
class Dog implements Serializable
{
    int i = 10;
    transient int j = 20;
}
class SerializationDesrializationDemo
{
    public static void main(String[] args) throws Exception
    {
        Dog d1 = new Dog();
        //Serialization
        FileOutputStream fos = new FileOutputStream("File.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d1);

        //Deserialization
        FileInputStream fis = new FileInputStream("File.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Dog d2 = (Dog)ois.readObject();
        System.out.println(d2.i + "--->" + d2.j);
    }
}
```

```
java -cp /tmp/1Jlbzl3VCN SerializationDesrializationDemo
10--->0
```

static vs transient

Static variable is not a part of Object State as it represents a class variable. Hence, it won't participate in Serialization. Due to this, declaring static variable as transient, there is no use.

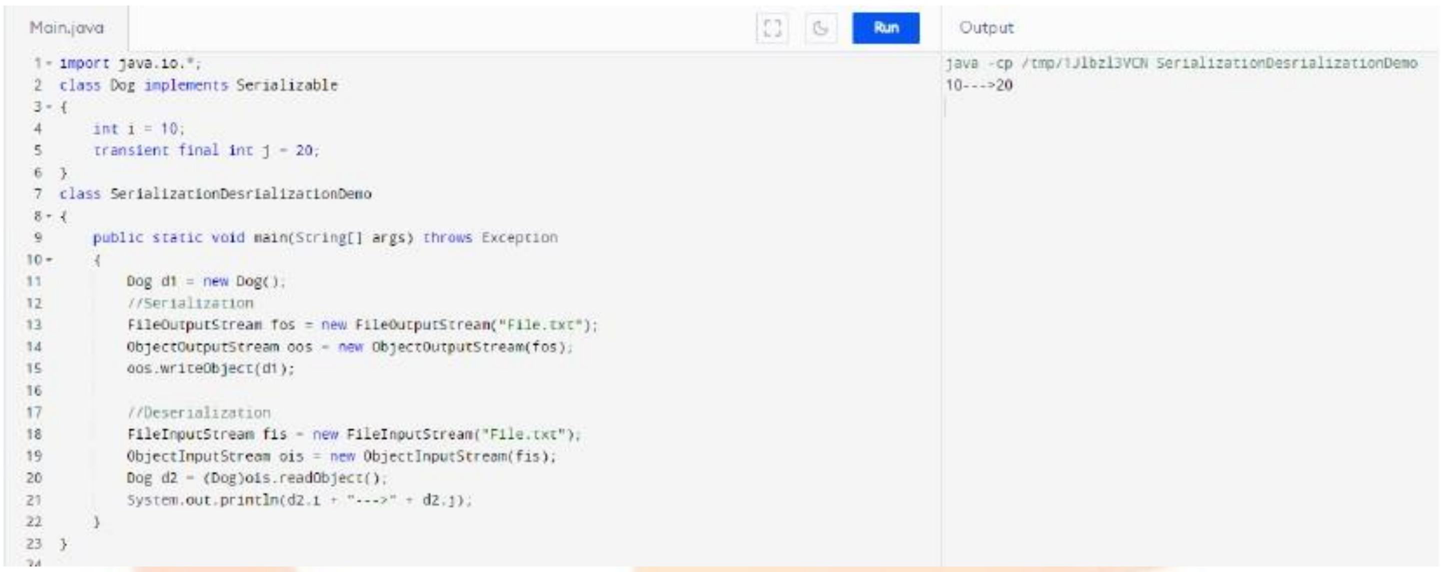


```java
import java.io.*;
class Dog implements Serializable
{
    int i = 10;
    transient static int j = 20;
}
class SerializationDesrializationDemo
{
    public static void main(String[] args) throws Exception
    {
        Dog d1 = new Dog();
        //Serialization
        FileOutputStream fos = new FileOutputStream("File.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d1);

        //Deserialization
        FileInputStream fis = new FileInputStream("File.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Dog d2 = (Dog)ois.readObject();
        System.out.println(d2.i + "--->" + d2.j);
    }
}
```

```
java -cp /tmp/1Jlbzl3VCN SerializationDesrializationDemo
10--->20
```

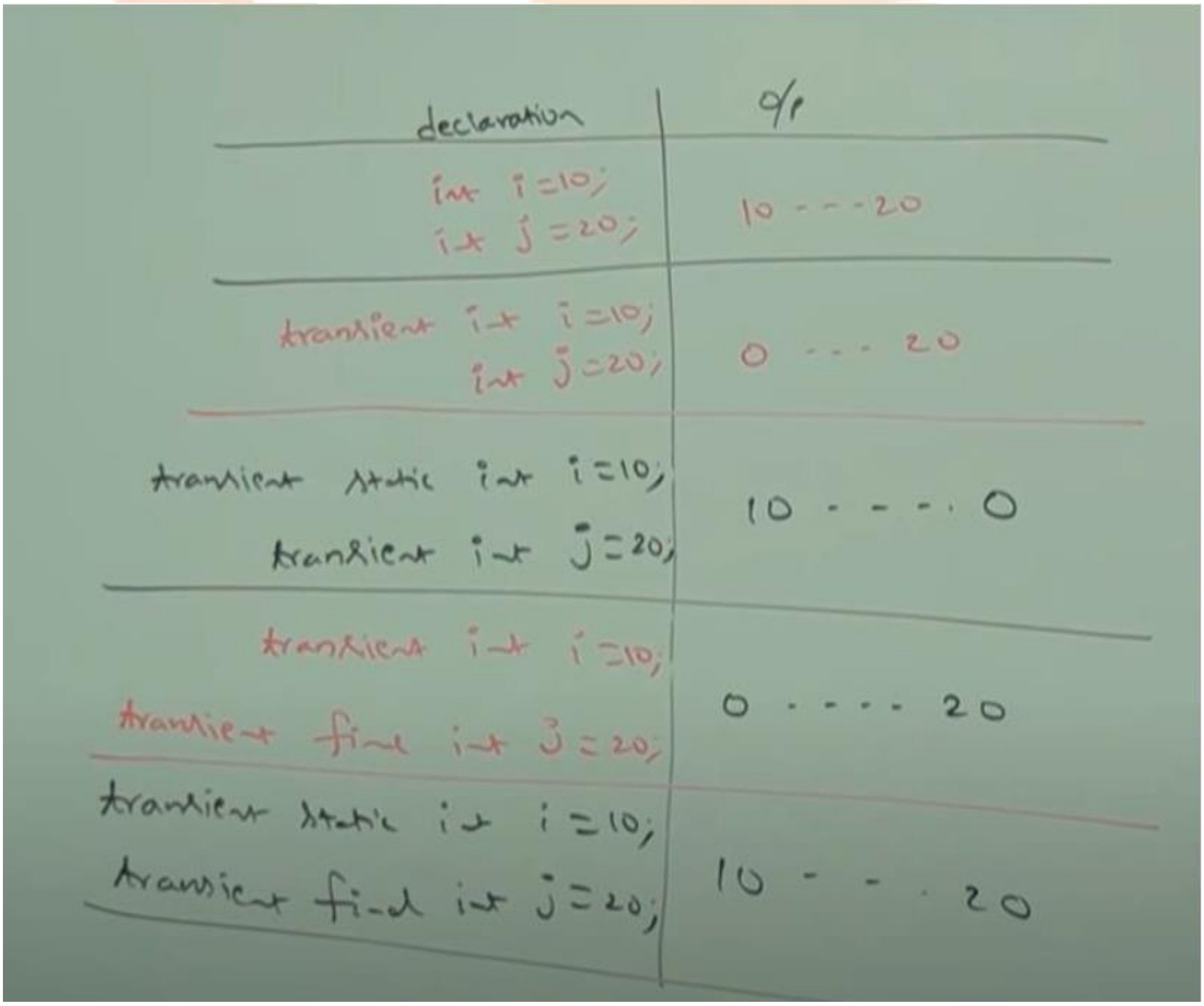
static vs final

Final Variables will be participated in serialization directly by the value. Hence, declaring a final variable as transient, there is no impact or no use.



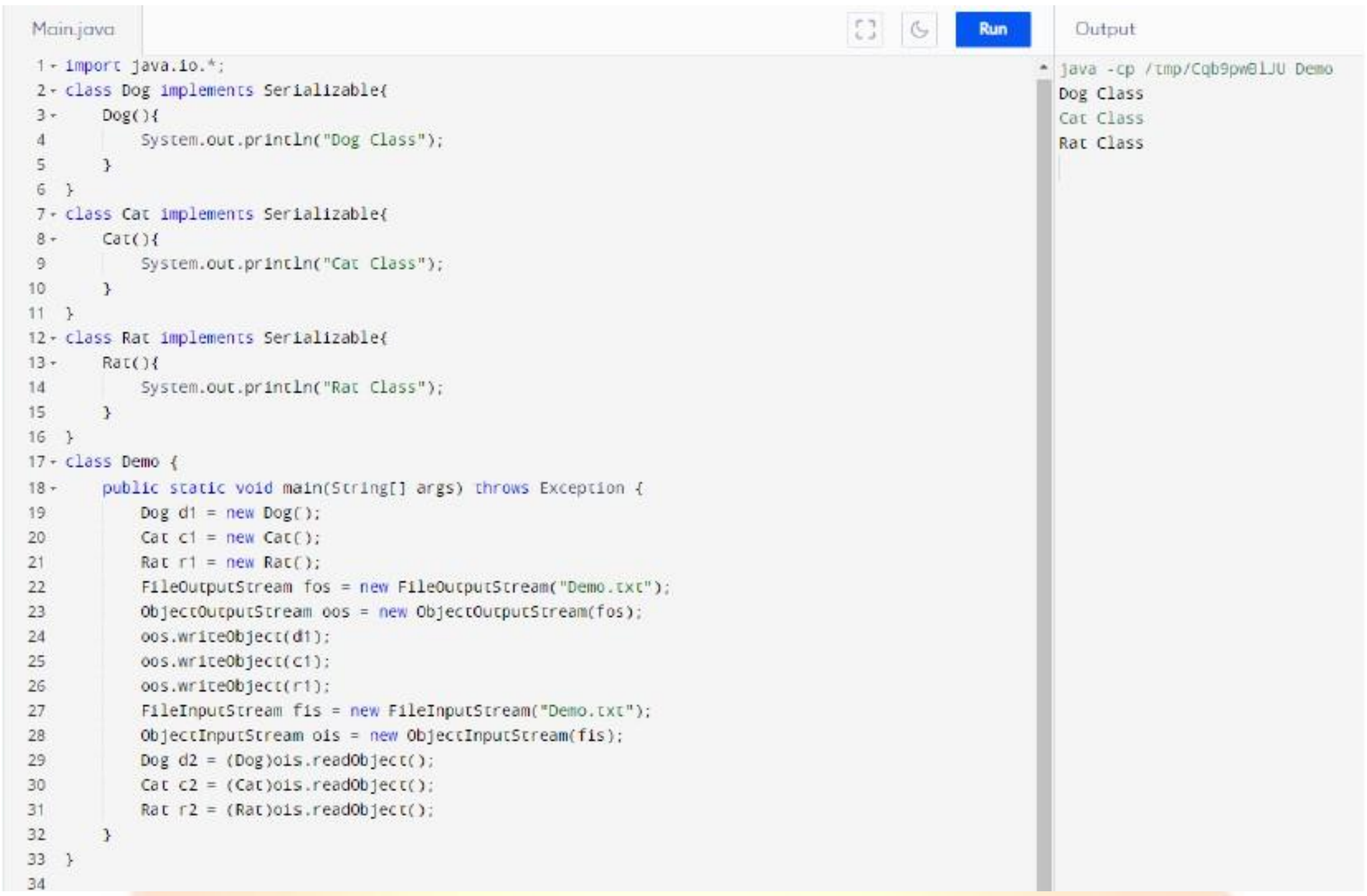
```java
import java.io.*;
class Dog implements Serializable
{
    int i = 10;
    transient final int j = 20;
}
class SerializationDesrializationDemo
{
    public static void main(String[] args) throws Exception
    {
        Dog d1 = new Dog();
        //Serialization
        FileOutputStream fos = new FileOutputStream("File.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d1);

        //Deserialization
        FileInputStream fis = new FileInputStream("File.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Dog d2 = (Dog)ois.readObject();
        System.out.println(d2.i + "--->" + d2.j);
    }
}
```

```
java -cp /tmp/1Jlbzl3VCN SerializationDesrializationDemo
10--->20
```



declaration	o/p
int i=10; int j=20;	10---20
transient int i=10; int j=20;	0 ... 20
transient static int i=10; transient int j=20;	10 ---. 0
transient int i=10; transient final int j=20;	0 20
transient static int i=10; transient final int j=20;	10 - - . 20

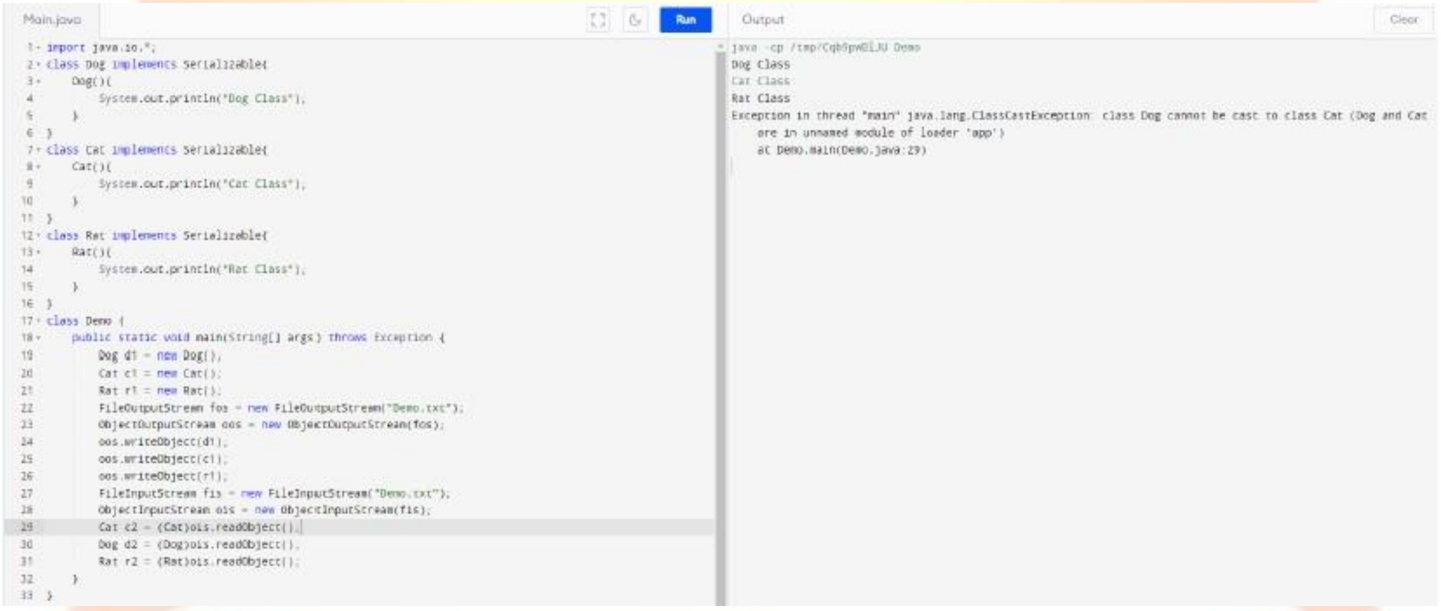
Note: We can serialize any number of Objects to the file, but the order of deserialization should be same as in which we serialize those objects. Hence, Order of Objects is important in Serialization.



```java
import java.io.*;
class Dog implements Serializable{
    Dog(){
        System.out.println("Dog Class");
    }
}
class Cat implements Serializable{
    Cat(){
        System.out.println("Cat Class");
    }
}
class Rat implements Serializable{
    Rat(){
        System.out.println("Rat Class");
    }
}
class Demo {
    public static void main(String[] args) throws Exception {
        Dog d1 = new Dog();
        Cat c1 = new Cat();
        Rat r1 = new Rat();
        FileOutputStream fos = new FileOutputStream("Demo.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d1);
        oos.writeObject(c1);
        oos.writeObject(r1);
        FileInputStream fis = new FileInputStream("Demo.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Dog d2 = (Dog)ois.readObject();
        Cat c2 = (Cat)ois.readObject();
        Rat r2 = (Rat)ois.readObject();
    }
}
```

```
java -cp /tmp/Cqb9pw0lJU Demo
Dog Class
Cat Class
Rat Class
```

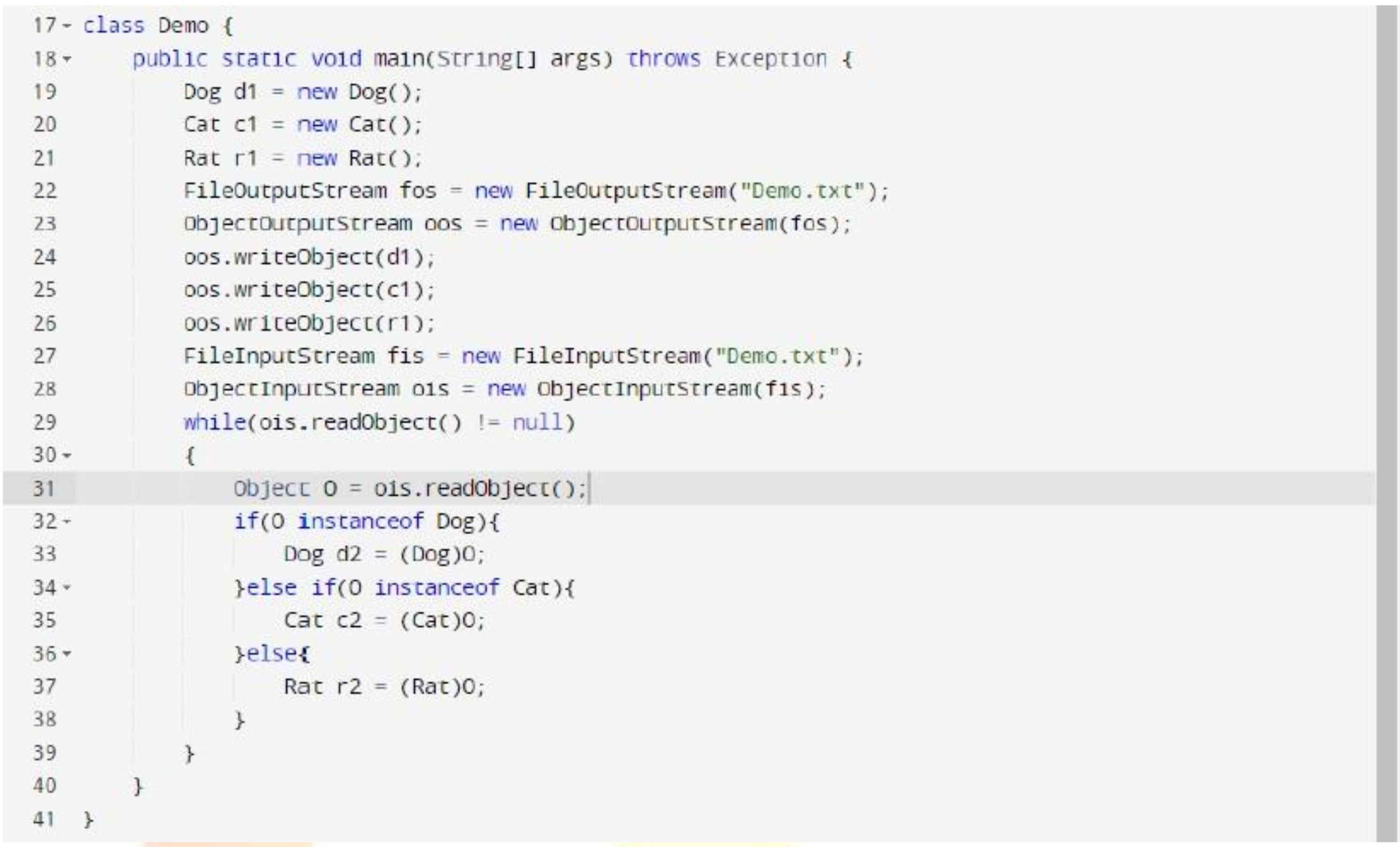
If we change the order then we will get Runtime exception saying ClassCastException.



```java
import java.io.*;
class Dog implements Serializable{
    Dog(){
        System.out.println("Dog Class");
    }
}
class Cat implements Serializable{
    Cat(){
        System.out.println("Cat Class");
    }
}
class Rat implements Serializable{
    Rat(){
        System.out.println("Rat Class");
    }
}
class Demo {
    public static void main(String[] args) throws Exception {
        Dog d1 = new Dog();
        Cat c1 = new Cat();
        Rat r1 = new Rat();
        FileOutputStream fos = new FileOutputStream("Demo.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d1);
        oos.writeObject(c1);
        oos.writeObject(r1);
        FileInputStream fis = new FileInputStream("Demo.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Cat c2 = (Cat)ois.readObject();
        Dog d2 = (Dog)ois.readObject();
        Rat r2 = (Rat)ois.readObject();
    }
}
```

```
java -cp /tmp/Cqb9pw0lJU Demo
Dog Class
Cat Class
Rat Class
Exception in thread "main" java.lang.ClassCastException: class Dog cannot be cast to class Cat (Dog and Cat
    are in unnamed module of loader 'app')
    at Demo.main(Demo.java:29)
```

If we don’t know the order of Objects in serialization then we can use Object Class to hold the Object and can check that object belongs to which class by using instanceof() method and accordingly perform Operation. Consider below code:

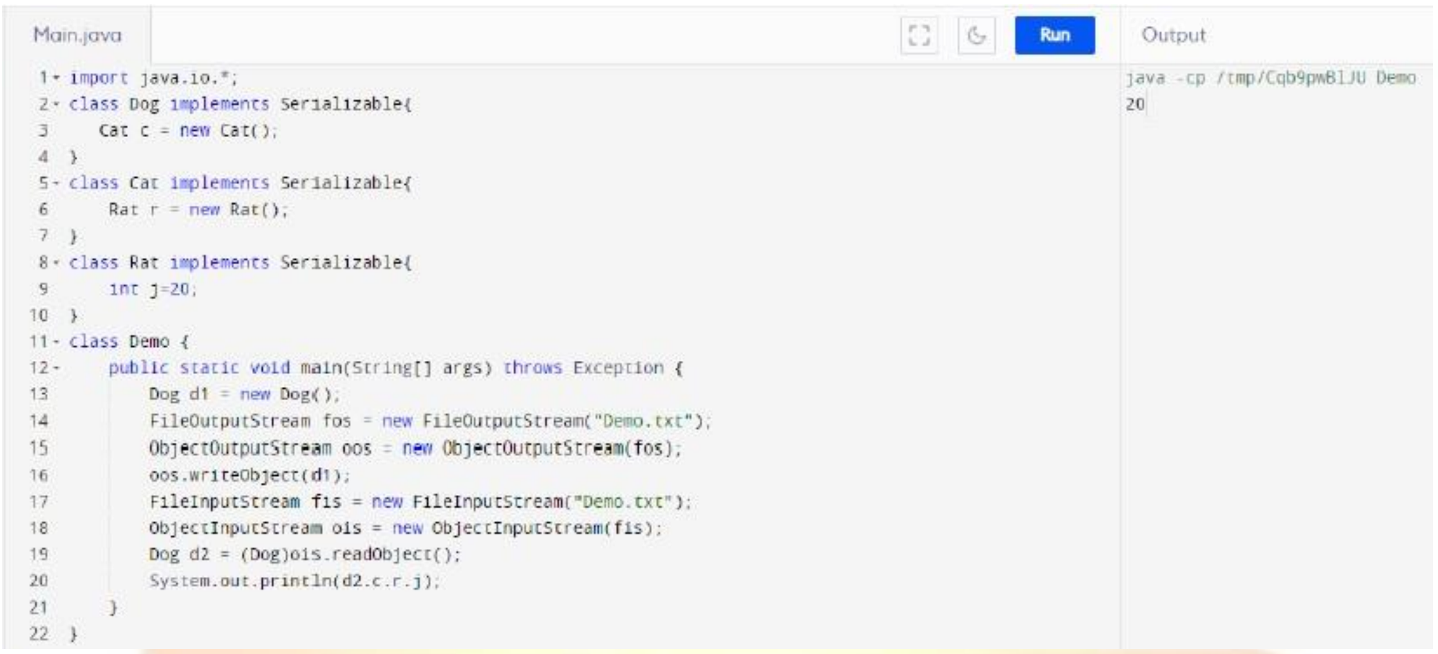


```java
class Demo {
    public static void main(String[] args) throws Exception {
        Dog d1 = new Dog();
        Cat c1 = new Cat();
        Rat r1 = new Rat();
        FileOutputStream fos = new FileOutputStream("Demo.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d1);
        oos.writeObject(c1);
        oos.writeObject(r1);
        FileInputStream fis = new FileInputStream("Demo.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        while(ois.readObject() != null)
        {
            Object O = ois.readObject();
            if(O instanceof Dog){
                Dog d2 = (Dog)O;
            }else if(O instanceof Cat){
                Cat c2 = (Cat)O;
            }else{
                Rat r2 = (Rat)O;
            }
        }
    }
}
```

3.. Object Graphs in Serialization

Whenever we are serializing an Object, set of all objects which are reachable from that Object will be serialized automatically. This group of Objects is nothing but Object Graph.

In Object Graph, every object should be serializable. If any of the object is not serializable then we will get Run time Exception Saying NotSerializableException.



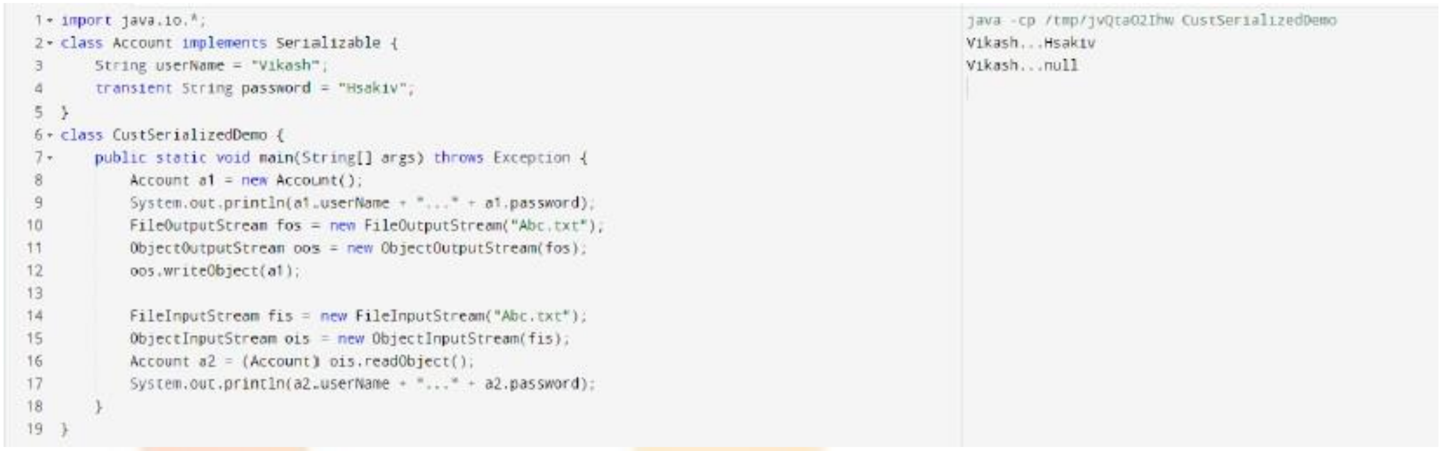
```java
import java.io.*;
class Dog implements Serializable{
    Cat c = new Cat();
}
class Cat implements Serializable{
     Rat r = new Rat();
}
class Rat implements Serializable{
     int j=20;
}
class Demo {
    public static void main(String[] args) throws Exception {
        Dog d1 = new Dog();
        FileOutputStream fos = new FileOutputStream("Demo.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d1);
        FileInputStream fis = new FileInputStream("Demo.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Dog d2 = (Dog)ois.readObject();
        System.out.println(d2.c.r.j);
    }
}
```

```
java -cp /tmp/Cqb9pwBlJU Demo
20
```

Here when we serialize only Dog Object d1, automatically Cat Object c and Rat Object r got serialized as Dog Class contains Cat Object and Cat Class Contains Rat Object and Rat Class contains variable j=10 and forms an Object Graph. All Dog, Cat and Rat class participating in serialization directly or indirectly implements Serializable Interface.

4.. Customized Serialization

During default Serialization, there may be a chance of loss of information because of transient keyword.



```
import java.io.*;
class Account implements Serializable {
    String userName = "Vikash";
    transient String password = "Hsakiv";
}
class CustSerializedDemo {
    public static void main(String[] args) throws Exception {
        Account a1 = new Account();
        System.out.println(a1.userName + "..." + a1.password);
        FileOutputStream fos = new FileOutputStream("Abc.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(a1);

        FileInputStream fis = new FileInputStream("Abc.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Account a2 = (Account) ois.readObject();
        System.out.println(a2.userName + "..." + a2.password);
    }
}
```

```
java -cp /tmp/jvQtaO2Ihw CustSerializedDemo
Vikash...Hsakiv
Vikash...null
```

In above example, before serialization Account Object can provide proper Username and Password. But after deserialization, Account Object can provide only user name but not password. This is due to declaring Password Variable as **transient** which leads to loss of information because of transient keyword. To recover this loss of information we should go for **Customized Serialization**.

We can implement Customized Serialization by using the following two methods:

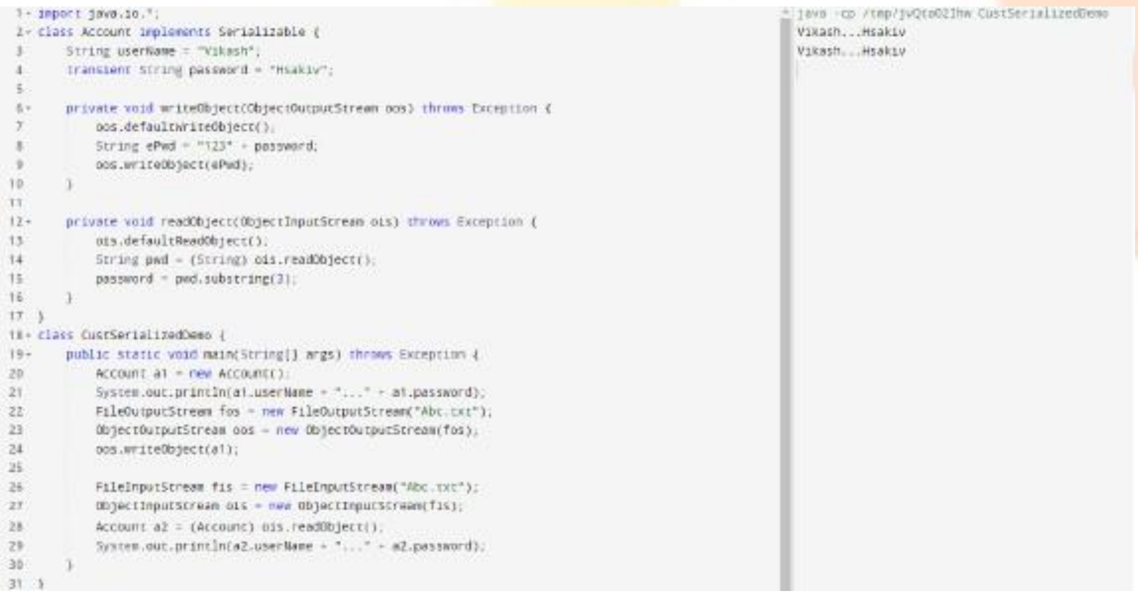
1. private void writeObject(ObjectOutputStream oos) throws Exception

This method will be executed automatically at the time of Serialization. Hence, at the time of Serialization if we want to perform any activities, we have to define that in this method only.

2. private void readObject(ObjectInputStream ois) throws Exception

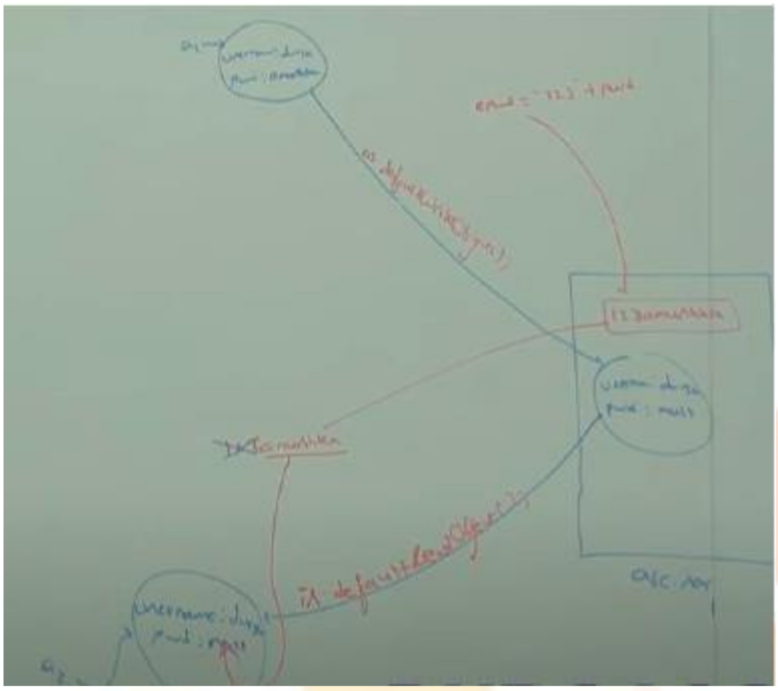
This method will be executed automatically at the time of Deserialization. Hence, at the time of Deserialization if we want to perform any activities, we have to define that in this method only.

The above methods are callback methods because these are executed automatically by the JVM. These methods will be added to the class for the object of which we are doing serialization and wants to add extra activities. In above class, we will add these two methods in Account Class.



```
import java.io.*;
class Account implements Serializable {
    String userName = "Vikash";
    transient String password = "Hsakiv";

    private void writeObject(ObjectOutputStream oos) throws Exception {
        oos.defaultWriteObject();
        String ePwd = "123" + password;
        oos.writeObject(ePwd);
    }

    private void readObject(ObjectInputStream ois) throws Exception {
        ois.defaultReadObject();
        String pwd = (String) ois.readObject();
        password = pwd.substring(3);
    }
}
class CustSerializedDemo {
    public static void main(String[] args) throws Exception {
        Account a1 = new Account();
        System.out.println(a1.userName + "..." + a1.password);
        FileOutputStream fos = new FileOutputStream("Abc.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(a1);

        FileInputStream fis = new FileInputStream("Abc.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Account a2 = (Account) ois.readObject();
        System.out.println(a2.userName + "..." + a2.password);
    }
}
```

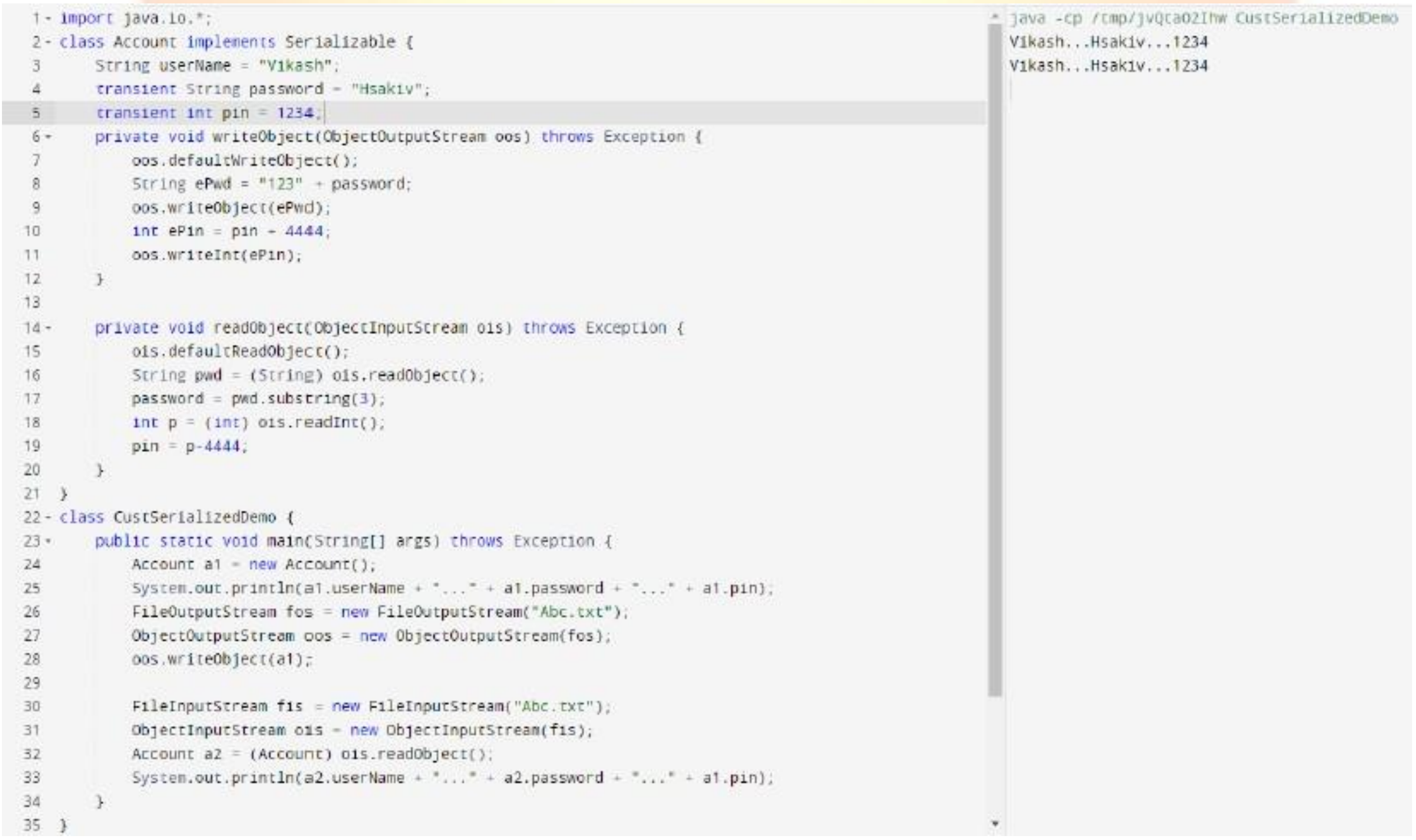
```
java -cp /tmp/jvQtaO2Ihw CustSerializedDemo
Vikash...Hsakiv
Vikash...Hsakiv
```



In above program before serialization and after serialization Account Object can provide valid User Name and Password.

Programmer can't call private methods directly from outside of the class but JVM can call private Methods directly from Outside of the class.

If we have multiple Transient Variable which we have written to the file during Serialization, then during Deserialization we have to follow same order as we have followed while Serialization. See below example:



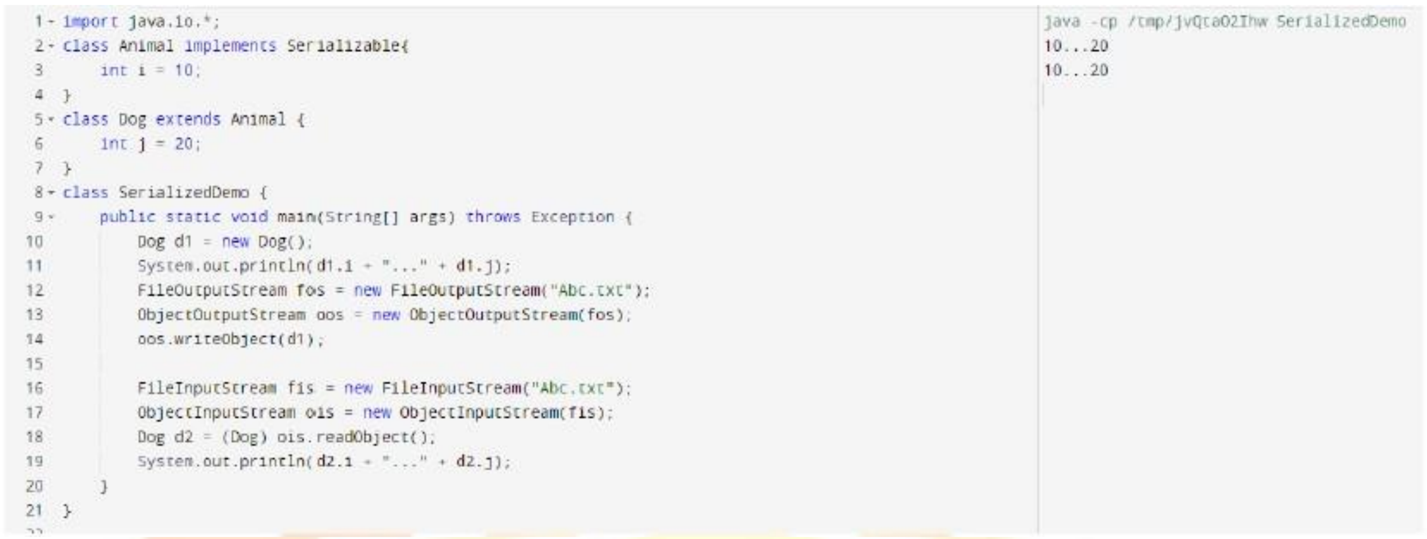
```java
import java.io.*;
class Account implements Serializable {
    String userName = "Vikash";
    transient String password = "Hsakiv";
    transient int pin = 1234;
    private void writeObject(ObjectOutputStream oos) throws Exception {
        oos.defaultWriteObject();
        String ePwd = "123" + password;
        oos.writeObject(ePwd);
        int ePin = pin + 4444;
        oos.writeInt(ePin);
    }

    private void readObject(ObjectInputStream ois) throws Exception {
        ois.defaultReadObject();
        String pwd = (String) ois.readObject();
        password = pwd.substring(3);
        int p = (int) ois.readInt();
        pin = p-4444;
    }
}
class CustSerializedDemo {
    public static void main(String[] args) throws Exception {
        Account a1 = new Account();
        System.out.println(a1.userName + "..." + a1.password + "..." + a1.pin);
        FileOutputStream fos = new FileOutputStream("Abc.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(a1);

        FileInputStream fis = new FileInputStream("Abc.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Account a2 = (Account) ois.readObject();
        System.out.println(a2.userName + "..." + a2.password + "..." + a1.pin);
    }
}
```

```
java -cp /tmp/jvQtaO2Ihw CustSerializedDemo
Vikash...Hsakiv...1234
Vikash...Hsakiv...1234
```

5.. Serialization with respect to Inheritance

Case 1

If Parent Class implements Serializable Interface, then its Serializable nature will be available directly from Parent to Child Class i.e., all child classes will implement Serializable Interface if Parent Class implements Serializable Interface.



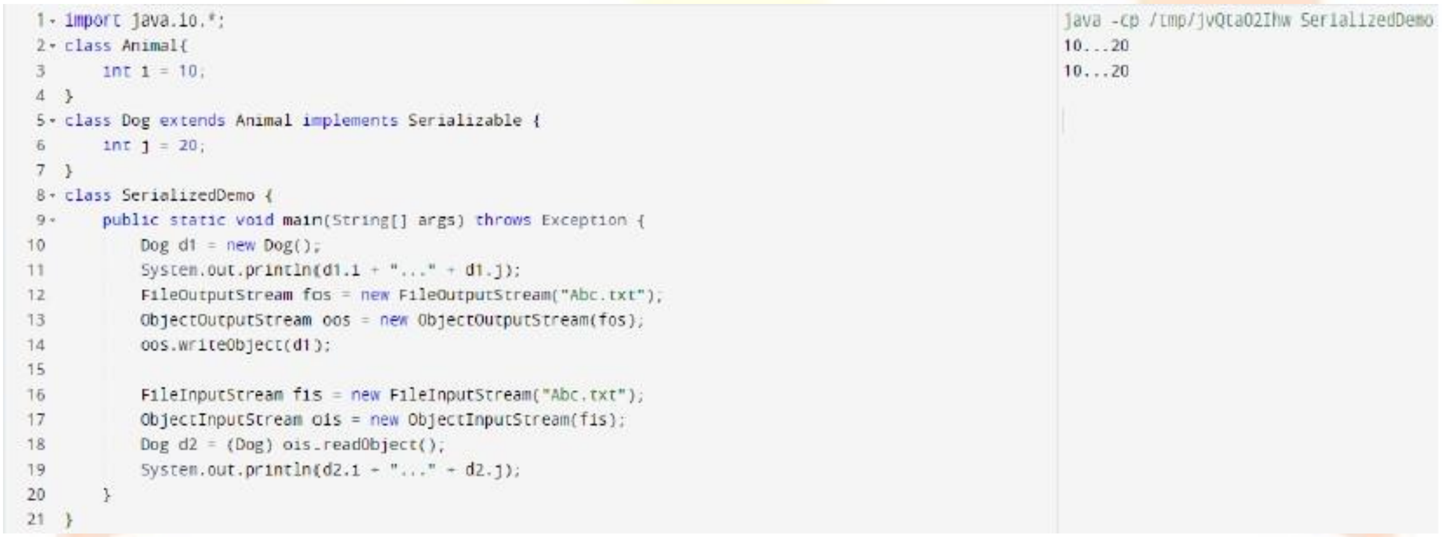
```java
import java.io.*;
class Animal implements Serializable{
    int i = 10;
}
class Dog extends Animal {
    int j = 20;
}
class SerializedDemo {
    public static void main(String[] args) throws Exception {
        Dog d1 = new Dog();
        System.out.println( d1.i + "..." + d1.j);
        FileOutputStream fos = new FileOutputStream("Abc.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d1);

        FileInputStream fis = new FileInputStream("Abc.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Dog d2 = (Dog) ois.readObject();
        System.out.println( d2.i + "..." + d2.j);
    }
}
```

```
java -cp /tmp/jvQtaO2Ihw SerializedDemo
10...20
10...20
```

Note: Object class doesn’t implement Serializable Interface.

Case 2

It is not mandatory that for making Child Class implementing Serializable Interface the parent class should also Implements Serializable Interface. So, When Parent class is not implementing Serializable Interface, Child Class can implement Serializable Interface and this code will perfectly compile fine.

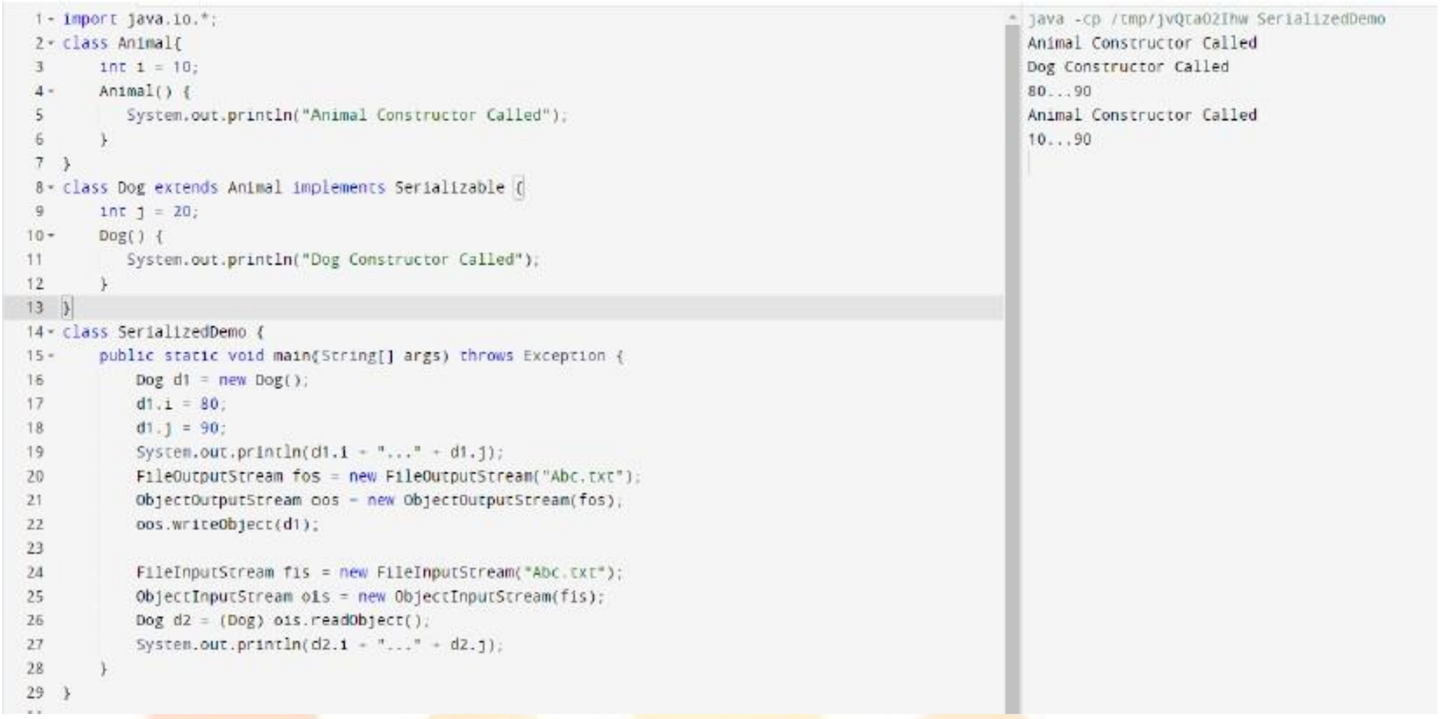


```java
import java.io.*;
class Animal{
    int i = 10;
}
class Dog extends Animal implements Serializable {
    int j = 20;
}
class SerializedDemo {
    public static void main(String[] args) throws Exception {
        Dog d1 = new Dog();
        System.out.println(d1.i + "..." + d1.j);
        FileOutputStream fos = new FileOutputStream("Abc.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d1);

        FileInputStream fis = new FileInputStream("Abc.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Dog d2 = (Dog) ois.readObject();
        System.out.println(d2.i + "..." + d2.j);
    }
}
```

```
java -cp /tmp/jvQtaO2Ihw SerializedDemo
10...20
10...20
```

At the time of serialization, JVM will check if there is any variable inheriting from non-serializable parent or not. If any variable inheritng from non-serializable parent then JVM will save default value to the file by ignoring Original Value.

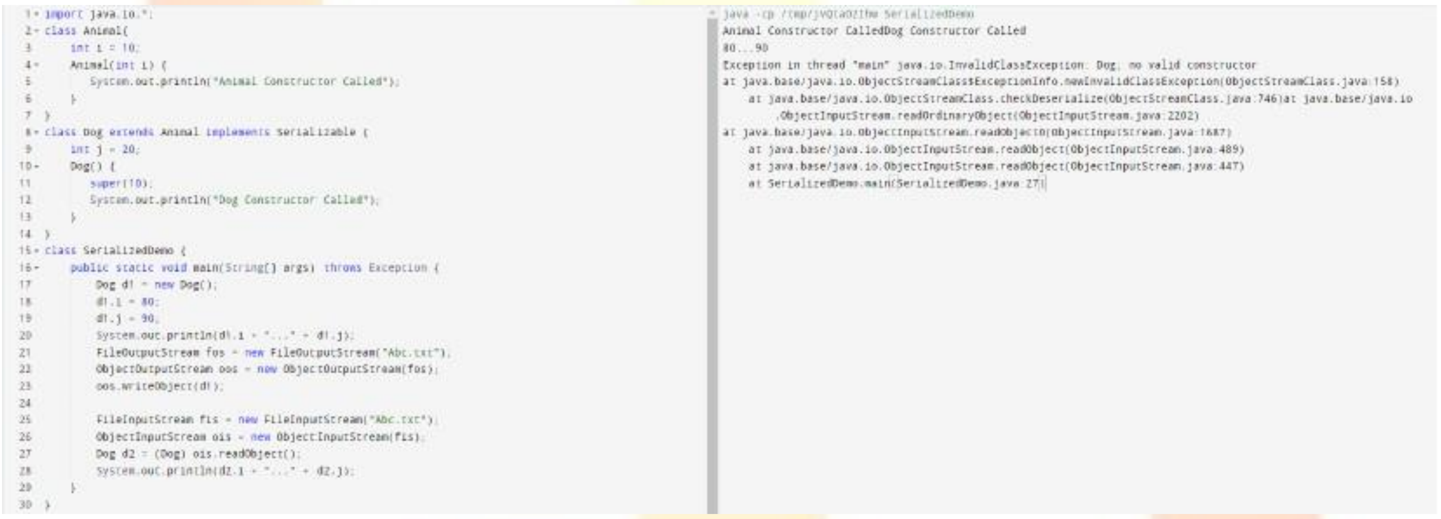
At the time of Deserialization, JVM will check if there is any variable inheriting from non-serializable parent or not. If any variable inheritng from non-serializable parent then JVM will execute Instance Control Flow in every non-serializable parent and share its instance variable to the current object. So if we change the value of i and j from 10, 20 to 80,90 then we will see the below Output.



```java
import java.io.*;
class Animal{
    int i = 10;
    Animal() {
        System.out.println("Animal Constructor Called");
    }
}
class Dog extends Animal implements Serializable {
    int j = 20;
    Dog() {
        System.out.println("Dog Constructor Called");
    }
}
class SerializedDemo {
    public static void main(String[] args) throws Exception {
        Dog d1 = new Dog();
        d1.i = 80;
        d1.j = 90;
        System.out.println(d1.i + "..." + d1.j);
        FileOutputStream fos = new FileOutputStream("Abc.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d1);

        FileInputStream fis = new FileInputStream("Abc.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Dog d2 = (Dog) ois.readObject();
        System.out.println(d2.i + "..." + d2.j);
    }
}
```

```
java -cp /tmp/jvQtaO2Ihw SerializedDemo
Animal Constructor Called
Dog Constructor Called
80...90
Animal Constructor Called
10...90
```

Note: Every non-serializable parent should compulsorily contain no-argument constructors which is default constructor of every class. If class doesn’t contain no-argument constructors then we will get InvalidClassException.



```java
import java.io.*;
class Animal{
    int i = 10;
    Animal(int i) {
        System.out.println("Animal Constructor Called");
    }
}
class Dog extends Animal implements Serializable {
    int j = 20;
    Dog() {
        super(10);
        System.out.println("Dog Constructor Called");
    }
}
class SerializedDemo {
    public static void main(String[] args) throws Exception {
        Dog d1 = new Dog();
        d1.i = 80;
        d1.j = 90;
        System.out.println(d1.i + "..." + d1.j);
        FileOutputStream fos = new FileOutputStream("Abc.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(d1);

        FileInputStream fis = new FileInputStream("Abc.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Dog d2 = (Dog) ois.readObject();
        System.out.println(d2.i + "..." + d2.j);
    }
}
```

```
java -cp /tmp/jvQtaO2Ihw SerializedDemo
Animal Constructor CalledDog Constructor Called
80...90
Exception in thread "main" java.io.InvalidClassException: Dog; no valid constructor
at java.base/java.io.ObjectStreamClass$ExceptionInfo.newInvalidClassException(ObjectStreamClass.java:158)
    at java.base/java.io.ObjectStreamClass.checkDeserialize(ObjectStreamClass.java:746)at java.base/java.io
        .ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:2202)
at java.base/java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1687)
    at java.base/java.io.ObjectInputStream.readObject(ObjectInputStream.java:489)
    at java.base/java.io.ObjectInputStream.readObject(ObjectInputStream.java:447)
    at SerializedDemo.main(SerializedDemo.java:27)
```

6.. Externalization

In Serialization, everything takes care by JVM and programmer doesn’t have any control. In Serialization, it is always possible to save the total object to the file but we can’t save part of the Object and this may create Performance Problem. Example – If an object has 1000 attributes, then if we go for serialization then all 1000 attributes will go for Serialization and we have to deserialize all those 1000 attributes, though we require only 1 or 2 Attributes. This degrades the performance of the System. To Overcome this Problem, we should go for Externalization.

The main advantage of Externalization over Serialization is that here, everything takes care by Programmer and JVM has no control over it. In Externalization we can serialize either complete Object or we can also serialize Part of an Object based on our requirement and this enhances the performance of the System.

For supporting Externalization ability, class should implement Externalizable Interface which is a Child Interface of Serializable Interface. Serializable Interface is a Marker Interface which doesn’t contains any methods while Externalizable interface contains only following two methods:

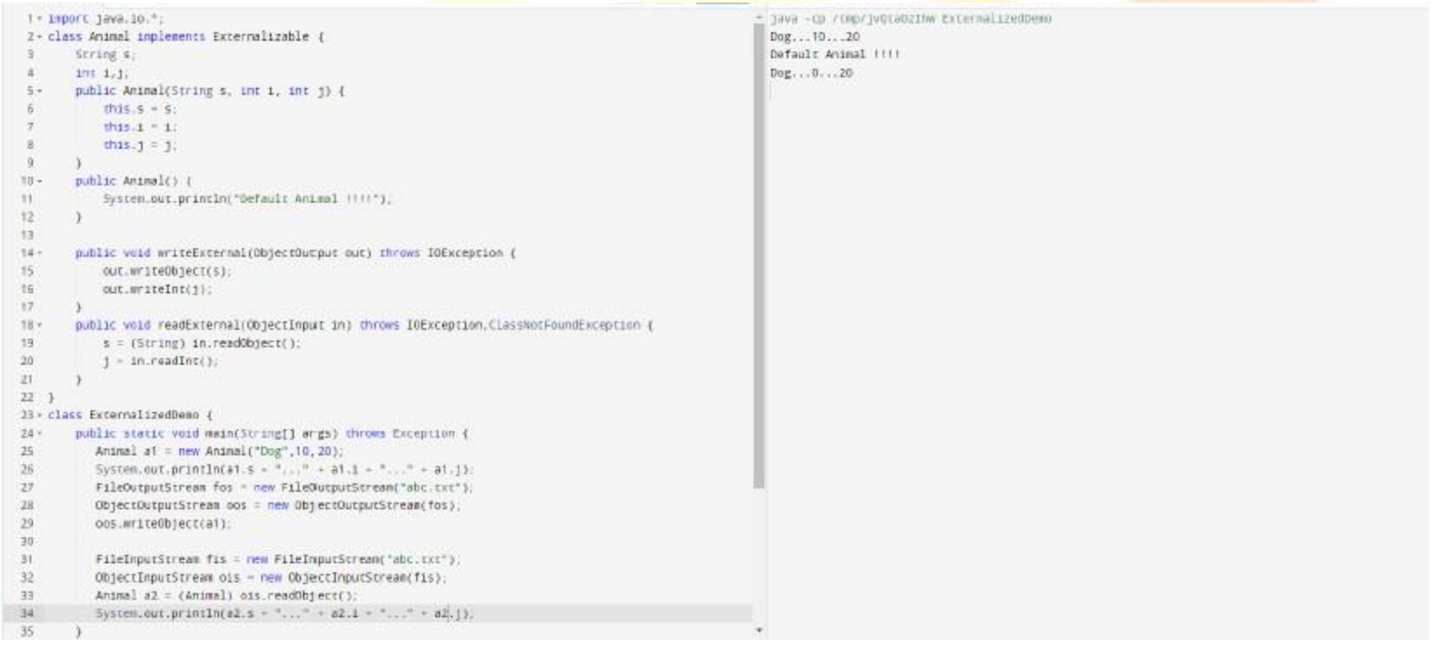
public void writeExternal(ObjectOutput out) throws IOException

This method will be executed automatically at the time of Serialization. Hence, at the time of Serialization if we want to perform serialization of any specific Properties rather than whole object, we have to define that in this method only.

public void readExternal() throws IOException

This method will be executed automatically at the time of Deserialization. Hence, at the time of Deserialization if we want to perform Deserialization of any specific Properties rather than whole object, we have to define that in this method only.

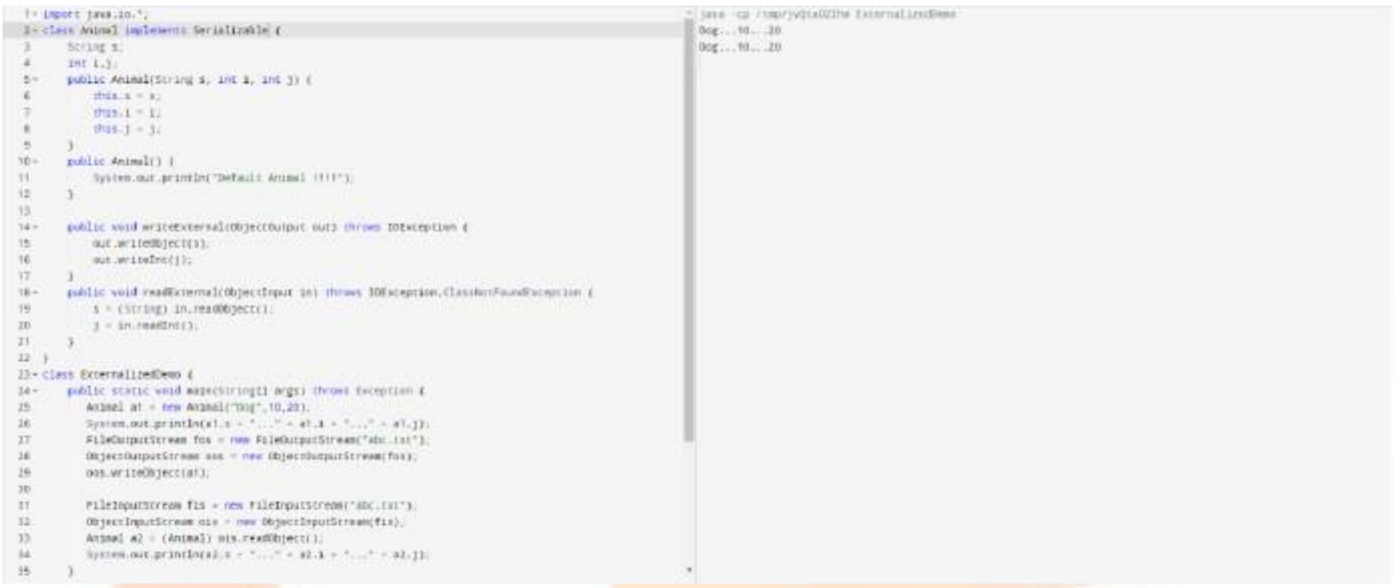
At the time of Deserialization, JVM will create a separate new Object by executing public no argument constructor. On that Object, JVM will call readExternal() method. Hence, every Externalizable implement class should contain public no-argument constructor else we will get a run time exception saying InvalidClassException.



```java
import java.io.*;
class Animal implements Externalizable {
    String s;
    int i,j;
    public Animal(String s, int i, int j) {
        this.s = s;
        this.i = i;
        this.j = j;
    }
    public Animal() {
        System.out.println("Default Animal !!!!");
    }

    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeObject(s);
        out.writeInt(j);
    }
    public void readExternal(ObjectInput in) throws IOException,ClassNotFoundException {
        s = (String) in.readObject();
        j = in.readInt();
    }
}
class ExternalizedDemo {
    public static void main(String[] args) throws Exception {
      Animal a1 = new Animal("Dog",10,20);
      System.out.println(a1.s + "..." + a1.i + "..." + a1.j);
      FileOutputStream fos = new FileOutputStream("abc.txt");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(a1);

      FileInputStream fis = new FileInputStream("abc.txt");
      ObjectInputStream ois = new ObjectInputStream(fis);
      Animal a2 = (Animal) ois.readObject();
      System.out.println(a2.s + "..." + a2.i + "..." + a2.j);
    }
```

```
java -cp /tmp/jvQtaO2IfW ExternalizedDemo
Dog...10...20
Default Animal !!!!
Dog...0...20
```

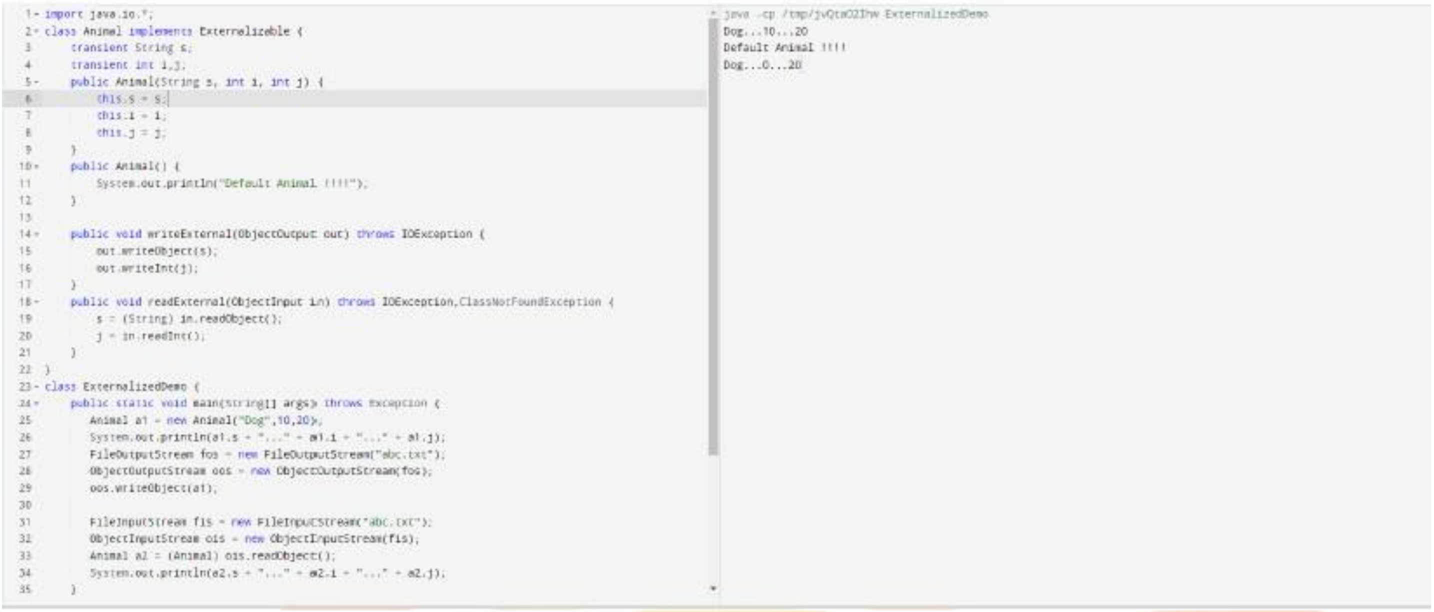
If the class implements Serializable then total object will be saved to the file. In this case output will be :



```java
import java.io.*;
class Animal implements Serializable {
    String s;
    int i,j;
    public Animal(String s, int i, int j) {
        this.s = s;
        this.i = i;
        this.j = j;
    }
    public Animal() {
        System.out.println("Default Animal !!!!");
    }

    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeObject(s);
        out.writeInt(j);
    }
    public void readExternal(ObjectInput in) throws IOException,ClassNotFoundException {
        s = (String) in.readObject();
        j = in.readInt();
    }
}
class ExternalizedDemo {
    public static void main(String[] args) throws Exception {
       Animal a1 = new Animal("Dog",10,20);
       System.out.println(a1.s + "..." + a1.i + "..." + a1.j);
       FileOutputStream fos = new FileOutputStream("abc.txt");
       ObjectOutputStream oos = new ObjectOutputStream(fos);
       oos.writeObject(a1);

       FileInputStream fis = new FileInputStream("abc.txt");
       ObjectInputStream ois = new ObjectInputStream(fis);
       Animal a2 = (Animal) ois.readObject();
       System.out.println(a2.s + "..." + a2.i + "..." + a2.j);
    }
```

```
java -cp /tmp/jvQta02Ihw ExternalizedDemo
Dog...10...20
Dog...10...20
```

Note: transient keyword doesn’t play any role in externalization concept so weather we use or not it doesn’t impact. As transient keyword was used to skip any of the attributes in writing that attribute to file while serialization as it was taken care by JVM. But since Externalization is done by programmer. So, programmer can happily skip any attribute based on requirement.



```java
import java.io.*;
class Animal implements Externalizable {
    transient String s;
    transient int i,j;
    public Animal(String s, int i, int j) {
        this.s = s;
        this.i = i;
        this.j = j;
    }
    public Animal() {
        System.out.println("Default Animal !!!!");
    }

    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeObject(s);
        out.writeInt(j);
    }
    public void readExternal(ObjectInput in) throws IOException,ClassNotFoundException {
        s = (String) in.readObject();
        j = in.readInt();
    }
}
class ExternalizedDemo {
    public static void main(String[] args) throws Exception {
       Animal a1 = new Animal("Dog",10,20);
       System.out.println(a1.s + "..." + a1.i + "..." + a1.j);
       FileOutputStream fos = new FileOutputStream("abc.txt");
       ObjectOutputStream oos = new ObjectOutputStream(fos);
       oos.writeObject(a1);

       FileInputStream fis = new FileInputStream("abc.txt");
       ObjectInputStream ois = new ObjectInputStream(fis);
       Animal a2 = (Animal) ois.readObject();
       System.out.println(a2.s + "..." + a2.i + "..." + a2.j);
    }
```

```
java -cp /tmp/jvQta02Ihw ExternalizedDemo
Dog...10...20
Default Animal !!!!
Dog...0...20
```

7.. SerialVersionUID

In Serialization, both sender and receiver need not be same person, need not to use same machine and need not be from the same location. The persons may be different, machines may be different and locations may be different.

In Serialization, both sender and receiver should have .class file at the beginning only. Just state of object is travelling from sender to receiver.

At the time of Serialization, with every object, Sender side JVM will generate save a unique Identifier based on .class file. At the time of deserialization, receiver side JVM will compare unique identifier associated with object with local class unique identifier. If both are matched then only deserialization will be performed otherwise we will get runtime exception saying InvalidClassException. This unique identifier is nothing but SerialVersionUID.

Problems of depending on default SerialVersionUID generated by JVM:

- Both Sender and Receiver should use same JVM with respect to Vendor, Platform and Version. Otherwise, Receiver will not be able to deserialize because of different SerialVersionUID.
- Both Sender and Receiver should use same .class file version. After Serialization, if there is any change in .class file at receiver side then Receiver will be unable to deserialize.
- To generate SerialVersionUID, internally JVM may use complex algorithm which may create performance problems.