9. Serialization

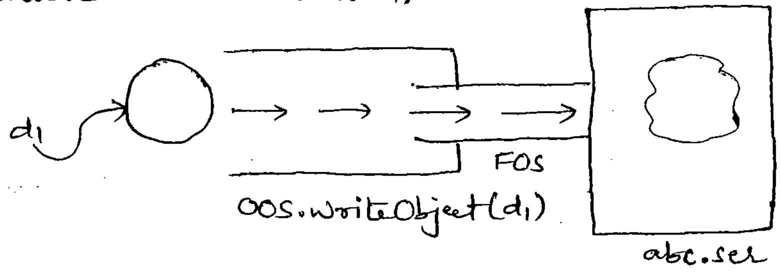
1. Introduction :—

Serialization :—

→ The processing of writing state of an object to a file is called Serialization. But strictly speaking, it is the process of converting an object from Java supported form to either File supported form or Network supported form.

→ By using FileOutputStream and ObjectOutputStream classes we can achieve Serialization.

Ex:

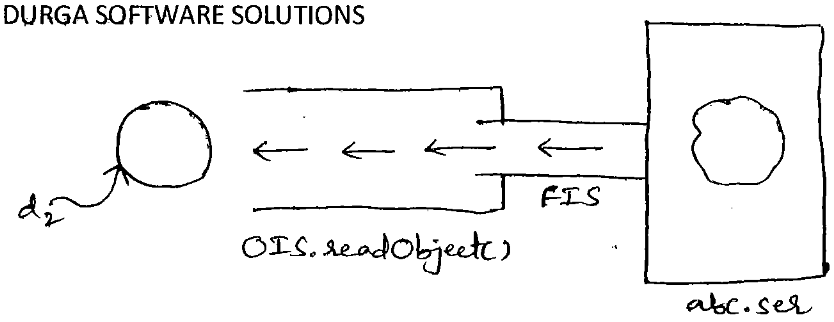


DeSerialization:—

→ The process of reading state of an object from a file is called Deserialization. But strictly speaking it is the process of converting an object from either file or network supported form into Java supported form.

→ By using FileInputStream and ObjectInputStream classes we can achieve Deserialization.

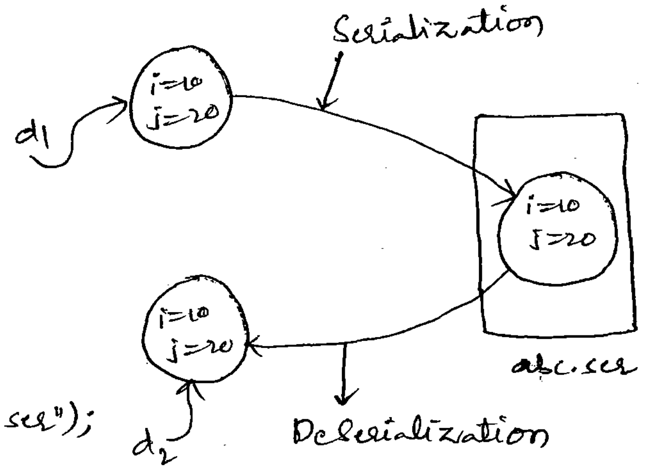
Ex:



Ex:

```
import java.io.*;
class Dog implements Serializable
{ int i=10;
  int j=20;
}
class SerializeDemo
{
  p s v m(-)throws Exception
  {
    Dog d1=new Dog();
                   { FOS fos=new FOS("abc.ser");
    Serialization  { OOS oos=new OOS(fos);
                   { oos.writeObject(d1);

                   { FIS fis=new FIS("abc.ser");
    Deserialization{ OIS ois=new OIS(fis);
                   { Dog d2=(Dog)ois.readObject();
    S.o.p(d2.i+"..."+d2.j);
  }   O/P: 10 ... 20
}
```



→ We can serialize only Serializable objects.

→ An object is said to be Serializable iff the corresponding class implements Serializable interface.

→ Serializable interface present in java.io package & it doesn't contain any methods. It is a marker interface.

→ If we are trying to Serialize a non-Serializable object then we will get runtime exception saying NotSerializableException.

*** transient keyword:—

→ transient is the modifier applicable only for variables.

→ While performing Serialization if we don't want to save the value of a particular variable to meet security constraints such type of variables we have to declare with transient keyword.

→ At the time of Serialization JVM ignores original value of transient variables and save default value to the file.

→ Hence transient means not to Serialize.

static Vs transient :—

→ static variable is not part of object state. Hence it won't participate in Serialization.

→ Due to this declaring static variable as transient there is no use.

final Vs transient :—

→ final variables will be participated in Serialization directly by their values.

→ Due to this declaring final variable as transient there is no use.

Declaration	Output

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=10;	0 . . . 10
transient static int i=10; transient final int j=20;	10 . . . 20

***
→ We can serialize multiple objects to the file. But in which order we serialize in the same order only deserialize.

Ex:
```
Dog d1 =new Dog();
Cat c1 =new cat();
Rat r1 = new Rat();
FOS fos=new FOS("abc.ser");
OOS oos = new OOS(fos);
oos. writeObject(d1);
oos. writeObject(c1);
oos. writeObject (r1);

FIS fis=new FIS("abc.ser");
OIS ois =new OIS (fis);
Dog d2 = (Dog)ois. readObject();
Cat c2 = (Cat) ois. readObject();
Rat r2 = (Rat) ois. readObject();
```
***
→ If we don't know order of objects in Serialization

Ex:
```
FIS fis=new FIS ("abc.ser");
OIS ois =new OIS (fis);
Object o = ~~f~~is. readObject();
```

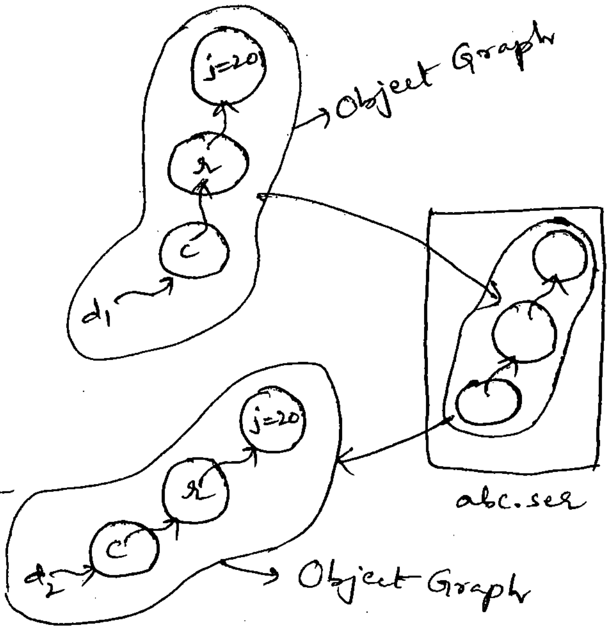
```
if (o instanceof Dog)
{
   Dog d = (Dog)o;
   // perform Dog specific functionality
}
else if (o instanceof Cat)
{
    Cat c = (Cat)o;
   // perform Cat specific functionality
}
|
|
```

2. Object Graphs in Serialization :—

→ Whenever we are serializing an object the set of all objects which are reachable from that object will be Serialized automatically. This group of objects is nothing Object Graph in Serialization.

→ In Object graph, every object should be Serializable. If atleast one object is not Serializable then we will get RE saying NotSerializableException.

Ex:
```
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 SerializeDemo1
{
  p s v m(-)throws Exception
  {
    Dog d1=new Dog();
    FOS fos=new FOS("abc.ser");
    OOS oos=new OOS(fos);
    oos.writeObject(d1);

    FIS fis=new FIS("abc.ser");
    OIS ois=new OIS(fis);
    Dog d2=(Dog)ois.readObject();
    S.o.p(d2.c.r.j); => o/p :20
  }
}
```

→ In the above example, whenever we are serializing Dog object automatically Cat and Rat objects will be serialized becoz these are part of Object graph of Dog object.

→ Among Dog, Cat & Rat if atleast one object is non-serializable then we will get RE saying NotSerializableException.

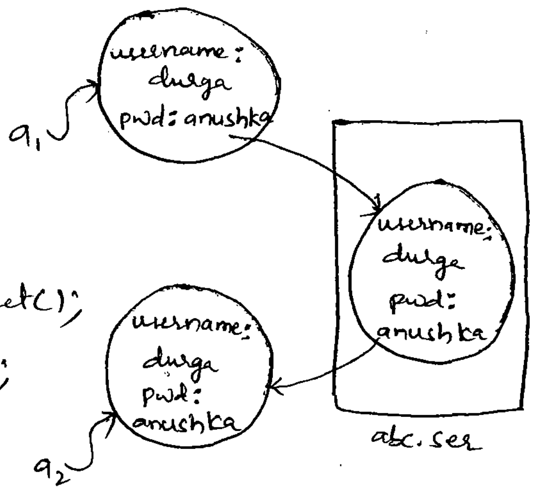
3. Customized Serialization:—

→ During default Serialization there may be a chance of loss of information due to transient keyword.

Ex:
```
import java.io.*;
class Account implements Serializable
{
  String username="durga";
transient String pwd="anushka";
}
```

```
class CustSerializeDemo
{
    p s v m(-)throws Exception
    {
        Account a1 = new Account();
        S.o.p(a1.username+"..."+a1.pwd); ⇒ o/p : durga ... anushka
        FOS fos = new FOS("abc.ser");
        OOS oos = new OOS(fos);
        oos.writeObject(a1);

        FIS fis = new FIS("abc.ser");
        OIS ois = new OIS(fis);
        Account a2 = (Account)ois.readObject();
        S.o.p(a2.username+"..."+a2.pwd);
    }
}       o/p : durga ... null
```



→ In the above example, before Serialization Account object can provide proper username and pwd. But after Deserialization Account object can provide only username, but not pwd.

→ This is due to declaring pwd variable as transient.

→ Hence during default Serialization there may be a chance of loss of information due to transient keyword.

→ To recover this loss of information we should go for Customized Serialization.

→ We can implement Customized Serialization by using the following 2 methods.

① $\boxed{\text{private void writeObject(ObjectOutputStream oos) throws Exception}}$

→ This method will be executed automatically at the time of Serialization. Hence while performing Serialization if we want to do any extra work we have to write code in this method only.

② $\boxed{\text{private void readObject(ObjectInputStream ois) throws Exception}}$

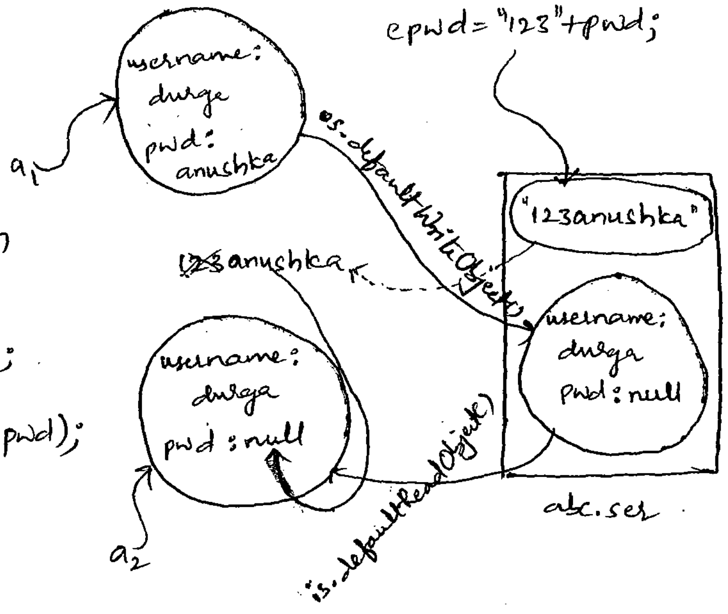
→ This method will be executed automatically at the time of Deserialization. Hence while performing Deserialization if we want to do any extra work we have to define that in this method only.

→ While performing which object Serialization we have to do this extra work in the corresponding class we have to define the above methods.

For Example, while performing Account object Serialization if we required to do extra work then in Account class we have to define above methods.

Ex:①
```
import java.io.*;
class Account implements Serializable
{
  String username="durga";
  transient String pwd="anushka";
  private void writeObject(OOS os) throws Exception
  {
     os.defaultWriteObject();
     String epwd="123"+pwd;
     os.writeObject(epwd);
  }
private void readObject(OIS is) throws Exception
{
   is.defaultReadObject();
    String epwd=(String) is.readObject();
    pwd=epwd.substring(3);
 }
}
class CustSerializeDemo1
{
  p s v m(-) throws Exception
  {
   Account a1=new Account();
   S.o.p(a1.username+"..."+a1.pwd);
   FOS fos=new FOS("abc.ser");
   OOS oos=new OOS(fos);
   oos.writeObject(a1);

    FIS fis=new FIS("abc.ser");
    OIS ois=new OIS(fis);
    Account a2=(Account)ois.readObject();
    S.o.p(a2.username+"..."+a2.pwd);
  }
}
```



```
O/P: durga...anushka
     durga...anushka
```

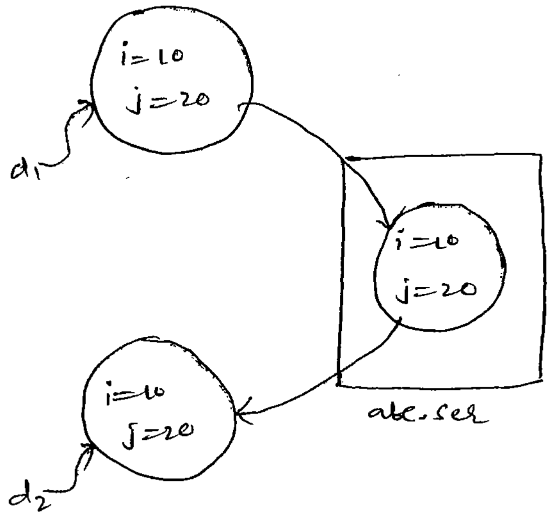
Ex②:
```
 import java.io.*;
 class Account implements Serializable
 {
    String username="durga";
    transient String pwd="anushka";
    taneient int pin =1234;
    private void writeObject (OOS os) throws Exception
    {
        os.defaultWriteObject();
        String epwd = "123"+ pwd;
        os. writeObject(epwd);
        int epin = pin+4444;
        os. writeObject(epin);
    }
    private void readObject (OIS is) throws Exception
    {
        is. defaultReadObject();
        String epwd = (String) is. readObject();
        pwd= epwd. substring(3);
        int epin = is.readInt();
        pin =epin-4444;
    }
}
```

4. Serialization w.r.t Inheritance :-

Case(i): If parent is Serializable then by default every child is Serializable i.e., Serializable nature is inheriting from parent to child. Hence eventhough child class doesn't implement Serializable if parent class implements Serializable then we can serialize child class object.

Ex:
```
    import java. io. *;
    class Animal implements Serializable
    {
        int i=10;
    }
```

```
class Dog extends Animal
{
  int j=20;
}
class InSerializeDemo
{
  p s v m(_) throws Exception
  {
   Dog d1=new Dog();
   FOS fos=new FOS("abc.ser");
   OOS oos=new OOS(fos)
   oos.writeObject(d1);
   FIS fis=new FIS("abc.ser");
   OIS ois=new OIS(fis);
   Dog d2=(Dog)ois.readObject();
   S.o.p(d2.i+"..."+d2.j);
  }
}   O/P: 10 ... 20
```



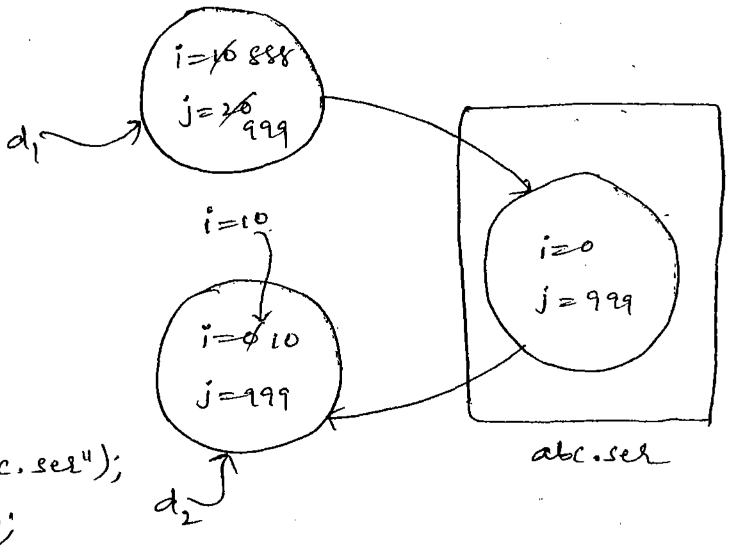
Case ii):

1. Eventhough parent class doesn't implement Serializable interface we can serialize child class object if child class implements Serializable. i.e.,
2. At the time of Serialization JVM will check is any instance variable is inheriting from non-Serializable parent or not. If any variable is inheriting from non-Serializable parent then JVM ignores original value & save default value to the file.
3. At the time of Deserialization JVM will check is any parent class is non-Serializable or not. If any parent class is non-Serializable then execute Instance Control Flow in that

non-Serializable parent & share its instance variables to the current object.

4. In Instance Control Flow execution of non-Serializable parent Jvm will always invoke no-argument constructor. Hence every non-Serializable class should compulsory contain no-argument constructor, o.w we will get RE saying InvalidClassException.

Ex:
```
import java.io.*;
class Animal
{
  int i=10;
  Animal()
  {
   s.o.p("Animal constructor called");
  }
}
class Dog extends Animal implements Serializable
{
  int j=20;
  Dog()
  {
   s.o.p("Dog constructor called");
  }
}
class InSerializeDemo1
{
  p s v m(-)
  {
   Dog d1=new Dog();
   d1.i=888;
   d1.j=999;
   FOS fos=new FOS("abc.ser");
   OOS oos=new OOS(fos);
   oos.writeObject(d1);
```



```
S.o.p("DeSerialization Started");
FIS fis=new FIS("abc.ser");
OIS ois = new OIS(fis);
Dog d2 = (Dog)ois.readObject();
S.o.p(d2.i+"..."+d2.j);
}
}
```

o/p:
```
Animal constructor called
Dog Constructor called
DeSerialization Started
Animal constructor called
10...999
```

***

5. Externalization:—

→ In Serialization, everything takes care by JVM & programmer doesn't have any control.

→ In Serialization, total object will be serialized always & it is not possible to serialize part of the object, which may creates performance problems in some cases.

→ To overcome these problems we should go for Externalization, where everything takes care by programmer & JVM doesn't have any control.

→ The advantage of Externalization is based on our requirement we can save either total object or part of the object. So that relatively performance will be improved.

→ To provide Externalizable ability for any Java object compulsory the corresponding class should implement Externalizable interface.

→ Externalizable interface is the child interface of Serializable & it contains 2 methods are writeExternal() & readExternal()

$$\begin{tikzcd} \text{Serializable (I)} \arrow[r] & 1.1V \\ \text{Externalizable (I)} \arrow[u] \arrow[r] & 1.1V \end{tikzcd}$$

① $\boxed{\text{public void writeExternal(ObjectOutput out) throws IOException}}$

→ This method will be executed automatically at the time of Serialization

→ Within this method we have to write code to save required variables to the file.

② $\boxed{\text{public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException}}$

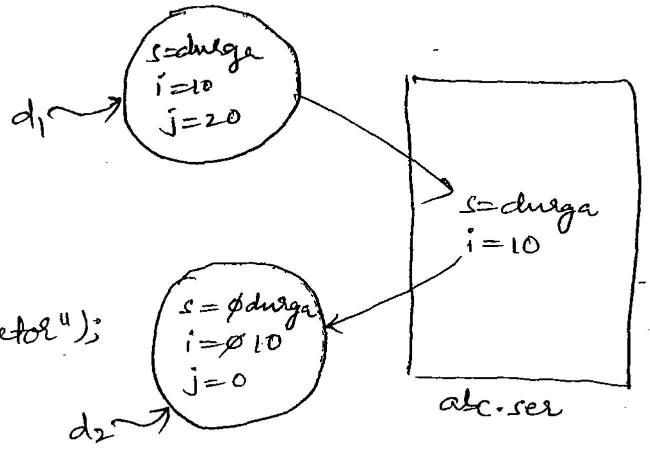
→ This method will be executed automatically at the time of DeSerialization.

→ Within this method we have to write code to read required variables from the file & assign to the current object.

→ Strictly speaking at the time of DeSerialization JVM will create a separate new object by executing public no-argument constructor, on that object readExternal(-) method will be executed.

→ Externalizable class should compulsory contains public no-argument constructor otherwise, we will get RE saying InvalidClassException.

ex:
```
import java.io.*;
public class ExternalizableDemo implements Externalizable
{
  String s;
  int i;
  int j;
  public ExternalizableDemo
  {
   S.o.p ("public no-arg constructor");
  }
```



```
public ExternalizableDemo (String s, int i, int j)
{
    this.s=s;
    this.i=i;
    this.j=j;
}
public void writeExternal (ObjectOutput out) throws IOException
{
    out.writeObject(s);
    out.writeInt(i);
}
public void readExternal (ObjectInput in) throws IOException,
                                      ClassNotFoundException
{
    s=(String) in.readObject();
    i=in.readInt();
}
P s v m(-) throw Exception
{
    ExternalizableDemo d1=new ExternalizableDemo("durga", 10, 20);
    FOS fos=new FOS("abc.ser");
    OOS oos=new OOS(fos);
    oos.writeObject(d1);
    FIS fis=new FIS("abc.ser");
    OIS ois =new OIS(fis);
    ExternalizableDemo d2=(ExternalizableDemo) ois.readObject();
    S.o.p(d2.s+"..."+d2.i+"..."+d2.j);
}
}
```

→ If the class implements <u>Externalizable</u> interface then the <u>o/p</u> is.

```
public no-arg constructor
durga...10...0
```

→ If the class implements Serializable interface then the o/p is $\boxed{\text{durga...10...20}}$

Note:- In Externalization, transient keyword won't play any role ofcourse it is not required.

***Differences b/w Serialization and Externalization:-

Serialization	Externalization
1. It is meant for default Serialization.	1. It is meant for customized Serialization.
2. Here everything takes care by Jvm and programmer doesn't have any control.	2. Here everything takes care by programmer and Jvm doesn't have any control.
3. In Serialization, total object will be serialized always & it is not possible to serialize part of the object.	3. In Externalization, based on our requirement we can save either total object or part of the object.
4. Relatively performance is low.	4. Rativelg performance is high.
5. Serialization is the best choice if we want to save total object to the file.	5. Externalization is the best choice if we want to save part of the objet to the file.
6. Serializable interface doesn't contain any methods & it is a marker interface.	6. Externalizable Interface contains 2 methods. writeExternal(-) and readExternal(-). It is not a marker interface.

Serialization	Externalization
7. Serializable class is not required to contain public no-argument constructor.	7. Externalizable class should compulsory contain public no-argument constructor o.w, we will get RE saying InvalidClassException.
8. transient keyword will play role in Serialization.	8. tansient keyword won't play any role in Externalization.

***
6. serialVersionUID:—

→ To perform Serialization & DeSerialization internally JVM will use a unique identifier, which is nothing but serialVersionUID.

→ At the time of Serialization JVM will save serialVersionUID.

→ At the time of DeSerialization JVM will compare serialVersionUID's. and if it is matched then only the object will be deserialized o.w, we will get RE saying InvalidClassException.

***
The problems in depending on Default serialVersionUID:—

1. After Serialization if we change .class file at server side then we can't perform DeSerialization becoz of mismatch in serialVersionUID's of local class and serialized object.

In this case, at the time of DeSerialization we will get RE saying InvalidClassException.

2. Both Sender and Receiver should use same version of JVM, if there is any incompatibily in JVM versions then Receiver is unable to deserialize

In this case, also, Receiver will get RE saying InvalidClassException.

3. To generate serialVersionUID internally JVM will use complex algorithm, which may creates performance problems.

→ We can solve above problems by configuring our own serialVersionUID.

→ We can configure serialVersionID as follows.

$\boxed{\text{private static final long serialVersionUID;}}$

Ex:
```
import java.io.*;
class Dog1 implements Serializable
{
  private static final long serialVersionUID = 1L;
  int i=10;
  int j=20;
}
```

Sender.java:—
```
import java.io.*;
class Sender
{
  p s v m(-) throws Exception
  {
    Dog1 d1=new Dog1();
    FOS fos=new FOS("abc.ser");
    OOS oos=new OOS(fos);
    oos.writeObject(d1);
  }
}
```

Receiver.java:—
```
import java.io.*;
class Receiver
{
```

```
p  s  v  m(-) throws Exception
{
    FIS fis=new FIS("abc.ser");
    OIS ois = new OIS(fis);
    Dog1 d2 = (Dog1) ois.readObject();
    S.o.p(d2.i+"..."+d2.j);
}
     O/P: 10...20.
}
```

→ In the above program, after Serialization eventhough if we are performing any change to the .class file we can deserialize object.

→ If we configure our own serialVersionUID both Sender & Receiver are not required to maintain same JVM versions.

Note①:- Some IDE's explicitly prompt the programmer to enter serialVersion UID.

② some IDE's explicitly generates serialVersionUID automatically instead of depending on JVM generated default serialVersionUID.