Serialization (1.1v):

The process 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 into either file supported form or network supported form. By using FileOutputStream and ObjectOutputStream classes we can achieve or we can implement serialization.

Serialization is the process of converting an object into a byte stream, which can then be saved to a file, transmitted over a network, or stored in a database. This byte stream typically contains the object's data and metadata needed to reconstruct the object later. In Java, you use the `Serializable` interface to mark a class as serializable. When an object of a serializable class is serialized, it can be saved as a file, sent over a network, or otherwise stored.

- It is used to persist an object's state.

- Serializable objects can be written to an `ObjectOutputStream`.

- Serialization is primarily used for data storage and data transmission.

- Serialized data is typically platform and language-independent, allowing objects to be reconstructed on different systems.

Deserialization:

The process of reading state of an object from the file is called deserialization but strictly speaking it is the process of converting an object from either file supported form or network supported form into java supported form. By using FileInputStream and ObjectInputStream we can implement deserialization.

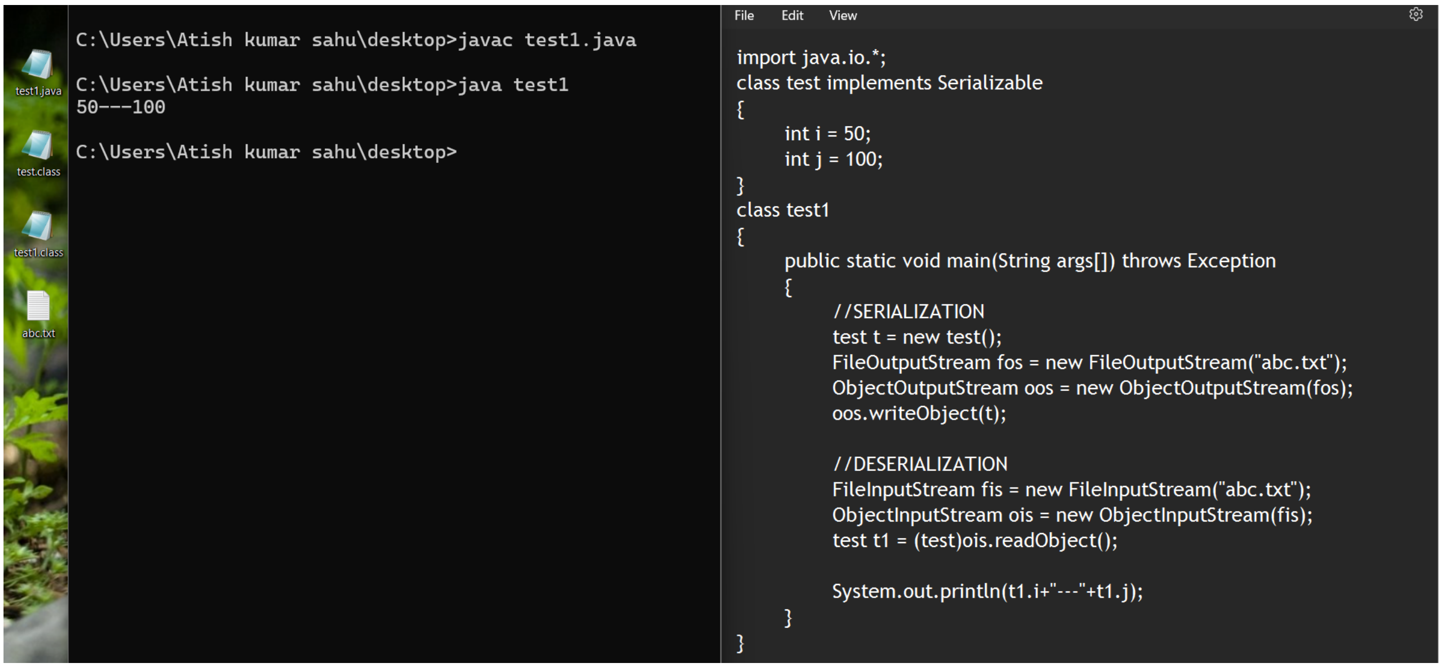
Deserialization is the process of reconstructing an object from a byte stream, which was created through serialization. During deserialization, the byte stream is read, and an object is reconstituted from the data and metadata within the stream. In Java, you use an `ObjectInputStream` to deserialize an object. The deserialized object will be an instance of the original class from which it was serialized.

- It is used to recreate an object's state from a previously serialized representation.

- Deserializable objects can be read from an `ObjectInputStream`.

- Deserialization is primarily used to read data from storage or receive data from the network and reconstruct objects in memory.

We can serialize only serializable objects. An object is set to be serializable if and only if the corresponding class implements serializable interface. Serializable interface present in java.io package and it doesn't contain any methods. It is a marker interface. If we are trying to serialize an non serializable object then we will get runtime exception saying "***NotSerializableException***".

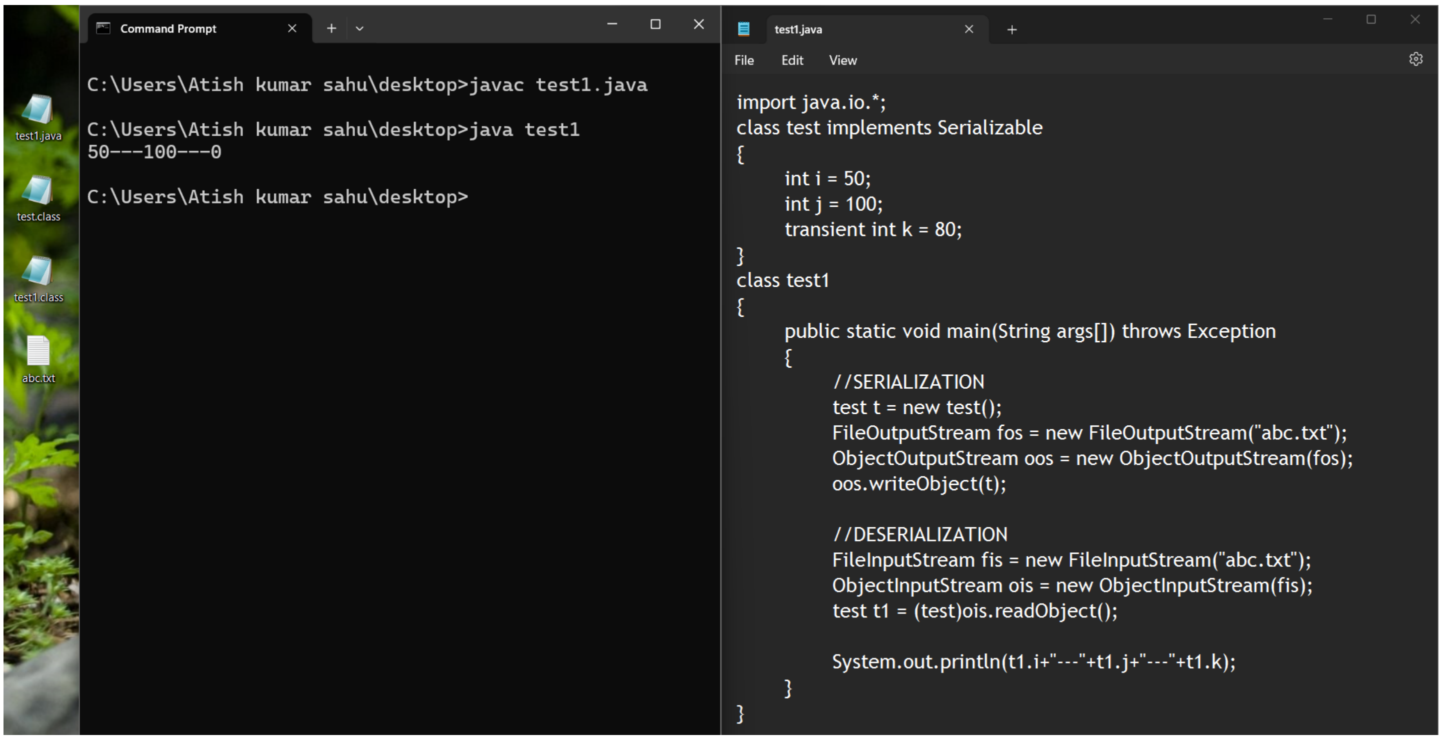


```
C:\Users\Atish kumar sahu\desktop>javac test1.java

C:\Users\Atish kumar sahu\desktop>java test1
50---100

C:\Users\Atish kumar sahu\desktop>
```

```
import java.io.*;
class test implements Serializable
{
	int i = 50;
	int j = 100;
}
class test1
{
	public static void main(String args[]) throws Exception
	{
		//SERIALIZATION
		test t = new test();
		FileOutputStream fos = new FileOutputStream("abc.txt");
		ObjectOutputStream oos = new ObjectOutputStream(fos);
		oos.writeObject(t);

		//DESERIALIZATION
		FileInputStream fis = new FileInputStream("abc.txt");
		ObjectInputStream ois = new ObjectInputStream(fis);
		test t1 = (test)ois.readObject();

		System.out.println(t1.i+"---"+t1.j);
	}
}
```

Transient Keyword:

Transient modifier 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 save default value to the file. Hence transient means not to serialize.

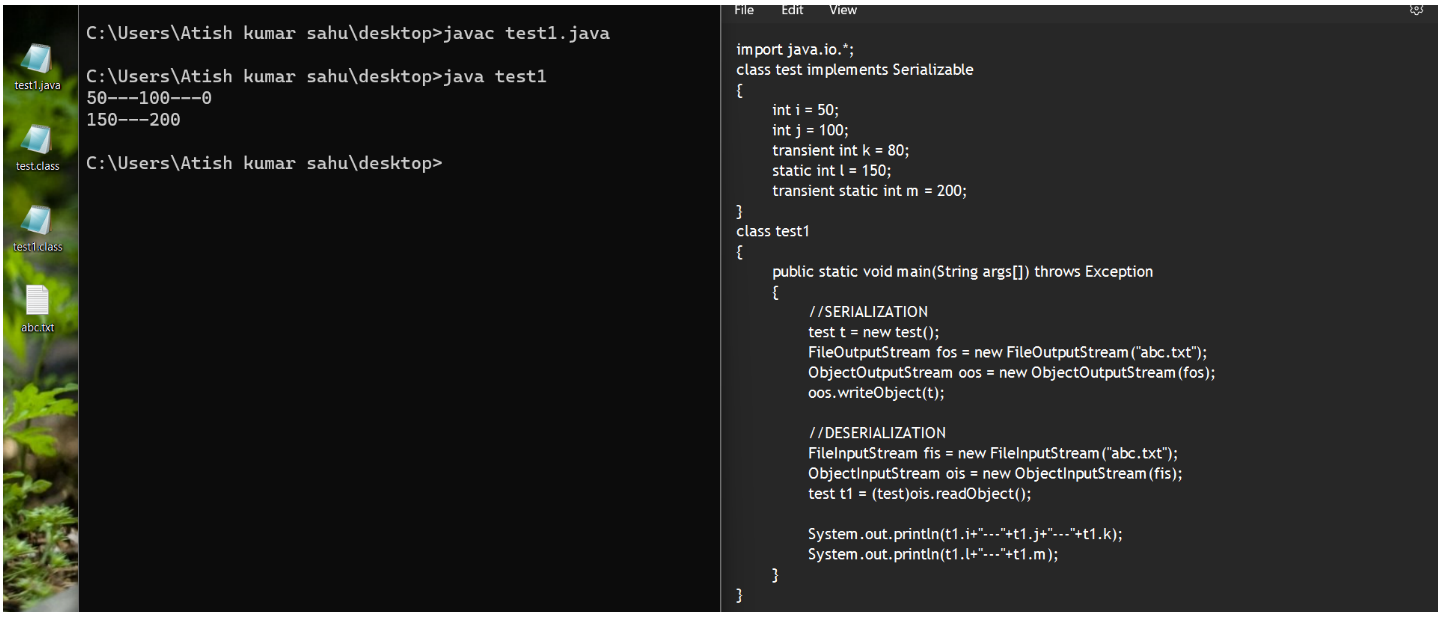


```
C:\Users\Atish kumar sahu\desktop>javac test1.java

C:\Users\Atish kumar sahu\desktop>java test1
50---100---0

C:\Users\Atish kumar sahu\desktop>
```

```
import java.io.*;
class test implements Serializable
{
	int i = 50;
	int j = 100;
	transient int k = 80;
}
class test1
{
	public static void main(String args[]) throws Exception
	{
		//SERIALIZATION
		test t = new test();
		FileOutputStream fos = new FileOutputStream("abc.txt");
		ObjectOutputStream oos = new ObjectOutputStream(fos);
		oos.writeObject(t);

		//DESERIALIZATION
		FileInputStream fis = new FileInputStream("abc.txt");
		ObjectInputStream ois = new ObjectInputStream(fis);
		test t1 = (test)ois.readObject();

		System.out.println(t1.i+"---"+t1.j+"---"+t1.k);
	}
}
```

Transient Vs Static:

Static variable is not part of object state and hence it won't participate in serialization. Due to this declaring static variable as transient there is no use.

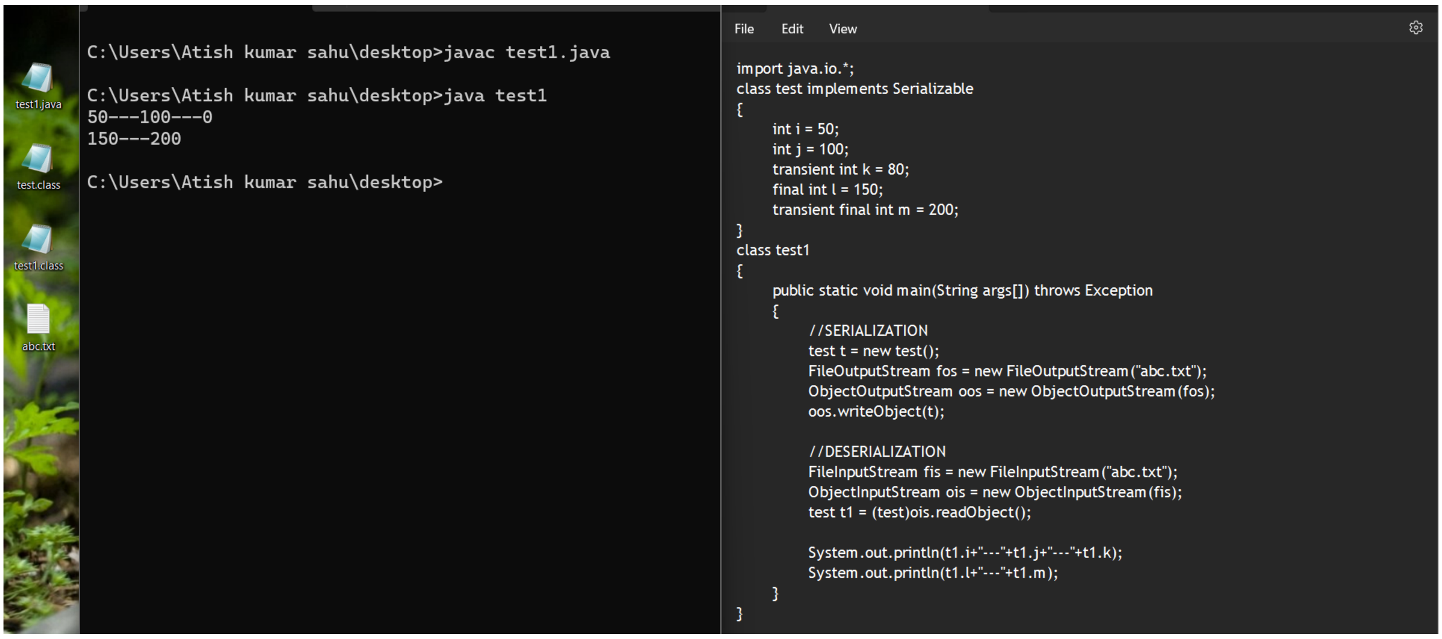


```
C:\Users\Atish kumar sahu\desktop>javac test1.java

C:\Users\Atish kumar sahu\desktop>java test1
50---100---0
150---200

C:\Users\Atish kumar sahu\desktop>
```

```
import java.io.*;
class test implements Serializable
{
        int i = 50;
        int j = 100;
        transient int k = 80;
        static int l = 150;
        transient static int m = 200;
}
class test1
{
        public static void main(String args[]) throws Exception
        {
                //SERIALIZATION
                test t = new test();
                FileOutputStream fos = new FileOutputStream("abc.txt");
                ObjectOutputStream oos = new ObjectOutputStream(fos);
                oos.writeObject(t);

                //DESERIALIZATION
                FileInputStream fis = new FileInputStream("abc.txt");
                ObjectInputStream ois = new ObjectInputStream(fis);
                test t1 = (test)ois.readObject();

                System.out.println(t1.i+"---"+t1.j+"---"+t1.k);
                System.out.println(t1.l+"---"+t1.m);
        }
}
```

Final Vs Transient;

Final variables will be participate in serialization directly by the value. hence declaring a final variable as transient there is no impact.



```
C:\Users\Atish kumar sahu\desktop>javac test1.java

C:\Users\Atish kumar sahu\desktop>java test1
50---100---0
150---200

C:\Users\Atish kumar sahu\desktop>
```

```
import java.io.*;
class test implements Serializable
{
        int i = 50;
        int j = 100;
        transient int k = 80;
        final int l = 150;
        transient final int m = 200;
}
class test1
{
        public static void main(String args[]) throws Exception
        {
                //SERIALIZATION
                test t = new test();
                FileOutputStream fos = new FileOutputStream("abc.txt");
                ObjectOutputStream oos = new ObjectOutputStream(fos);
                oos.writeObject(t);

                //DESERIALIZATION
                FileInputStream fis = new FileInputStream("abc.txt");
                ObjectInputStream ois = new ObjectInputStream(fis);
                test t1 = (test)ois.readObject();

                System.out.println(t1.i+"---"+t1.j+"---"+t1.k);
                System.out.println(t1.l+"---"+t1.m);
        }
}
```

Summary Table:

Declaration	Output
Int I = 10;	10
Int j = 20;	20
transient int I = 10;	0
Int j = 20;	20
Int I = 10;	10
transient int j = 20;	20
transient int I = 10;	0
transient static int j = 20;	20
transient final int I = 10;	10
transient int j = 20;	0
transient static int I = 10;	10
transient final int j = 20;	20

We can serialize any number of objects to the files but in which order serialized in the same order only we have to deserialize. That is order of object is important in serialization.

```
Dog d1 = new Dog();

Cat c1 = new Cat();

Rat r1 = new Rat();

FOS f = new FOS(abc.txt);                FOS = FileOutputStream

OOS o = new OOS(f);                      OOS = ObjectOutputStream

OOS.writeObject(d1);

OOS.writeObject(c1);

OOS.writeObject(r1);

FIS f = new FIS(abc.txt);                FIS = FileInputStream

OIS o = new OIS(f);                      OIS = ObjectInputStream

Dog d2 = (Dog) o.readObject();

Cat c2 = (Cat) o.readObject();

Rat r2 = (Rat) o.readObject();
```

Q. If we don’t know order of objects in serialization what we do?

FIS f = new FIS(abc.txt);

OIS o = new OIS(f);

Object oo = o.readObject();

If(oo instanceof Dog)

{

Dog d2 = (Dog)oo;

}

Else if(oo instance of)

{

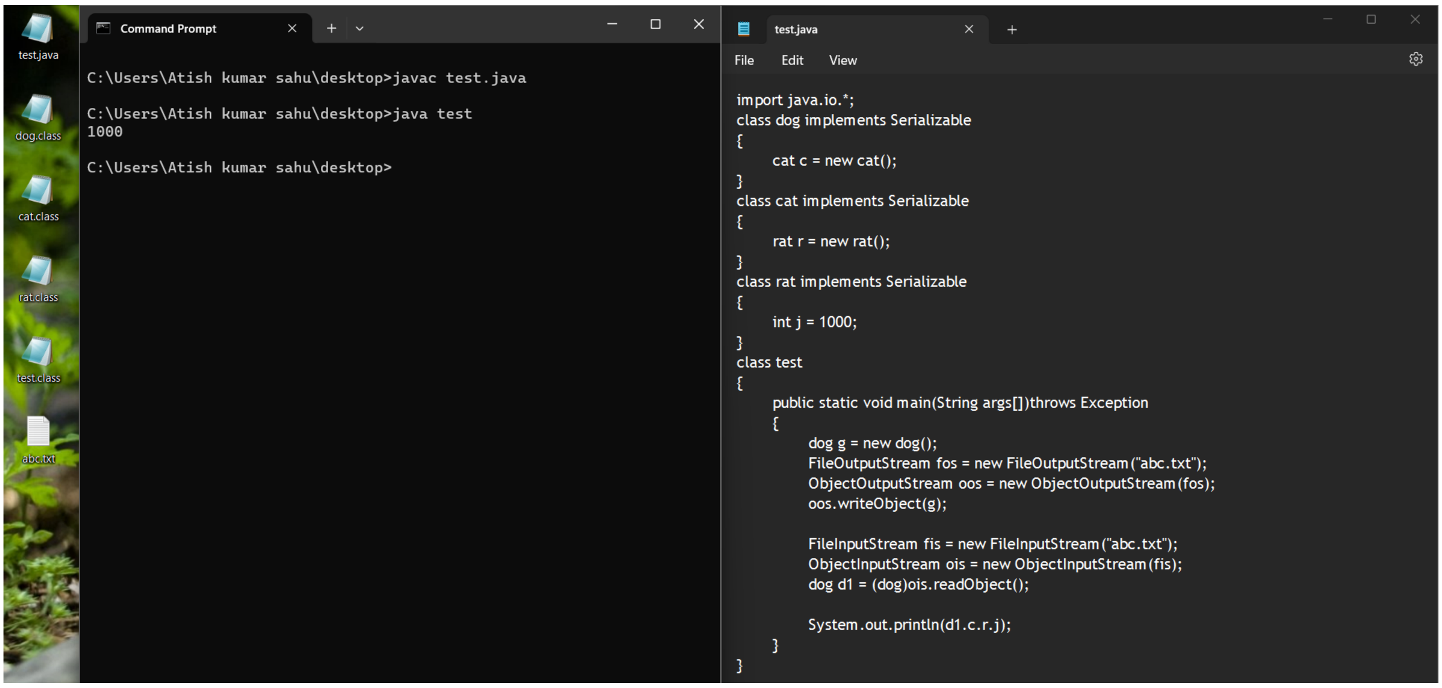
Cat c2 = (Cat)oo;

}

……….

Object Graphs In Serialization:

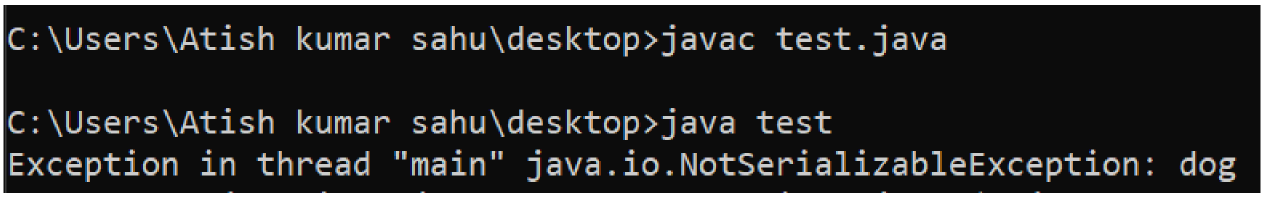
Whenever we are serializing an object, the setup all objects which are reachable from that object will be serializable automatically. This group of objects is nothing but object graph. In object graph every graph should be serializable if at least one object is not serializable then we will get runtime exception saying “***NotSerializableException***”.



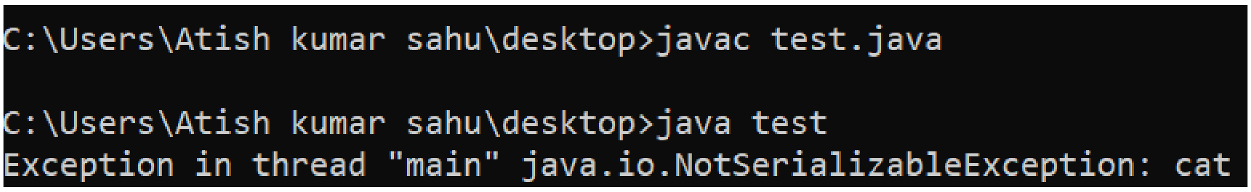
```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
1000

C:\Users\Atish kumar sahu\desktop>
```

```
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 = 1000;
}
class test
{
	public static void main(String args[])throws Exception
	{
		dog g = new dog();
		FileOutputStream fos = new FileOutputStream("abc.txt");
		ObjectOutputStream oos = new ObjectOutputStream(fos);
		oos.writeObject(g);

		FileInputStream fis = new FileInputStream("abc.txt");
		ObjectInputStream ois = new ObjectInputStream(fis);
		dog d1 = (dog)ois.readObject();

		System.out.println(d1.c.r.j);
	}
}
```

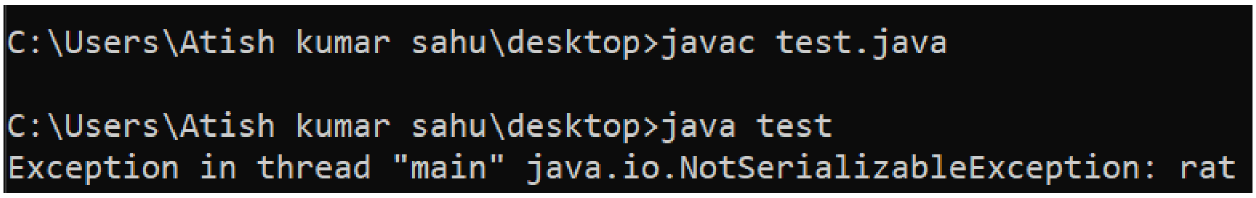
If you not using “implements Runnable” to the class which are going to serializable then the exception are like this



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
Exception in thread "main" java.io.NotSerializableException: dog
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
Exception in thread "main" java.io.NotSerializableException: cat
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
Exception in thread "main" java.io.NotSerializableException: rat
```

In the above program whenever we are serializing dog object automatically cat and rat objects got serialized because these are part of object graph of dog. Among dog, cat and rat object if at least one object is not serializable then we will get runtime exception saying "***NotSerializableException***".

Customized Serialization:

During default serialization there may be a chance of loss of information because of transient keyword

In the above example before serialization account object can provide proper name and password but after deserialization account object can provide only username but not password this is due to declaring password variable as transient. Hence during default serialization there may be a chance of loss of information because of transient keyword. To recover the loss of information we should go for customized serialization.

We can implement customized serialization by using the following two methods

Private void writeObject(ObjectOutputStream os)throws exception

The above method will be executed automatically at the time of serialization hence yet the time of serialization if we want to perform any activity we have to define that in this method only.

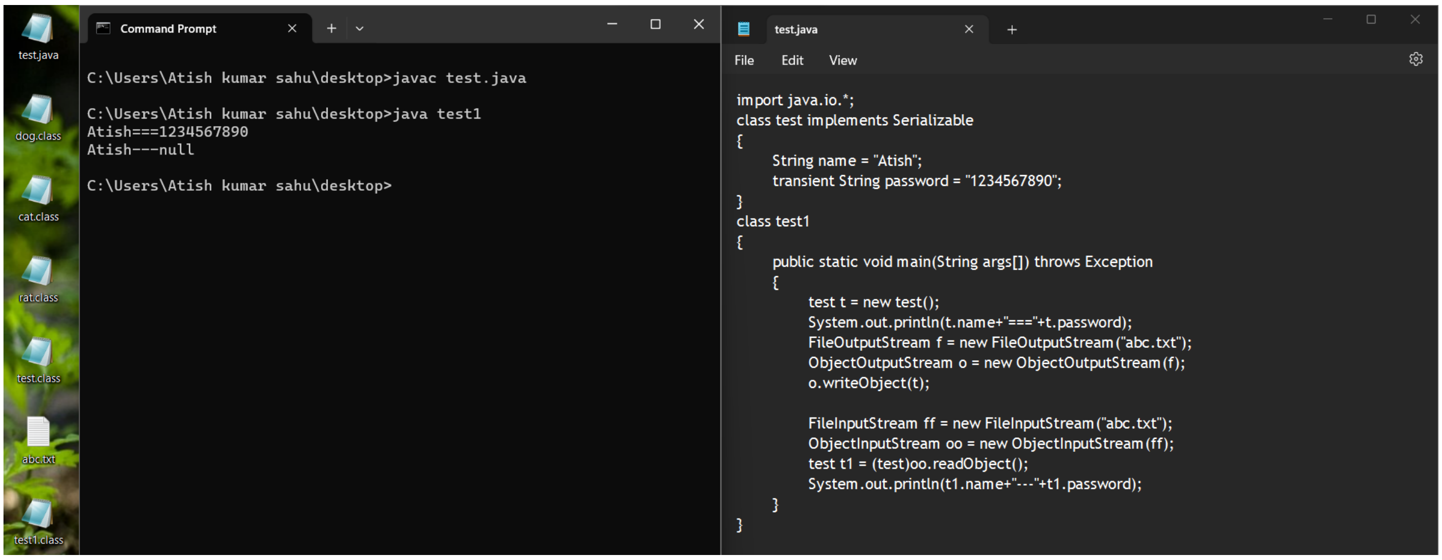
Private void readObject(ObjectInputStream is)throws exception

This above method will be executed automatically at the time of deserialization hence yet the time of deserialization if we want to perform any activity we have to define that in this method only.

THE ABOVE TWO METHODS ARE CALL BACK METHODS BECAUSE THESE ARE EXECUTED AUTOMATICALLY BY THE JVM.

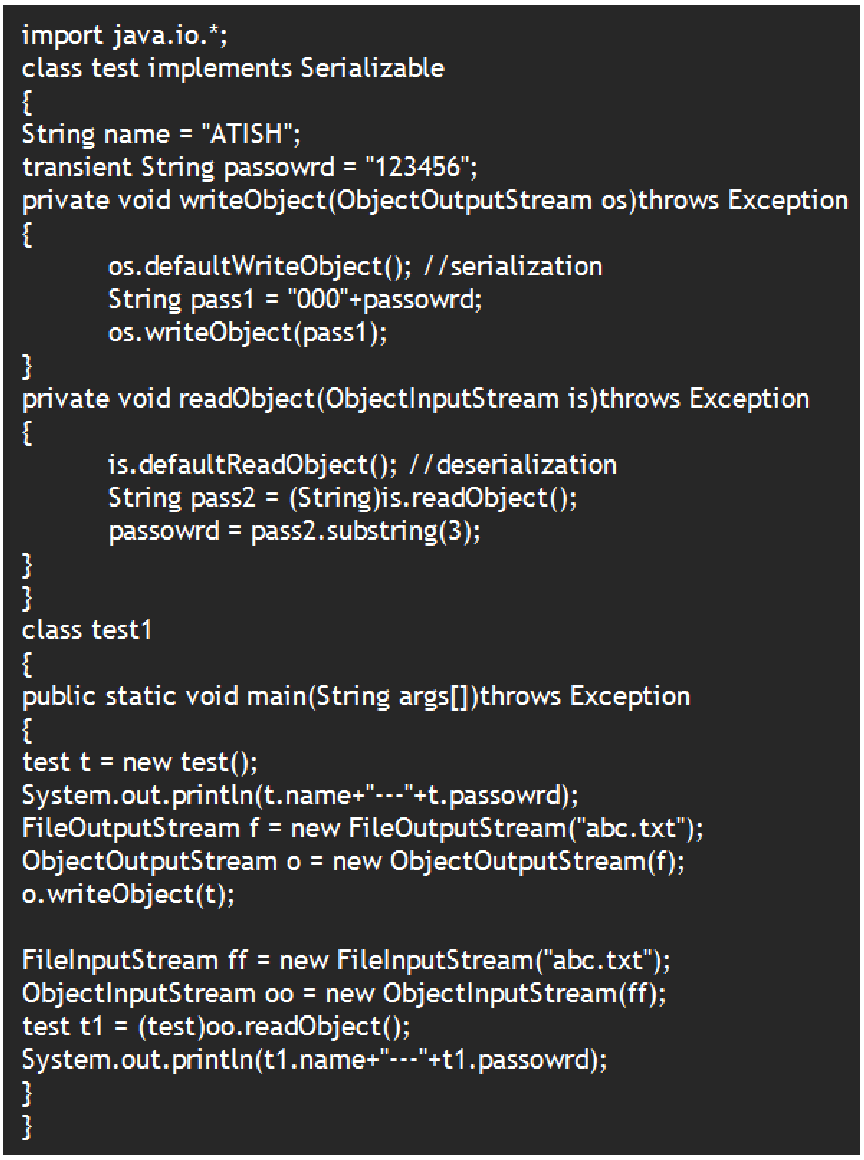
WHILE PERFORMING WHICH OBJECT SERIALIZATION WE HAVE TO DO EXTRA WORK IN THE CORRESPONDING CLASS WE HAVE TO DEFINE ABOVE TWO METHDOS. EX: WHILE

PERFORMING ACCOUNT OBJECT SERIALIZATION IF WE REQUIRED TO DO EXTRA WORK IN THE ACCOUNT CLASS WE HAVE TO DEFINE ABOVE METHODS.

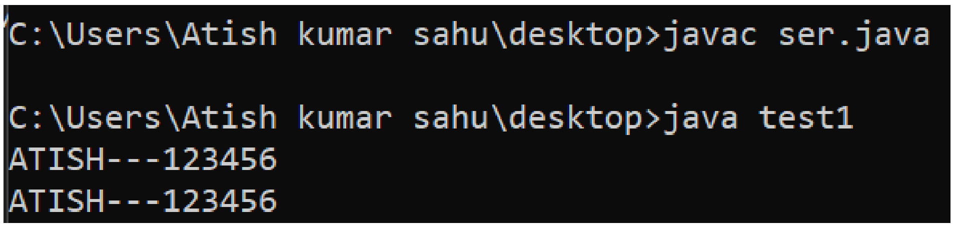


```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test1
Atish===1234567890
Atish---null

C:\Users\Atish kumar sahu\desktop>
```

```
import java.io.*;
class test implements Serializable
{
        String name = "Atish";
        transient String password = "1234567890";
}
class test1
{
        public static void main(String args[]) throws Exception
        {
                test t = new test();
                System.out.println(t.name+"==="+t.password);
                FileOutputStream f = new FileOutputStream("abc.txt");
                ObjectOutputStream o = new ObjectOutputStream(f);
                o.writeObject(t);

                FileInputStream ff = new FileInputStream("abc.txt");
                ObjectInputStream oo = new ObjectInputStream(ff);
                test t1 = (test)oo.readObject();
                System.out.println(t1.name+"---"+t1.password);
        }
}
```



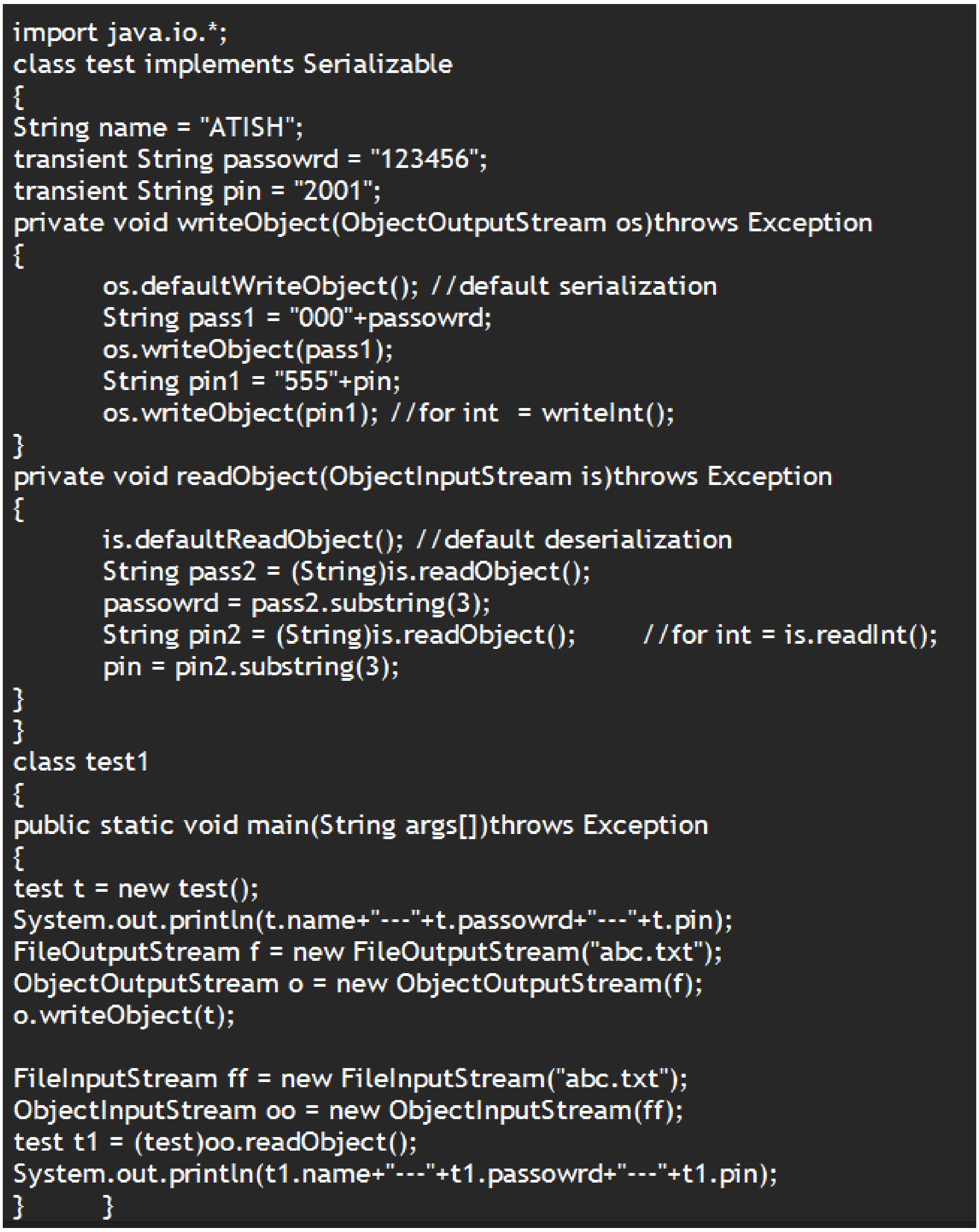
```
import java.io.*;
class test implements Serializable
{
String name = "ATISH";
transient String passowrd = "123456";
private void writeObject(ObjectOutputStream os)throws Exception
{
        os.defaultWriteObject(); //serialization
        String pass1 = "000"+passowrd;
        os.writeObject(pass1);
}
private void readObject(ObjectInputStream is)throws Exception
{
        is.defaultReadObject(); //deserialization
        String pass2 = (String)is.readObject();
        passowrd = pass2.substring(3);
}
}
class test1
{
public static void main(String args[])throws Exception
{
test t = new test();
System.out.println(t.name+"---"+t.passowrd);
FileOutputStream f = new FileOutputStream("abc.txt");
ObjectOutputStream o = new ObjectOutputStream(f);
o.writeObject(t);

FileInputStream ff = new FileInputStream("abc.txt");
ObjectInputStream oo = new ObjectInputStream(ff);
test t1 = (test)oo.readObject();
System.out.println(t1.name+"---"+t1.passowrd);
}
}
```



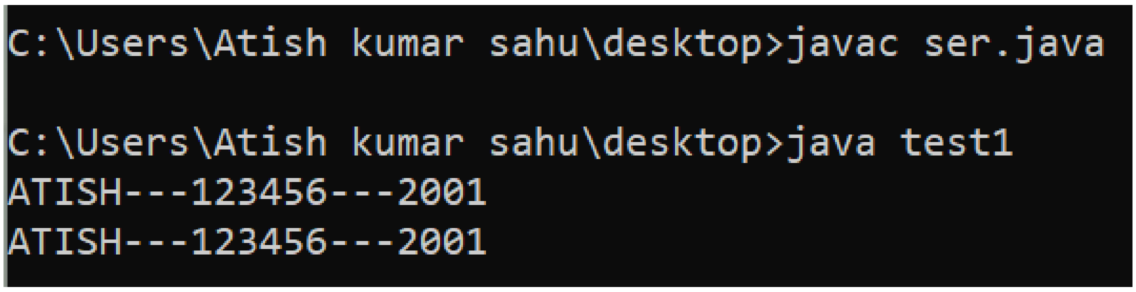
```
C:\Users\Atish kumar sahu\desktop>javac ser.java

C:\Users\Atish kumar sahu\desktop>java test1
ATISH---123456
ATISH---123456
```

In the above program before serialization and after serialization test object can provide proper username 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.



```
import java.io.*;
class test implements Serializable
{
String name = "ATISH";
transient String passowrd = "123456";
transient String pin = "2001";
private void writeObject(ObjectOutputStream os)throws Exception
{
        os.defaultWriteObject(); //default serialization
        String pass1 = "000"+passowrd;
        os.writeObject(pass1);
        String pin1 = "555"+pin;
        os.writeObject(pin1); //for int  = writeInt();
}
private void readObject(ObjectInputStream is)throws Exception
{
        is.defaultReadObject(); //default deserialization
        String pass2 = (String)is.readObject();
        passowrd = pass2.substring(3);
        String pin2 = (String)is.readObject();      //for int = is.readInt();
        pin = pin2.substring(3);
}
}
class test1
{
public static void main(String args[])throws Exception
{
test t = new test();
System.out.println(t.name+"---"+t.passowrd+"---"+t.pin);
FileOutputStream f = new FileOutputStream("abc.txt");
ObjectOutputStream o = new ObjectOutputStream(f);
o.writeObject(t);

FileInputStream ff = new FileInputStream("abc.txt");
ObjectInputStream oo = new ObjectInputStream(ff);
test t1 = (test)oo.readObject();
System.out.println(t1.name+"---"+t1.passowrd+"---"+t1.pin);
}     }
```

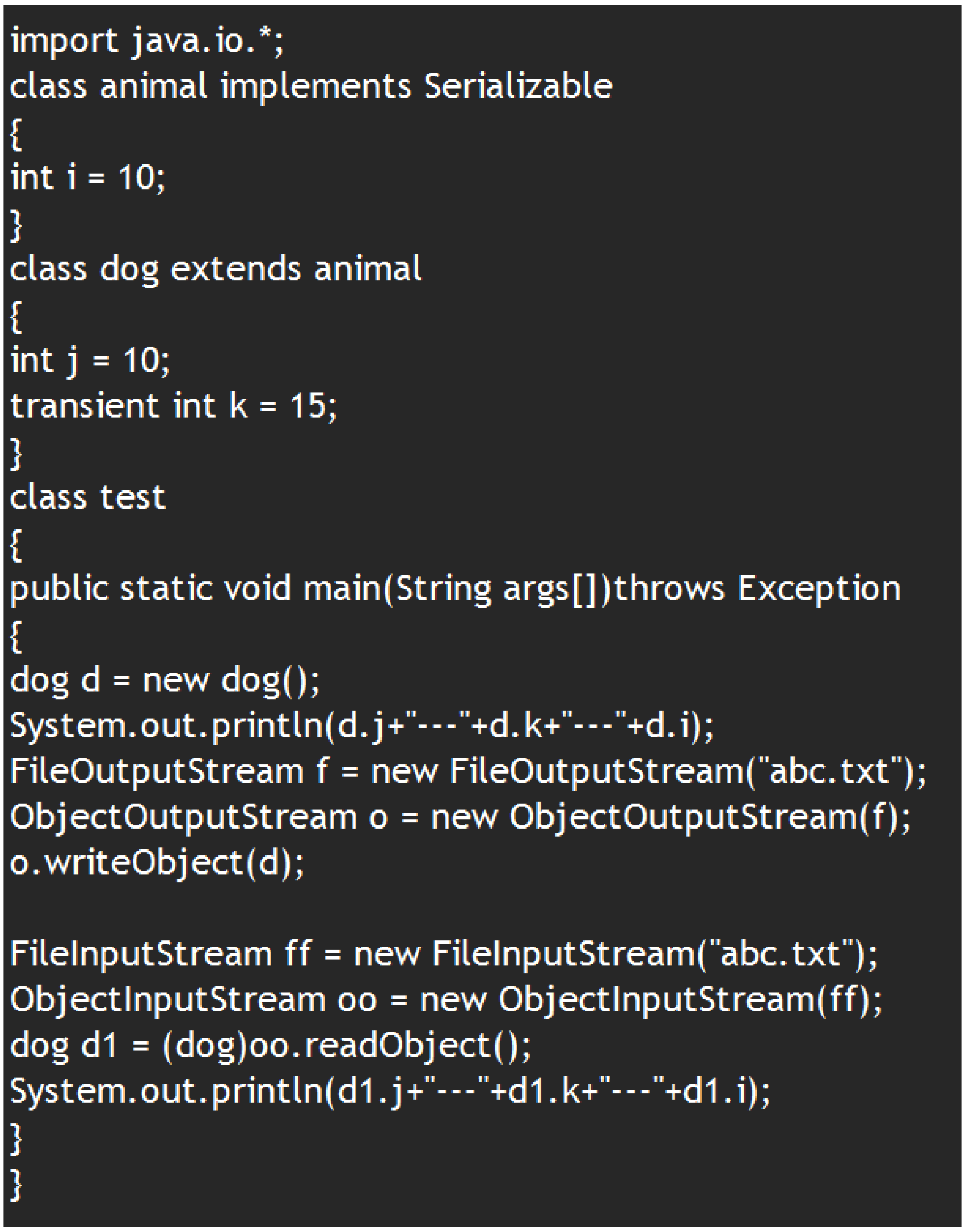


```
C:\Users\Atish kumar sahu\desktop>javac ser.java

C:\Users\Atish kumar sahu\desktop>java test1
ATISH---123456---2001
ATISH---123456---2001
```

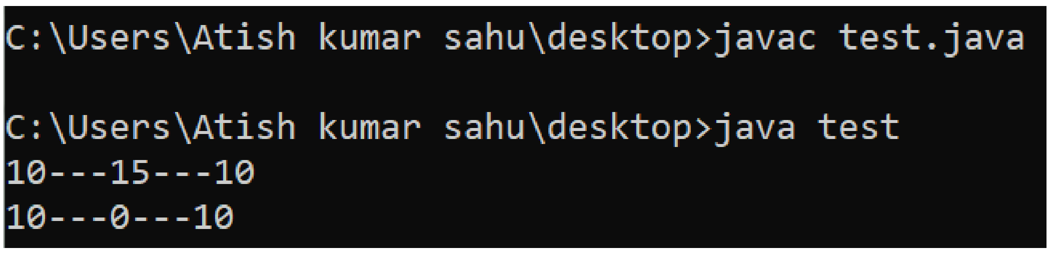
Inheritance In Serialization:

CASE – 1:

even though child class doesn’t implements serializable we can serialize child class object if parent class implements serializable interface. That is serializable nature is inheriting from parent to child. Hence if parent is serializable then by default every child is serializable.



```java
import java.io.*;
class animal implements Serializable
{
int i = 10;
}
class dog extends animal
{
int j = 10;
transient int k = 15;
}
class test
{
public static void main(String args[])throws Exception
{
dog d = new dog();
System.out.println(d.j+"---"+d.k+"---"+d.i);
FileOutputStream f = new FileOutputStream("abc.txt");
ObjectOutputStream o = new ObjectOutputStream(f);
o.writeObject(d);

FileInputStream ff = new FileInputStream("abc.txt");
ObjectInputStream oo = new ObjectInputStream(ff);
dog d1 = (dog)oo.readObject();
System.out.println(d1.j+"---"+d1.k+"---"+d1.i);
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
10---15---10
10---0---10
```

In the above example even though dog class doesn’t implements serializable we can serialize dog object because its parent animal class implements serializable.

Object CLASS DOESN’T IMPLEMENTS SERIALIZABLE INTERFACE.

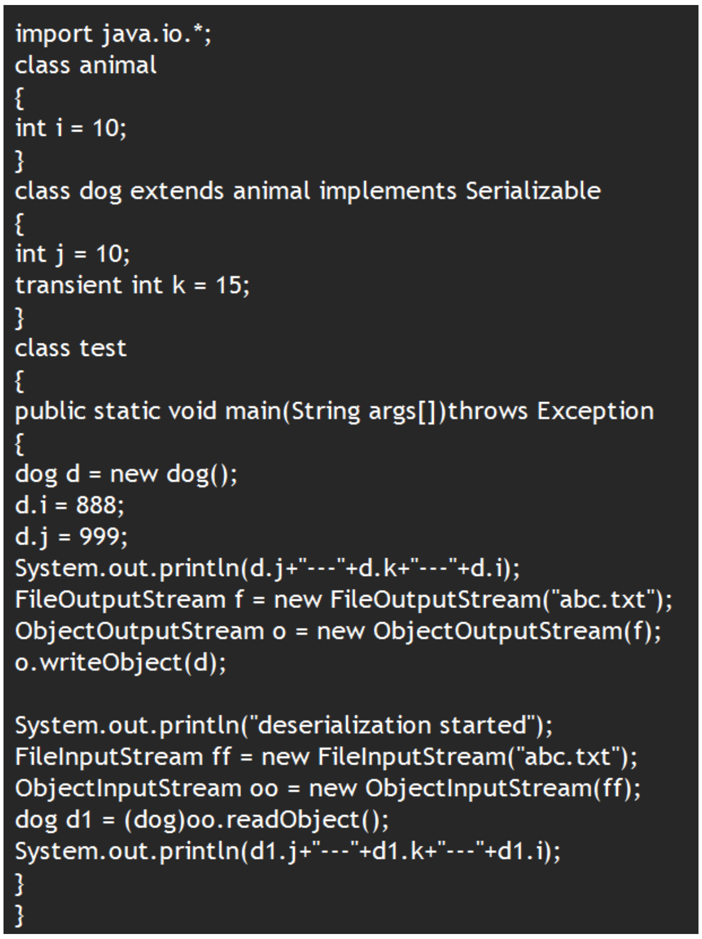
CASE - 2

even though parent class doesn’t implements serializable we can serialize child class object if child class implements serializable interface. That is to serialize child class object parent class need not be serializable.

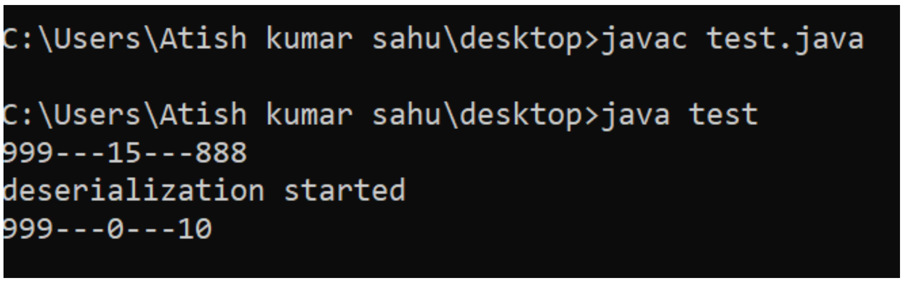
At the time of serialization JVM will check is any variable inheriting from non-serializable parent or not. If any variable inheriting from non-serializable parent then JVM ignores original value and save the default value to the file.

At the time of deserialization JVM will check is any parent class non-serializable or not. If any parent class is non-serializable then JVM will execute instance control flow in every non-serializable parent and share its instance value to the current object.

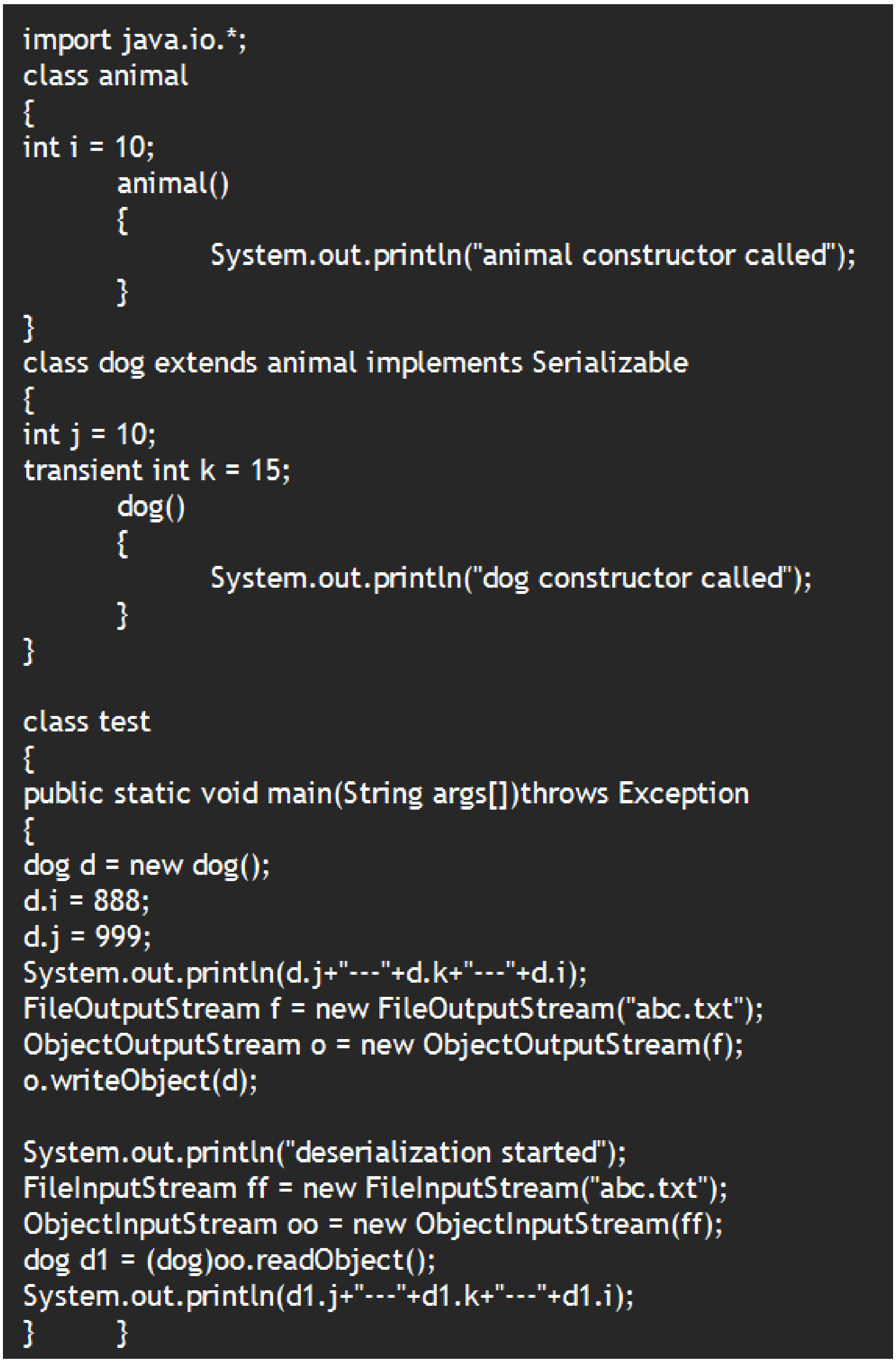
While executing instance control flow of non-serializable parent JVM will always call no argument constructor hence every non-serializable class should compulsory contain no argument constructor. It may be default constructor generated by compiler or customized constructor explicitly provided by programmer. Otherwise we will get runtime exception saying ”***InvalidClassException***”.



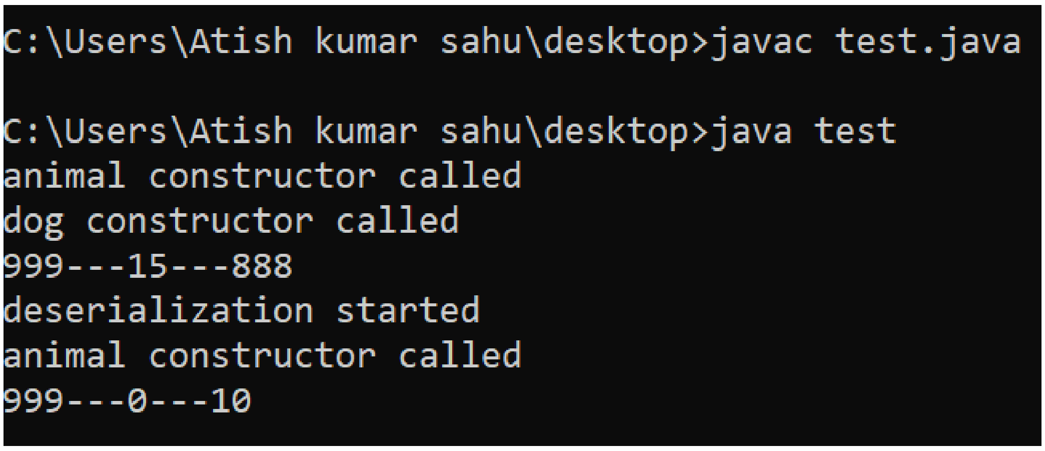
```
import java.io.*;
class animal
{
int i = 10;
}
class dog extends animal implements Serializable
{
int j = 10;
transient int k = 15;
}
class test
{
public static void main(String args[])throws Exception
{
dog d = new dog();
d.i = 888;
d.j = 999;
System.out.println(d.j+"---"+d.k+"---"+d.i);
FileOutputStream f = new FileOutputStream("abc.txt");
ObjectOutputStream o = new ObjectOutputStream(f);
o.writeObject(d);

System.out.println("deserialization started");
FileInputStream ff = new FileInputStream("abc.txt");
ObjectInputStream oo = new ObjectInputStream(ff);
dog d1 = (dog)oo.readObject();
System.out.println(d1.j+"---"+d1.k+"---"+d1.i);
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
999---15---888
deserialization started
999---0---10
```



```
import java.io.*;
class animal
{
int i = 10;
        animal()
        {
                System.out.println("animal constructor called");
        }
}
class dog extends animal implements Serializable
{
int j = 10;
transient int k = 15;
        dog()
        {
                System.out.println("dog constructor called");
        }
}

class test
{
public static void main(String args[])throws Exception
{
dog d = new dog();
d.i = 888;
d.j = 999;
System.out.println(d.j+"---"+d.k+"---"+d.i);
FileOutputStream f = new FileOutputStream("abc.txt");
ObjectOutputStream o = new ObjectOutputStream(f);
o.writeObject(d);

System.out.println("deserialization started");
FileInputStream ff = new FileInputStream("abc.txt");
ObjectInputStream oo = new ObjectInputStream(ff);
dog d1 = (dog)oo.readObject();
System.out.println(d1.j+"---"+d1.k+"---"+d1.i);
}       }
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
animal constructor called
dog constructor called
999---15---888
deserialization started
animal constructor called
999---0---10
```

Externalization: 1.1v

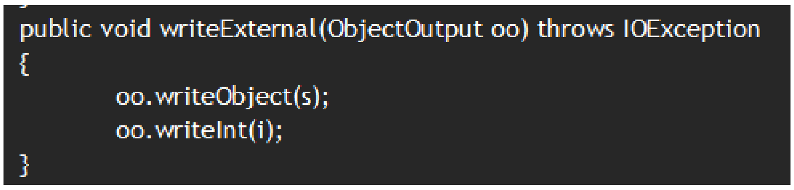
In serialization everything takes care by JVM and programmer doesn't have any control in serialization it is always possible to save total object to the file and it is not possible to save part of the object, which may create performance problems.

To overcome this problem we should go for externalization. The main advantage of externalization over serialization is everything takes care by programmer and JVM doesn't have any control. Based on our requirement we can save either total object or part of the object, which improves performance of the system.

To provide externalizable ability for any java object compulsory the corresponding class should implement externalizable interface. Externalizable interface defines two methods

"***writeExternal***()" "***readExternal***()" Externalizable is the child interface of serializable.

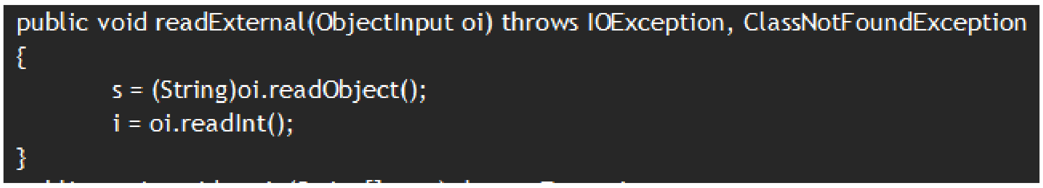
<u>writeExternal:</u>



```
public void writeExternal(ObjectOutput oo) throws IOException
{
        oo.writeObject(s);
        oo.writeInt(i);
}
```

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.

<u>readExternal:</u>



```
public void readExternal(ObjectInput oi) throws IOException, ClassNotFoundException
{
        s = (String)oi.readObject();
        i = oi.readInt();
}
```

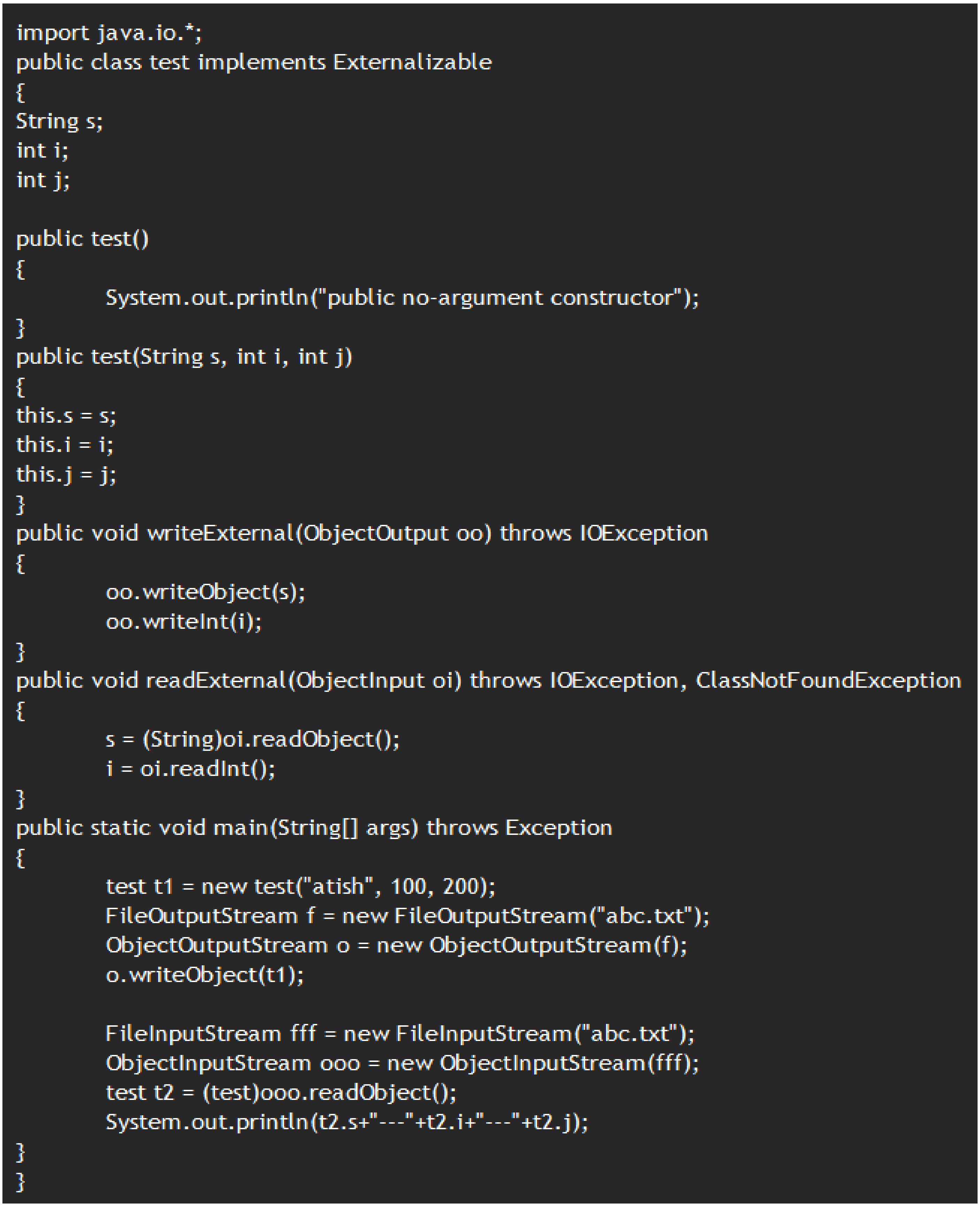
This method will be executed automatically at the time of deserialization with in this method we have to write code to read required variables from the file and assign to current object.

But strictly speaking at the time of deserialization JVM will create a separate new object by executing public no argument constructor. On that object JVM will call read external methods.

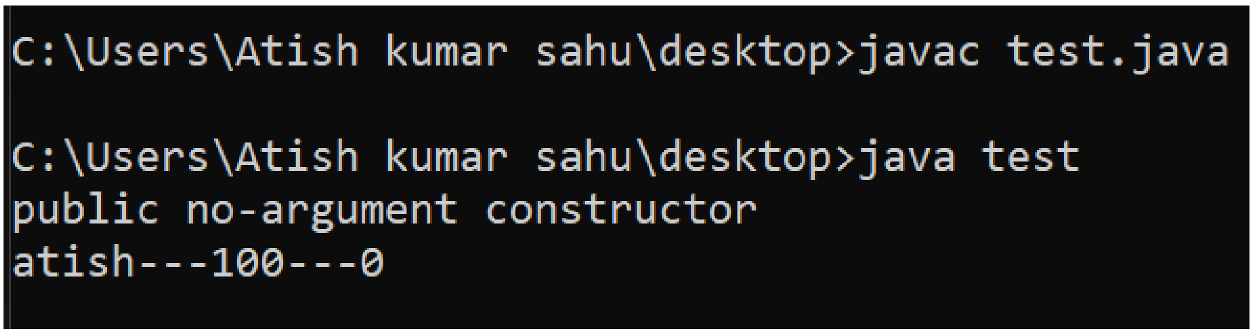
Hence every externalizable implemented class should compulsory contain public no argument constructor otherwise we will get runtime exception saying "InvalidClassException".

If the class implements serializable then total object will be saved to the file in this case output is "atish, 100, 200". If the class implements externalizable then only require variable will be saved to the file in this case output is "public no-argument constructor atish, 100,0".

In serialization transient keyword will play role but in externalization transient keyword won't play any role. Of course transient keyword is not required in externalization.



```
import java.io.*;
public class test implements Externalizable
{
String s;
int i;
int j;

public test()
{
	System.out.println("public no-argument constructor");
}
public test(String s, int i, int j)
{
this.s = s;
this.i = i;
this.j = j;
}
public void writeExternal(ObjectOutput oo) throws IOException
{
	oo.writeObject(s);
	oo.writeInt(i);
}
public void readExternal(ObjectInput oi) throws IOException, ClassNotFoundException
{
	s = (String)oi.readObject();
	i = oi.readInt();
}
public static void main(String[] args) throws Exception
{
	test t1 = new test("atish", 100, 200);
	FileOutputStream f = new FileOutputStream("abc.txt");
	ObjectOutputStream o = new ObjectOutputStream(f);
	o.writeObject(t1);

	FileInputStream fff = new FileInputStream("abc.txt");
	ObjectInputStream ooo = new ObjectInputStream(fff);
	test t2 = (test)ooo.readObject();
	System.out.println(t2.s+"---"+t2.i+"---"+t2.j);
}
}
```



```
C:\Users\Atish kumar sahu\desktop>javac test.java

C:\Users\Atish kumar sahu\desktop>java test
public no-argument constructor
atish---100---0
```

Difference Between Serialization & Externalization:

Serialization:

1. it is meant for default serialization.
2. here everything takes care by JVM and programmer doesn’t have any control.
3. in this case it is always possible to save total object to the file and it is not possible to save part of the object.
4. relatively performance is low.
5. it is the best choice if we want to save total object to the file.
6. serializable interface doesn’t contain any methods. And it is a marker interface.
7. serializable implemented class not required to contain public no-argument constructor.
8. transient keyword will play role in serialization.

Externalization:

1. it is meant for customized serialization.
2. here everything takes care by programmer and JVM doesn’t have any control.
3. based on our requirement we can save either total object or part of the object.
4. relatively performance is high.
5. it is the best choice if we want to save part of the object to the file.
6. externalizable interface contain two methods “writeExternal” and “readExternal”. Hence it is not a marker interface.
7. externalizable implemented class should compulsory contain public no-argument constructor. Otherwise we will get runtime exception saying “InvalidClassException”.
8. transient keyword won’t play any role in externalization. Of course it won’t be required.

SerialVersionUID:

In serialization both sender and receiver need not be same person, need not to use same machine and need not be from same location. The person may be different, the machines may be different and locations may be different. In serialization both sender and receiver should has .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 save a unique identifier. JVM is responsible to generate this unique identifier based on (.class) file at the time of deserialization receiver side JVM will compare unique identifier associated with the 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 s nothing but serial version UID.

Problems On Depending On Default Serial Version UID Generated By JVM:

Both sender and receiver should use same JVM with respect to vendor and platform and version, otherwise receiver unable to deserialize because of different serial version UID.

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 unable to deserialize.

To generate serial version UID internally JVM may use complex algorithm which may create performance problems.

We can solve above problems by configuring our own serial version UID. We can configure our own serial version UID as follows ***Private static final long serialVersionUID = 1L;***

```
import java.io.*;
class test implements serializable
{
private static final long serialVersionUID = 1L;
int i = 20;
int j = 40;
}
import java.io.*;
class sender
{
public static void main(String args[])throws Exception
{
test t1 = new test();
FOS f = new FOS("abc.txt");         //FOS = FileOutputStream
OOS o = new OOS(f); //OOS = ObjectOutputStream
o.writeObject(t1);
}
}
import java.io.*;
class receiver
{
public static void main(String args[]) throws exception
{
FIS ff = new FIS("abc.txt"); //FIS = FileInputStream
OIS oo = new OIS(ff); //OIS = ObjectInputStream
test t2 = (test)oo.readObject();
S.O.P(t2.i+"---"+"t2.j");
}
}
javac test.java
javac sender.java
javac receiver.java

java test
java sender
//add new variable at test class and save
javac test.java
java receiver
```

In the above program after serialization if we perform any change to the(.class) file yet receiver side we won’t get any problem at the time of deserialization. In this case sender and receiver not required to maintain same JVM version.

SOME IDE PROMPT PROGRAMMER TO ENTER SERIAL VERSION UID EXPLICITLY. SOME IDE MAY GENERATE SERIAL VERSION UID AUTOMATICALLY.