structure after Serialization has been done, you will not be able to deserialize the object, see the example below:

Let's change the Employee class in our previous example and remove the transient keyword from salary variable:

Program 1:

```
public class Employee implements Serializable {
    private String name;
    private int age;
    private int salary;

    public Employee(String name, int age, int salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }
    @Override
    public String toString() {
        return "Employee [name=" + name + ", age=" + age
                + ", salary=" + salary + "]";
    }
}
```

We have already serialized the Employee object in our previous example, now let's try to de-serialize it back:

```
public class DeserializationTest {
    public static void main(String[] args) {
        String file = "C:\\temp\\byteStream.txt";
        try {
            FileInputStream fis = new FileInputStream(file);
            ObjectInputStream ois = new ObjectInputStream(fis);
            Employee emp1 = (Employee) ois.readObject();

            fis.close();
            ois.close();
            System.out.println("Employee object is de-serialized : " + emp1);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
java.io.InvalidClassException: com.serialization.demo.Employee; local class incompatible: stream classdesc serialVersionUID = -3697389390179909057, local class serialVersionUID = -3759917827722067163
        at java.io.ObjectStreamClass.initNonProxy(Unknown Source)
        at java.io.ObjectInputStream.readNonProxyDesc(Unknown Source)
        at java.io.ObjectInputStream.readClassDesc(Unknown Source)
        at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.readObject(Unknown Source)
        at com.serialization.demo.DeserializationTest.main(DeserializationTest.java:13)
```

Hence, it is strongly recommended that all serializable classes explicitly declare serialVersionUID value, in case you are using an IDE and not giving any serialVersionUID, compiler will give you a warning like below:

Example with SerialVersionUID:

Program 2:

```
public class Employee implements Serializable {

    private static final long serialVersionUID = 21L;

    private String name;
    private int age;
    private transient int salary;
    public Employee(String name, int age, int salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }
    @Override
    public String toString() {
        return "Employee [name=" + name + ", age=" + age
                + ", salary=" + salary + "]";
    }
}
```

SerializationDemo.java:

```
public class SerializationDemo {
    public static void main(String[] args) {
        Employee emp = new Employee("John", 20, 31000);
        String file = "C:\\temp\\emp.ser";
        try {
            FileOutputStream fos = new FileOutputStream(file);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(emp);

            fos.close();
            oos.close();
            System.out.println("Employee object is serialized : " + emp);
        } catch (IOException e1) {
            System.out.println("IOException is caught");
        }

        try {
            FileInputStream fis = new FileInputStream(file);
            ObjectInputStream ois = new ObjectInputStream(fis);
            Employee emp1 = (Employee) ois.readObject();

            fis.close();
            ois.close();
            System.out.println("Employee object is de-serialized : " + emp1);
        } catch (IOException e) {
            System.out.println("IOException is caught");
        } catch (ClassNotFoundException e) {
            System.out.println("ClassNotFoundException is caught");
        }
    }
}
```

Output:

```
Employee object is serialized : Employee [name=John, age=20, salary=31000]
Employee object is de-serialized : Employee [name=John, age=20, salary=0]
```

Now, let's add one more field 'company' and remove the transient keyword from our Employee class. Here, as we are changing the class structure, let's see if we get the error of *InvalidClassException* again:

Program 3:

```
public class Employee implements Serializable {

    private static final long serialVersionUID = 21L;

    private String name;
    private int age;
    private int salary;
    private String companyName;
    public Employee(String name, int age, int salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }
    @Override
    public String toString() {
        return "Employee [name=" + name + ", age=" + age
                + ", salary=" + salary + ", companyName=" + companyName + "]";
    }
}
```

DeserializationTest.java:

```
public class DeserializationTest {
    public static void main(String[] args) {
        String file = "C:\\temp\\emp.ser";
        try {
            FileInputStream fis = new FileInputStream(file);
            ObjectInputStream ois = new ObjectInputStream(fis);
            Employee emp1 = (Employee) ois.readObject();

            fis.close();
            ois.close();
            System.out.println("Employee object is de-serialized : " + emp1);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
Employee object is de-serialized : Employee [name=John, age=20, salary=0, companyName=null]
```

Some Points to remember:

- If a parent class has implemented Serializable interface then child class doesn't need to implement it but the reverse is not true
- Static data members and transient data members are not saved via Serialization process (serialVersionUID is an exception). So, if you don't want to save value of a non-static data member then make it transient
- Constructor of serialized class is never called when the serialized object is deserialized (in case of inheritance, no-arg constructor of parent gets called during de-serialization)

Question 58: Serialization scenarios with Inheritance

Case 1: If super class is Serializable then by default, its sub-classes are also Serializable

```
class Parent implements Serializable {
    int x;
    public Parent(int x) {
        this.x = x;
    }
}

class Child extends Parent {
    int y;
    public Child(int x, int y) {
        super(x);
        this.y = y;
    }
}
public class TestSerialization {
    public static void main(String[] args) {
        Child child = new Child(10,50);
        System.out.println("x : " + child.x);
        System.out.println("y : " + child.y);
        String file = "C:\\temp\\child.ser";

        serializeObject(file, child);
        deserializeObject(file);
    }
    private static void serializeObject(String file, Child child) {
        try {
            FileOutputStream fos = new FileOutputStream(file);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(child);
            fos.close();
            oos.close();
            System.out.println("The object has been serialized");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    private static void deserializeObject(String file) {
        try {
            FileInputStream fis = new FileInputStream(file);
            ObjectInputStream ois = new ObjectInputStream(fis);
            Child child1 = (Child) ois.readObject();
            fis.close();
            ois.close();
            System.out.println("The object has been deserialized");
            System.out.println("x : " + child1.x);
            System.out.println("y : " + child1.y);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
x : 10
y : 50
The object has been serialized
The object has been deserialized
x : 10
y : 50
```

Case 2: When super class does not implement the Serializable Interface, then also we can serialize the subclass provided that it implements Serializable interface.

In this case, when we de-serialize the subclass object, then no-arg constructor of its parent class gets called. So, the serializable sub-class must have access to the default no-arg constructor of its parent class (general rule is that the Serializable sub-class must have access to the no-arg constructor of first non-Serializable super class).

```java
class Parent {
    int x;
    public Parent(int x) {
        this.x = x;
    }
}

class Child extends Parent implements Serializable {
    int y;
    public Child(int x, int y) {
        super(x);
        this.y = y;
    }
}
public class TestSerialization {
    public static void main(String[] args) {
        Child child = new Child(20,40);
        System.out.println("x : " + child.x);
        System.out.println("y : " + child.y);
        String file = "C:\\temp\\child1.ser";

        serializeObject(file, child);
        deserializeObject(file);
    }
```

(Note: serializeObject() and deserializeObject() remains same as the Case 1 program)

Output:

```
x : 20
y : 40
The object has been serialized
java.io.InvalidClassException: com.serialization.demo.Child; no valid constructor
        at java.io.ObjectStreamClass$ExceptionInfo.newInvalidClassException(Unknown Source)
        at java.io.ObjectStreamClass.checkDeserialize(Unknown Source)
        at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.readObject(Unknown Source)
        at com.serialization.demo.TestSerialization.deserializeObject(TestSerialization.java:54)
        at com.serialization.demo.TestSerialization.main(TestSerialization.java:33)
```

When no-arg constructor is present in Super class:

```
class Parent {
    int x;
    public Parent(int x) {
        this.x = x;
        System.out.println("Parent class one-arg constructor");
    }
    public Parent() {
        x = 100;
        System.out.println("Parent class no-arg constructor");
    }
}
```

```
class Child extends Parent implements Serializable {
    int y;
    public Child(int x, int y) {
        super(x);
        this.y = y;
        System.out.println("Child class two-arg constructor");
    }
    public Child() {
        System.out.println("Child class no-arg constructor");
    }
}
public class TestSerialization {
    public static void main(String[] args) {
        Child child = new Child(20,40);
        System.out.println("x : " + child.x);
        System.out.println("y : " + child.y);
        String file = "C:\\temp\\child2.ser";

        serializeObject(file, child);
        deserializeObject(file);
    }
```

(Note: serializeObject() and deserializeObject() remains same as the Case 1 program)

Output:

```
Parent class one-arg constructor
Child class two-arg constructor
x : 20
y : 40
The object has been serialized
Parent class no-arg constructor
The object has been deserialized
x : 100
y : 40
```

Question 59: Stopping Serialization and De-serialization

Suppose, parent class implements Serializable interface but we don't need the child class to be serialized

Here, we can implement writeObject() and readObject() methods in sub-class and throw NotSerializableException from these methods :

```java
class Parent implements Serializable {
    int x;
    public Parent(int x) {
        this.x = x;
    }
}

class Child extends Parent {
    int y;
    public Child(int x, int y) {
        super(x);
        this.y = y;
    }
    private void writeObject(ObjectOutputStream oos) throws IOException {
      throw new NotSerializableException("Serialization is not supported on this object");
    }
    private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {
      throw new NotSerializableException("Serialization is not supported on this object");
    }
}

public class TestSerialization {
    public static void main(String[] args) {
        Child child = new Child(5,25);
        System.out.println("x : " + child.x);
        System.out.println("y : " + child.y);
        String file = "C:\\temp\\child3.ser";

        serializeObject(file, child);
        deserializeObject(file);
    }
```

(Note: serializeObject() and deserializeObject() remains same as the Question 58 - Case 1 program)

Output:

```
x : 5
y : 25
java.io.NotSerializableException: Serialization is not supported on this object
        at com.serialization.demo.Child.writeObject(TestSerialization.java:25)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
        at java.lang.reflect.Method.invoke(Unknown Source)
        at java.io.ObjectStreamClass.invokeWriteObject(Unknown Source)
        at java.io.ObjectOutputStream.writeSerialData(Unknown Source)
        at java.io.ObjectOutputStream.writeOrdinaryObject(Unknown Source)
        at java.io.ObjectOutputStream.writeObject0(Unknown Source)
        at java.io.ObjectOutputStream.writeObject(Unknown Source)
        at com.serialization.demo.TestSerialization.serializeObject(TestSerialization.java:47)
        at com.serialization.demo.TestSerialization.main(TestSerialization.java:39)

java.io.NotSerializableException: Serialization is not supported on this object
        at com.serialization.demo.Child.readObject(TestSerialization.java:28)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
        at java.lang.reflect.Method.invoke(Unknown Source)
        at java.io.ObjectStreamClass.invokeReadObject(Unknown Source)
        at java.io.ObjectInputStream.readSerialData(Unknown Source)
        at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.readObject(Unknown Source)
        at com.serialization.demo.TestSerialization.deserializeObject(TestSerialization.java:61)
        at com.serialization.demo.TestSerialization.main(TestSerialization.java:40)
```

Serialization and De-serialization process can be customized also by providing writeObject() and readObject() methods in the class that we want to serialize.

```
private void writeObject(ObjectOutputStream oos) throws IOException {
  // Any Custom logic
 oos.defaultWriteObject();
  // Any Custom logic
}
private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {
 // Any Custom logic
 ois.defaultReadObject();
  // Any Custom logic
}
```

Declaring both methods as private is necessary (public methods will not work), so other than the JVM nothing else can see them. This also proves that neither method is not inherited nor overridden or overloaded. The JVM automatically checks these methods and calls them during the serialization-deserialization process. The JVM can call these private methods, but other objects cannot. Thus, the integrity of the class is maintained and the serialization protocol can continue to work as normal.

For example, you can have encryption and decryption logic in these methods.

Question 60: What is Externalizable Interface?

Answer: The default serialization process is very slow as it is fully recursive, so whenever we try to serialize one object, the serialization process tries to serialize all the fields of our class (except static and transient variables). So, if we have a class with lots of variables present and we do not want to serialize all of them, we have to make all of those variables as transient, all these fields will always be assigned with default values. This makes the entire process very slow.

Externalizable interface is used when we want to implement custom logic to serialize/deserialize an object. Externalizable interface extends the Serializable interface, and it has two methods, writeExternal() and readExternal() which are used for serialization and de-serialization. This way, we can change the JVM's default serialization behavior because while using Externalizable, we decide what to store in stream.

Program 1:

Employee.java:

```java
package com.externalizable.demo;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

public class Employee implements Externalizable {
    private String name;
    private int age;
    private transient int salary;
    private String companyName;
    public Employee() {
        System.out.println("No-arg constructor of Employee class");
    }

    public Employee(String name, int age, int salary, String companyName) {
        this.name = name;
        this.age = age;
        this.salary = salary;
        this.companyName = companyName;
    }
    @Override
    public String toString() {
        return "Employee [name=" + name + ", age=" + age + ", salary=" +
                    salary + ", companyName=" + companyName + "]";
    }
    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeObject(name);
        out.writeInt(age);
        out.writeInt(salary);
    }
    @Override
    public void readExternal(ObjectInput in) throws IOException,
                                        ClassNotFoundException {
        name = (String) in.readObject();
        age = in.readInt();
        salary = in.readInt();
    }
}
```

TestExternalizable.java:

```java
public class TestExternalizable {

    public static void main(String[] args) {
        Employee emp = new Employee("Mike", 15, 25000, "ABC");
        String file = "C:\\temp\\object.ser";
        try {
            FileOutputStream fos = new FileOutputStream(file);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(emp);

            fos.close();
            oos.close();
            System.out.println("Employee object is serialized : \n" + emp);
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
        try {
            FileInputStream fis = new FileInputStream(file);
            ObjectInputStream ois = new ObjectInputStream(fis);
            Employee emp1 = (Employee) ois.readObject();

            fis.close();
            ois.close();
            System.out.println("Employee object is de-serialized : \n" + emp1);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

}
```

Output:

```
Employee object is serialized :
Employee [name=Mike, age=15, salary=25000, companyName=ABC]
No-arg constructor of Employee class
Employee object is de-serialized :
Employee [name=Mike, age=15, salary=25000, companyName=null]
```

Now, one thing to remember here is that the public no-arg constructor gets called before readExternal() method, so we have to provide this no-arg constructor or else we will get an exception during run-time.

Comment the public no-arg constructor from Employee.java:

Program 2:

```
public class Employee implements Externalizable {
    private String name;
    private int age;
    private transient int salary;
    private String companyName;
//  public Employee() {
//      System.out.println("No-arg constructor of Employee class");
//  }
```

Now, run TestExternalizable.java, you will get below output:

```
Employee object is serialized :
Employee [name=Mike, age=15, salary=25000, companyName=ABC]
java.io.InvalidClassException: com.externalizable.demo.Employee; no valid constructor
        at java.io.ObjectStreamClass$ExceptionInfo.newInvalidClassException(Unknown Source)
        at java.io.ObjectStreamClass.checkDeserialize(Unknown Source)
        at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.readObject(Unknown Source)
        at com.externalizable.demo.TestExternalizable.main(TestExternalizable.java:28)
```

So, if interviewer asks a question that you have a class which has 1000 variables and you want to serialize only 10 specific variables, your answer should be using an Externalizable interface.

Some points to remember:

- Using Externalizable interface, we can implement custom logic for serialization and deserialization of object
- When using Externalizable, we have to explicitly mention what fields or variables we want to serialize
- When using Externalizable, a public no-arg constructor is required
- Using Externalizable, we can also serialize transient and static variables
- readExternal() method must read the values in the same sequence and with the same types as were written by writeExternal() method

Question 61: Externalizable with Inheritance

If a class is implementing Externalizable and also has a parent class, how we can save the data of parent class and how we can recover it back. In case of Externalizable, we have to implement the readExternal() and writeExternal() methods in each and every class we want to serialize or deserialize.

Case 1: When both parent and child classes are implementing Externalizable interfaces:

Department.java:

```
package com.externalizable.demo;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

public class Department implements Externalizable {
    private int deptId;
    private String deptName;
    private int capacity;

    public Department() {
        System.out.println("No-arg constructor of Department class");
    }
    public void setDeptId(int deptId) { this.deptId = deptId; }
    public void setDeptName(String deptName) { this.deptName = deptName; }
    public void setCapacity(int capacity) { this.capacity = capacity; }

    public int getDeptId() { return deptId; }
    public String getDeptName() { return deptName; }
    public int getCapacity() { return capacity; }

    @Override
    public String toString() {
        return "Department [deptId=" + deptId + ", deptName=" + deptName
                + ", capacity=" + capacity + "]";
    }
    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeInt(deptId);
        out.writeUTF(deptName);
    }
    @Override
    public void readExternal(ObjectInput in) throws IOException,
                                ClassNotFoundException {
        deptId = in.readInt();
        deptName = in.readUTF();
    }
}
```

Student.java:

```java
package com.externalizable.demo;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

public class Student extends Department implements Externalizable {

    private int studentId;
    private String studentName;
    private int age;

    public Student() {
        System.out.println("No-arg constructor of Student class");
    }

    public void setStudentId(int studentId) { this.studentId = studentId; }
    public void setStudentName(String studentName) { this.studentName = studentName; }
    public void setAge(int age) { this.age = age; }

    public int getStudentId() { return studentId; }
    public String getStudentName() { return studentName; }
    public int getAge() { return age; }
    @Override
    public String toString() {
        return super.toString() + "\n" +
                "Student [studentId=" + studentId + ", studentName=" + studentName
                        + ", age=" + age + "]";
    }
    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        super.writeExternal(out);
        out.writeInt(studentId);
        out.writeUTF(studentName);
    }
    @Override
    public void readExternal(ObjectInput in) throws IOException,
                                        ClassNotFoundException {
        super.readExternal(in);
        studentId = in.readInt();
        studentName = in.readUTF();
    }

}
```

Here, in the overridden writeExternal() and readExternal() methods in child class Student, we are also calling super.writeExternal() and super.readExternal() methods to serialize and de-serialize fields of parent class Department.

ExternalizableDemo.java:

```java
public class ExternalizableDemo {
    public static void main(String[] args) {
        Student st = new Student();
        st.setStudentId(1);
        st.setStudentName("Lisa");
        st.setAge(20);
        st.setDeptId(10);
        st.setDeptName("IT");
        st.setCapacity(60);
        String file = "C:\\temp\\object1.ser";
```

```
        try {
            FileOutputStream fos = new FileOutputStream(file);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(st);
            fos.close();
            oos.close();
            System.out.println("Student object is serialized : \n" + st);
        } catch (IOException e) {
            e.printStackTrace();
        }

        try {
            FileInputStream fis = new FileInputStream(file);
            ObjectInputStream ois = new ObjectInputStream(fis);
            Student st1 = (Student) ois.readObject();
            fis.close();
            ois.close();
            System.out.println("Student object is de-serialized : \n" + st1);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
No-arg constructor of Department class
No-arg constructor of Student class
Student object is serialized :
Department [deptId=10, deptName=IT, capacity=60]
Student [studentId=1, studentName=Lisa, age=20]
No-arg constructor of Department class
No-arg constructor of Student class
Student object is de-serialized :
Department [deptId=10, deptName=IT, capacity=0]
Student [studentId=1, studentName=Lisa, age=0]
```

The capacity and age variable values are 0 because we did not serialize these two variables.

Case 2: When only child class is implementing the Externalizable interface

Department.java:

```
package com.externalizable.demo;

public class Department {
    private int deptId;
    private String deptName;
    private int capacity;

    public Department() {
        System.out.println("No-arg constructor of Department class");
    }
```

```
    public void setDeptId(int deptId) { this.deptId = deptId; }
    public void setDeptName(String deptName) { this.deptName = deptName; }
    public void setCapacity(int capacity) { this.capacity = capacity; }

    public int getDeptId() { return deptId; }
    public String getDeptName() { return deptName; }
    public int getCapacity() { return capacity; }

    @Override
    public String toString() {
        return "Department [deptId=" + deptId + ", deptName=" + deptName
                + ", capacity=" + capacity + "]";
    }
}
```

Student.java:

```
package com.externalizable.demo;

import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

public class Student extends Department implements Externalizable {
    private int studentId;
    private String studentName;
    private int age;
    public Student() {
        System.out.println("No-arg constructor of Student class");
    }
    public void setStudentId(int studentId) { this.studentId = studentId; }
    public void setStudentName(String studentName) { this.studentName = studentName; }
    public void setAge(int age) { this.age = age; }

    public int getStudentId() { return studentId; }
    public String getStudentName() { return studentName; }
    public int getAge() { return age; }

    @Override
    public String toString() {
        return super.toString() + "\n" +
                "Student [studentId=" + studentId + ", studentName=" + studentName
                        + ", age=" + age + "]";
    }

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeInt(getDeptId());
        out.writeUTF(getDeptName());
        out.writeInt(studentId);
        out.writeUTF(studentName);
    }
    @Override
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
        setDeptId(in.readInt());
        setDeptName(in.readUTF());
        studentId = in.readInt();
        studentName = in.readUTF();
    }
}
```

Here, we are using getters and setters of parent class, Department, to serialize and de-serialize its fields.

ExternalizableDemo.java:

```java
public class ExternalizableDemo {
    public static void main(String[] args) {
        Student st = new Student();
        st.setStudentId(1);
        st.setStudentName("Lisa");
        st.setAge(20);
        st.setDeptId(10);
        st.setDeptName("IT");
        st.setCapacity(60);
        String file = "C:\\temp\\object2.ser";

        try {
            FileOutputStream fos = new FileOutputStream(file);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(st);
            fos.close();
            oos.close();
            System.out.println("Student object is serialized : \n" + st);
        } catch (IOException e) {
            e.printStackTrace();
        }

        try {
            FileInputStream fis = new FileInputStream(file);
            ObjectInputStream ois = new ObjectInputStream(fis);
            Student st1 = (Student) ois.readObject();
            fis.close();
            ois.close();
            System.out.println("Student object is de-serialized : \n" + st1);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
No-arg constructor of Department class
No-arg constructor of Student class
Student object is serialized :
Department [deptId=10, deptName=IT, capacity=60]
Student [studentId=1, studentName=Lisa, age=20]
No-arg constructor of Department class
No-arg constructor of Student class
Student object is de-serialized :
Department [deptId=10, deptName=IT, capacity=0]
Student [studentId=1, studentName=Lisa, age=0]
```

Question 62: Difference between Serializable and Externalizable

Answer:

- Serializable is a marker interface which means it does not contain any method whereas Externalizable is a child interface of Serializable and it contains two methods writeExternal() and readExternal()
- When using Serializable, JVM takes full responsibility for serializing the class object but in case of Externalizable, the programmer has full control over serialization logic
- Serializable interface is a better fit when we want to serialize the entire object whereas Externalizable is better suited for custom serialization
- Default serialization is easy to implement but it comes with some issues and performance cost whereas in case of Externalizable, the programmer has to provide the complete serialization logic which is a little hard but results in better performance

- Default serialization does not call any constructor whereas a public no-arg constructor is needed when using Externalizable interface
- When a class implements Serializable interface, it gets tied with default serialization which can easily break if structure of the class changes like adding/removing any variable whereas using Externalizable, you can create your own binary format for your object

Question 63: How to make a class Immutable?

Answer: As we know, String is an Immutable class in Java, i.e. once initialized its value never change. We can also make our own custom Immutable class, where the class object's state will not change once it is initialized.

Benefits of Immutable class:

- Thread-safe: With immutable classes, we don't have to worry about the thread-safety in case of multi-threaded environment as these classes are inherently thread-safe
- Cacheable: An immutable class is good for Caching because while we don't have to worry about the value changes

How to create an Immutable class in java:

- Declare the class as final so that it cannot be extended
- Make all fields as private so that direct access to them is not allowed
- Make all fields as final so that its value can be assigned only once
- Don't provide 'setter' methods for variables
- When the class contains a mutable object reference,
    1. While initializing the field in constructor, perform a deep copy
    2. While returning the object from its getter method, make sure to return a copy rather than the actual object reference

Example:

We will make Employee class as immutable, but Employee class contains a reference of Address class

Address.java:

```
package com.immutable.demo;

public class Address {
    private String city;
    private String state;

    public Address(String city, String state) {
        this.city = city;
        this.state = state;
    }
    public String getCity() { return city; }
    public String getState() { return state; }
    public void setCity(String city) { this.city = city; }
    public void setState(String state) { this.state = state; }

    @Override
    public String toString() {
        return "Address [city=" + city + ", state=" + state + "]";
    }
}
```

Employee.java:

```java
package com.immutable.demo;

public final class Employee {
    private final String name;
    private final int age;
    private final Address address;

    public Employee(String name, int age, Address address) {
        this.name = name;
        this.age = age;
        Address cloneAddress = new Address(address.getCity(), address.getState());
        this.address = cloneAddress;
    }

    public String getName() { return name; }
    public int getAge() { return age; }
    public Address getAddress() {
        return new Address(address.getCity(), address.getState());
    }

    @Override
    public String toString() {
        return "Employee [name=" + name + ", age=" + age
                    + ", address=" + address + "]";
    }
}
```

TestImmutable.java:

```java
package com.immutable.demo;

public class TestImmutable {
    public static void main(String[] args) {
        Address address = new Address("Chennai", "Tamil Nadu");
        Employee employee = new Employee("Mike", 15, address);

        System.out.println("Original Employee object : \n" + employee);

        address.setCity("Mumbai");
        address.setState("Maharashtra");
        System.out.println("Employee object after local variable address change :\n" + employee);

        Address empAddress = employee.getAddress();
        empAddress.setCity("Jaipur");
        empAddress.setState("Rajasthan");
        System.out.println("Employee object after employee address change:\n" + employee);
    }
}
```

Here, after creating Employee object, the first change is done in local address object and then we used the employee's getter method to access the address object and tried to change the value in it.

Output:

```
Original Employee object :
Employee [name=Mike, age=15, address=Address [city=Chennai, state=Tamil Nadu]]
Employee object after local variable address change :
Employee [name=Mike, age=15, address=Address [city=Chennai, state=Tamil Nadu]]
Employee object after employee address change:
Employee [name=Mike, age=15, address=Address [city=Chennai, state=Tamil Nadu]]
```

As, you can see that the value remained the same.

If we don't follow the rule about mutable object reference present in the class, let's see what will happen in that case.

Let's change the Employee class constructor and getter method:

Employee.java:

```java
package com.immutable.demo;

public final class Employee {
    private final String name;
    private final int age;
    private final Address address;

    public Employee(String name, int age, Address address) {
        this.name = name;
        this.age = age;
        this.address = address;
    }
    public String getName() { return name; }
    public int getAge() { return age; }
    public Address getAddress() {
        return address;
    }
    @Override
    public String toString() {
        return "Employee [name=" + name + ", age=" + age
                + ", address=" + address + "]";
    }
}
```

Now, if we run our *TestImmutable.java* class, below is the output:

```
Original Employee object :
Employee [name=Mike, age=15, address=Address [city=Chennai, state=Tamil Nadu]]
Employee object after local variable address change :
Employee [name=Mike, age=15, address=Address [city=Mumbai, state=Maharashtra]]
Employee object after employee address change:
Employee [name=Mike, age=15, address=Address [city=Jaipur, state=Rajasthan]]
```

Why we perform deep copy in constructor:

- When you assign the actual address object in the constructor, then remember it is storing the reference of address object, so if you change the value in this address object, it will reflect in the employee object

Why we don't return original reference from the getter:

- When you return the original address object from the getter method then you can use the returned object reference to change the values in employee object

Question 64: Explain Class loaders in Java

Answer: **ClassLoader** is a java class which is used to load *.class* files in memory. When we compile a java class, JVM creates a bytecode which is platform independent. The bytecode is present in .class file. When we try to use a class, then classloader loads it into memory.

There are 3 types of built-in class loaders in java:

1. *Bootstrap class loader*: it loads JDK class files from jre/lib/rt.jar and other core classes. It is the parent of all class loaders, it is also called Primordial classloader.
2. *Extensions class loader*: it loads classes from JDK extensions directory, it delegates class loading request to its parent, Bootstrap and if the loading of class is unsuccessful, it loads classes from jre/lib/ext directory or any other directory pointed by *java.ext.dirs* system property.
3. *System class loader*: It loads application specific classes from the CLASSPATH. We can set classpath while invoking the program using -cp or classpath command line options. It is a child of Extension ClassLoader.

Java class loader is based on three principles:

1. Delegation principle: It forwards the request for class loading to its parent class loader. It only loads the class if the parent does not find or load the class.
2. Visibility principle: According to Visibility principle, the child ClassLoader can see all the classes loaded by parent ClassLoader. But the parent class loader cannot see classes loaded by the child class loader.
3. Uniqueness principle: According to this principle, a class loaded by Parent should not be loaded by Child ClassLoader again. It is achieved by delegation principle.

Suppose, you have created a class Employee.java and compiled this class and Emloyee.class file is created. Now, you want to use this class, the first request to load this class will come to System/ Application ClassLoader, which will delegate the request to its parent, Extension ClassLoader which further delegates to Primordial or Bootstrap class loader

Now, Bootstrap ClassLoader will look for this class in rt.jar, since this class is not there, the request will come to Extension ClassLoader which looks in jre/lib/ext directory and tries to locate this class there, if this class is found there then Extension ClassLoader will load this class and Application ClassLoader will not load this class, this has been done to maintain the Uniqueness principle. But if the class is not loaded by Extension ClassLoader, then this Employee.class will be loaded by Application ClassLoader from the CLASSPATH.

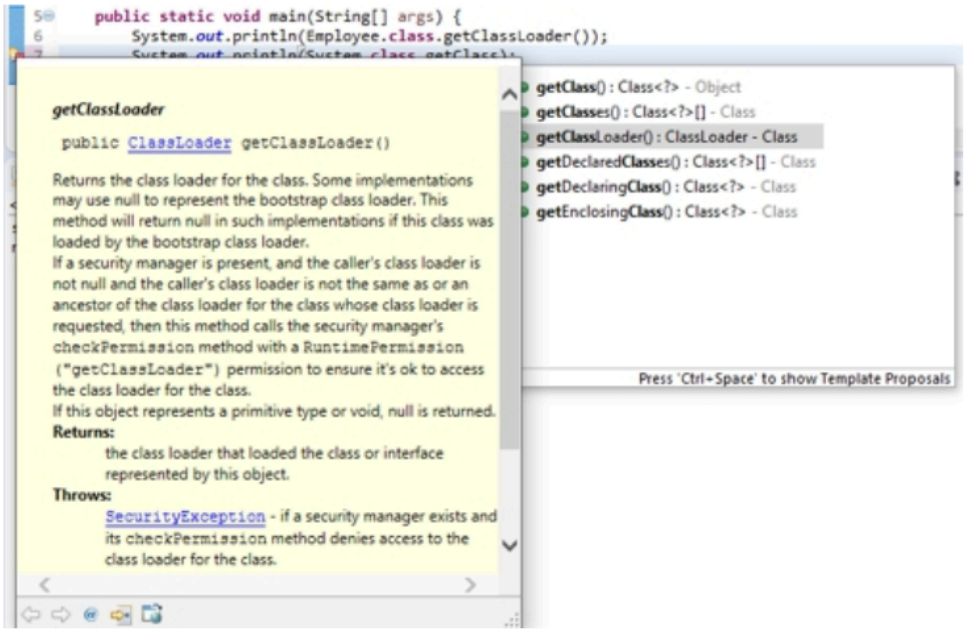
Employee.java:

```
package com.classloader.demo;

public class Employee {

    public static void main(String[] args) {
        System.out.println(Employee.class.getClassLoader());
        System.out.println(System.class.getClassLoader());
    }

}
```

Output:

```
sun.misc.Launcher$AppClassLoader@73d16e93
null
```

If you are thinking why null is printed when we tried to know which classloader is loading the java.lang.System class then take a look at the Javadoc :



We can also create our own custom class loader by extending the ClassLoader class.

Question 65: What is Singleton Design Pattern and how it can be implemented?

Answer: This is a famous interview question, as it involves many follow-up questions as well. I will cover all of them here:

What is Singleton Design Pattern?

Singleton design pattern comes under Creational Design Patterns category and this pattern ensures that only one instance of class exists in the JVM.

Singleton pattern is used in:

- logging, caching, thread pool etc.
- other design patterns like Builder, Prototype, Abstract Factory etc.
- core java classes like java.lang.Runtime etc.

How to implement Singleton Pattern

To implement a Singleton pattern, we have different approaches but all of them have the below common concepts:

- private constructor to restrict instantiation of the class from other classes.
- private static variable of the same class that is the only instance of the class.
- public static method that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.

The approaches to implement singleton pattern are:

Eager Initialization: In eager initialization, the instance of Singleton Class is created at the time of class loading, this is the easiest method to create a singleton class but it has a drawback that instance is created even though client application might not even use it.

Example:

A.java: