**Serialization**

1. **What is the serialization?**

Serialization is a process converting an object into a byte array. This byte array represents the class, version and internal state of the object. JVM can use this byte array to transmit/read the object over a network.

1. **What is the purpose of serialization?**

Some of the uses of serialization are:

1. Communication: It is used for transmitting an object over network between two machines.
2. Persistence: We can store the object’s state in a database and retrieve it from database later on.
3. Caching: Serialization can be used for caching to improve performance. We may need 10 minutes to build an object, but it may take just 10 seconds to de-serialize the object.
4. Cross JVM Synchronization: It can be used in same way across multiple JVM that follow different architecture.
5. **What is Deserialization?**

Deserialization is the process of reconstructing the object from the serialized state. It is the reverse process of serialization.

1. **What is Serialization and Deserialization conceptually?**

Serialization is to convert Object data into a stream of bytes

Deserialization is to convert a stream of bytes back into a copy of the original object.

1. **Why do we mark a data member transient?**

Member variables of an object are marked transient to indicate that they should not be serialized.

During serialization process the transient variables are not considered part of the persistent state of an object.

1. **Is it allowed to mark a method as transient?**

No, Java does not allow marking a method as transient. The transient keyword is valid only for member variables.

1. **How does marking a field as transient makes it possible to serialize an object?**

Let say we have a class ABC that implements Serializable interface, but it contains a member variable object of class XYZ that does not implement Serializable interface. Due to this it is not possible to Serialize the class ABC.

To solve this issue, we can mark the member variable XYZ as Transient in class ABC. This will allow us to serialize the class ABC.

1. **What is Externalizable interface in Java?**

Externalizable interface extends Serializable interface in Java. It is used for giving the Class control over saving and restoring the contents of its instances.

A class implements methods writeExternal() and readExternal() to store and restore the object.

1. **How do we Serialize object, write a program to Serialize and DeSerialize object and persist it in file ?**

You must be able to write Serialization code to impress interviewer. In order to serialize object our class needs to implement java.io.Serializable interface. Serializable interface is Marker interface i.e. it does not have any methods of its own, but it tells Jvm that object has to converted into byte stream.

**SERIALIZATION**

Create object of ObjectOutput and give it’s reference variable name oout and call writeObject() method and pass our employee object as parameter [**oout.writeObject(object1) ]**

|  |
| --- |
| OutputStream fout = **new** FileOutputStream("ser.txt");  ObjectOutput oout = **new** ObjectOutputStream(fout);  System.*out*.println("Serialization process has started, serializing employee objects...");  **oout.writeObject(object1);** |

**DESERIALIZATION :**

Create object of ObjectInput and give it’s reference variable name oin and call readObject() method [**oin.readObject() ]**

|  |
| --- |
| InputStream fin=**new** FileInputStream("ser.txt");  ObjectInput oin=**new** ObjectInputStream(fin);  System.*out*.println("DeSerialization process has started, displaying employee objects...");  Employee emp;  emp=(Employee)**oin.readObject();** |

1. **Difference between Externalizable and Serialization interface ?**

Here comes the time to **impress interviewer** by differentiating Serializable and Externalizable use.

|  |  |  |
| --- | --- | --- |
|  | [**SERIALIZABLE**](http://www.javamadesoeasy.com/2015/02/serialize-and-deserialize-object.html) | [**EXTERNALIZABLE**](http://www.javamadesoeasy.com/2015/02/serialize-and-deserialize-object-by.html) |
| Methods | It is a **marker** interface it doesn’t have any method. | It’s not a marker interface.  It has method’s called **writeExternal()** and **readExternal()** |
| Default Serialization process | **YES**, Serializable provides its own **default serialization process**, we just need to implement Serializable interface. | **NO**, we need to override **writeExternal()** and **readExternal()** for serialization process to happen. |
| Customize serialization process | We **can** customize **default serialization process** by **defining following** methods in our class >**readObject()** and **writeObject()**  **Note**: We are not overriding these methods, we are defining them in our class. | Serialization process is completely customized  We need to **override** Externalizable interface’s **writeExternal()** and **readExternal()** methods. |
| Control over Serialization | It provides **less control** over Serialization as it’s not mandatory to define **readObject()** and **writeObject()** methods. | Externalizable provides you **great control** over serialization process as it is important to override  **writeExternal()** and **readExternal()** methods. |
| Constructor call during **deSerialization** | Constructor is **not** called during deSerialization. | Constructor **is called** during deSerialization. |

1. **How can you customize Serialization and DeSerialization process when you have implemented Serializable interface ?**

 Here comes the quite **challenging** , where you could prove how strong your Serialization concepts are.We can [customize **Serialization** process by defining **writeObject()**  method & **DeSerialization** process by defining **readObject()** method](http://www.javamadesoeasy.com/2015/02/customize-serialization-process-by.html).

Let’s customize **Serialization** process by defining **writeObject()**  method :

|  |
| --- |
| **private void writeObject(ObjectOutputStream os) {**            System.*out*.println("In, writeObject() method.");  **try** {                   os.writeInt(**this**.id);                   os.writeObject(**this**.name);            } **catch** (Exception e) {                   e.printStackTrace();            }     } |

We have serialized id and name manually by writing them in file.

Let’s customize **DeSerialization** process by defining **readObject()**  method :

|  |
| --- |
| **private void readObject(ObjectInputStream ois) {**            System.*out*.println("In, readObject() method.");  **try** {                   id=ois.readInt();                   name=(String)ois.readObject();            } **catch** (Exception e) {                   e.printStackTrace();            }     } |

We have DeSerialized id and name manually by reading them from file.

1. **Wap to explain how can we Serialize and DeSerialize object by implementing Externalizable interface ?**

For [serializing object by implementing Externalizable interface](http://www.javamadesoeasy.com/2015/02/serialize-and-deserialize-object-by.html), we need to override writeExternal() and readExternal() for serialization process to happen.

For **Serialization** process override **writeExternal()**  method & for **DeSerialization** process by override **readExternal()** method.

Let’s customize **Serialization** process by overriding [**writeExternal()**](http://www.javamadesoeasy.com/2015/02/serialize-and-deserialize-object-by.html)method :

|  |
| --- |
| **public** **void** **writeExternal**(ObjectOutput oo) **throws** IOException {            System.*out*.println("in writeExternal()");            oo.writeInt(id);            oo.writeObject(name);    } |

We have serialized id and name manually by writing them in file.

Let’s customize **DeSerialization** process by overriding [**readExternal()**](http://www.javamadesoeasy.com/2015/02/serialize-and-deserialize-object-by.html)  method :

|  |
| --- |
| **public** **void** **readExternal**(ObjectInput in) **throws** IOException, ClassNotFoundException {            System.*out*.println("in readExternal()");  **this**.id=in.readInt();  **this**.name=(String)in.readObject();    } |

We have DeSerialized id and name manually by reading them from file.

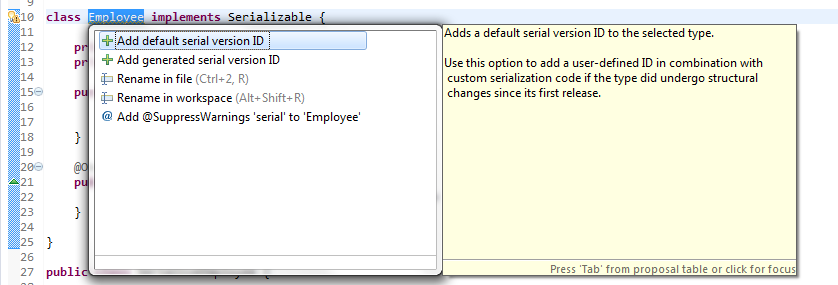
1. **How can you avoid certain member variables of class from getting Serialized?**

Mark member variables as [**static**](http://www.javamadesoeasy.com/2015/05/static-keyword-in-java-variable-method.html)or **transient**, and those member variables will no more be a part of Serialization.

1. **What is serialVersionUID?**

. The serialization at runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization.

We can use eclipse to generate serialVersionUID for our class (as done in below snapshot)



How to avoid **warning** ‘The serializable class Employee does not declare a static final serialVersionUID field of type long’ ?

Again answer is we can use eclipse to generate serialVersionUID for our class (as mentioned in above screenshot, click on warning button on left in line 10).

1. **What will be** [**impact of not defining serialVersionUID**](http://www.javamadesoeasy.com/2015/02/impact-of-not-defining-serialversionuid.html) **in class ?**

 This is one my favourite , i am going to discuss it in a very detailed manner. serialVersionUID is used for **version control of object**.

If we  don’t define serialVersionUID in the class, and any **modification** is made in class, then we **won’t be able to deSerialize our class** because **serialVersionUID generated by java compiler for modified class will be different from old serialized object**. And deserialization process will end up throwing **java.io.InvalidClassException**  (because of serialVersionUID mismatch)

Let’s frame another by twisting few words in it.

*If you have serialized a class & then added few fields in it and then deserialize already serialized version of class, how can you ensure that you don’t end up throwing* ***InvalidClassException****?*

**>**Simply we need to define **serialVersionUID** in class.

When we Deserialize class ( class which has been modified after Serialization and also class **doesn’t declare SerialVersionUID**) **InvalidClassException** is thrown.

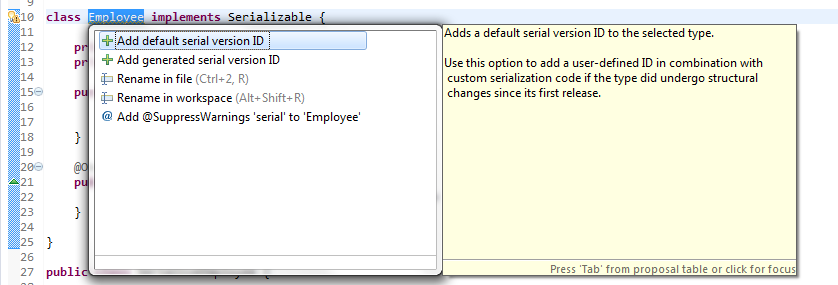
When we Deserialize class ( class which has been modified after Serialization and also class **declare SerialVersionUID**) its gets DeSerialized **successfully**.

serialVersionUIDis used for **version control of object**.

If we  don’t define serialVersionUID in the class, and any **modification** is made in class, then we **won’t be able to deSerialize our class** because **serialVersionUID generated by java compiler for modified class will be different from old serialized object**. And deserialization process will end up throwing **java.io.InvalidClassException**  (because of serialVersionUID mismatch)

The serialization at runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization.

We can use eclipse to generate serialVersionUID for our class (as done in below snapshot)



How to avoid **warning** ‘The serializable class Employee does not declare a static final serialVersionUID field of type long’ ?

Again answer is we can use eclipse to generate serialVersionUID for our class (as mentioned in above screenshot, click on warning button on left in line 10).

*If you have serialized a class & then added few fields in it and then deserialize already serialized version of class, how can you ensure that you don’t end up throwing* ***InvalidClassException****?*

**>**Simply we need to define **serialVersionUID** in class.

When we Deserialize class ( class which has been modified after Serialization and also class **doesn’t declare SerialVersionUID**) **InvalidClassException** is thrown.

When we Deserialize class ( class which has been modified after Serialization and also class **declare SerialVersionUID**) its gets DeSerialized **successfully**.

Let’s discuss this interesting topic in detail with programs-

First we will serialize a class (class which implements Serialization, but we haven’t declared SerialVersionUID)

**Program 1 -** to Serialize Object (without **serialVersionUID**)**>**

|  |
| --- |
| **package** serDeser4AddSUID;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.io.ObjectOutput;  **import** java.io.ObjectOutputStream;  **import** java.io.OutputStream;  **import** java.io.Serializable;  /\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  /\*Author : AnkitMittal  Copyright- contents must not be reproduced in any form\*/  **class** Employee **implements** Serializable {          //we haven’t declared **SerialVersionUId**  **private** Integer id;  **private** String name;    **public** Employee(Integer id, String name) {  **this**.id = id;  **this**.name = name;     }     @Override  **public** String toString() {  **return** "Employee [id=" + id + ", name=" + name + "]";     }  }  **public** **class** SerializeEmployee {  **public** **static** **void** main(String[] args) {            Employee object1 = **new** Employee(1, "amy");            Employee object2 = **new** Employee(2, "ankit");  **try** {                   OutputStream fout = **new** FileOutputStream("ser.txt");                   ObjectOutput oout = **new** ObjectOutputStream(fout);                   System.*out*.println("Serialization process has started, serializing employee objects...");                   oout.writeObject(object1);                   oout.writeObject(object2);                   fout.close();          oout.close();                   System.*out*.println("Object Serialization completed.");              } **catch** (IOException ioe) {                   ioe.printStackTrace();            }     }  }  /\*OUTPUT  Serialization process has started, serializing employee objects...  Object Serialization completed.  \*/ |

Then modify class by adding one field in class, but ensure that you **don’t run the Serialization process again**.

Modify the Serialized class (but don’t serialize the class again)**>**

|  |
| --- |
| **class** Employee **implements** Serializable {    **private** Integer id;  **private** String name;  **private String addedField;**  **public** Employee(Integer id, String name) {  **this**.id = id;  **this**.name = name;     }     @Override  **public** String toString() {  **return** "Employee [id=" + id + ", name=" + name + "]";     }  } |

Now, we have added **addedField** in class which was already Serialized, let’s see in absence of SerialVersionUID whether we will be able to DeSerialize our class or not.

**Program 2 -**  to DeSerialize object - program will throw **InvalidClassException>**

|  |
| --- |
| **package** serDeser4AddSUID;  **import** java.io.FileInputStream;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.io.ObjectInput;  **import** java.io.ObjectInputStream;  **public** **class** DeSerializeEmployee {    **public** **static** **void** main(String[] args){  **try**{                   InputStream fin=**new** FileInputStream("ser.txt");                   ObjectInput oin=**new** ObjectInputStream(fin);                     System.*out*.println("DeSerialization process has started, displaying employee objects...");                   Employee emp;  **while**( (emp=(Employee)oin.readObject())!=**null** ){                         System.*out*.println(emp);                   }                   fin.close();         oin.close();              }**catch**(IOException | ClassNotFoundException e){                   e.printStackTrace();            }              System.*out*.println("Object deSerialization completed.");         }  }  /\*OUTPUT  DeSerialization process has started, displaying employee objects...  java.io.InvalidClassException: serDeser4AddSUID.Employee; local class incompatible: stream classdesc serialVersionUID = 4822384361417160410, local class serialVersionUID = 5590647880449995492  Object deSerialization completed.     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 serDeser4AddSUID.DeSerializeEmployee.main(DeSerializeEmployee.java:18)  \*/ |

DeSerialization process has ended up throwing **InvalidClassException**.

Now, let’s see what will happen when we declare **serialVersionUID** in Serializable class.

**Program 3 -** to Serialize Object (with **serialVersionUID**)**>**

|  |
| --- |
| **package** serDeser4AddSUID;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.io.ObjectOutput;  **import** java.io.ObjectOutputStream;  **import** java.io.OutputStream;  **import** java.io.Serializable;  /\*Author : AnkitMittal  Copyright- contents must not be reproduced in any form\*/  **class** Employee **implements** Serializable {    **private** **static** **final** **long** *serialVersionUID* = 1L;  **private** Integer id;  **private** String name;    **public** Employee(Integer id, String name) {  **this**.id = id;  **this**.name = name;     }     @Override  **public** String toString() {  **return** "Employee [id=" + id + ", name=" + name + "]";     }  }  **public** **class** SerializeEmployee {  **public** **static** **void** main(String[] args) {            Employee object1 = **new** Employee(1, "amy");            Employee object2 = **new** Employee(2, "ankit");  **try** {                   OutputStream fout = **new** FileOutputStream("ser.txt");                   ObjectOutput oout = **new** ObjectOutputStream(fout);                   System.*out*.println("Serialization process has started, serializing employee objects...");                   oout.writeObject(object1);                   oout.writeObject(object2);                   fout.close();          oout.close();                   System.*out*.println("Object Serialization completed.");              } **catch** (IOException ioe) {                   ioe.printStackTrace();            }     }  }  /\*OUTPUT  Serialization process has started, serializing employee objects...  Object Serialization completed.  \*/ |

Then **modify** class by adding one field in class, but ensure that you **don’t run the Serialization process again**.

Modify the Serialized class (but don’t serialize the class again)**>**

|  |
| --- |
| **class** Employee **implements** Serializable {    **private** **static** **final** **long** *serialVersionUID* = 1L;  **private** Integer id;  **private** String name;  **private String addedField;**  **public** Employee(Integer id, String name) {  **this**.id = id;  **this**.name = name;     }     @Override  **public** String toString() {  **return** "Employee [id=" + id + ", name=" + name + "]";     }  } |

Now, we have added **addedField** in class which was already Serialized, let’s see in presence of SerialVersionUID whether we will be able to DeSerialize our class or not.

**Program 4 -** to DeSerialize object - Object will be DeSerialized **successfully (**without **InvalidClassException) >**

|  |
| --- |
| **package** serDeser4AddSUID;  **import** java.io.FileInputStream;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.io.ObjectInput;  **import** java.io.ObjectInputStream;  **public** **class** DeSerializeEmployee {    **public** **static** **void** main(String[] args){  **try**{                   InputStream fin=**new** FileInputStream("ser.txt");                   ObjectInput oin=**new** ObjectInputStream(fin);                     System.*out*.println("DeSerialization process has started, displaying employee objects...");                   Employee emp;  **while**( (emp=(Employee)oin.readObject())!=**null** ){                         System.*out*.println(emp);                   }                   fin.close();         oin.close();              }**catch**(IOException | ClassNotFoundException e){                   //e.printStackTrace();            }              System.*out*.println("Object deSerialization completed.");         }  }  /\*OUTPUT  DeSerialization process has started, displaying employee objects...  Employee [id=1, name=amy]  Employee [id=2, name=ankit]  Object deSerialization completed.  \*/ |

DeSerialization process has ended up **successfully**.

**SUMMARY**

**Deserialize class ( class has been modified after Serialization and also class doesn’t declare SerialVersionUID)**

* Serialize a class (class which implements Serialization, but **don’t declare SerialVersionUID**)
* Then modify class by adding one field in class, but ensure that you don’t run the Serialization process again.
* DeSerialization process will end up throwing **InvalidClassException**.

**Deserialize class ( class has been modified after Serialization and also class declare SerialVersionUID)**

>Now, let’s see what will happen when we **declare serialVersionUID** in Serializable class.

>Then modify class by adding one field in class, but ensure that you don’t run the Serialization process again.

>DeSerialization process will end **successfully**.

1. **What are compatible and incompatible changes in Serialization process?**

**Compatible Changes :**Compatible changes are those changes which **does not affect** deSerialization process even if class was updated after being serialized (provided serialVersionUID has been declared)

* **Adding new fields** - We can add new member variables in class.
* **Adding writeObject()/readObject()  methods** - We may add these methods to customize serialization process.
* **Removing writeObject()/readObject() methods** - We may remove these methods and then default customization process will be used.
* **Changing access modifier of a field** - The change to access modifiers i.e. public, default, protected, and private have no effect on the ability of serialization to assign values to the fields.
* **Changing a field from static to non static OR changing transient filed to non transient field**. - it’s like addition of fields.

**InCompatible Changes :**InCompatible changes are those changes which affect deSerialization process if class was updated after being serialized (provided serialVersionUID has been declared)

* **Deletion of fields.**
* **Changing a nonstatic field to static or  non transient field to transient field. -** it’s equal to deletion of fields.
* **Modifying the writeObject() / readObject() method** - we must not modify these method, though adding or removing them completely is compatible change.

1. **What if Serialization is not available, is any any other alternative way to transfer object over network?**

>We can can convert **JSON** to transfer the object. JSON is helpful in stringifying and de stringifying object.

>**Hibernate** (ORM tool) helps in persisting object as it in database and later we can read persisted object.

>We can convert object into **XML** (as done in web services) and transfer object over network.

1. **Why static member variables are not part of java serialization process ?**

Serialization is applicable on objects or primitive data types only, but [**static**](http://www.javamadesoeasy.com/2015/05/static-keyword-in-java-variable-method.html)members are **class level variables**, therefore, **different object’s of same class have same value for static member**.

So, serializing static member will consume unnecessary space and time.

Also, if modification is made in static member by any of the object, it won’t be in sync with other serialized object’s value.

1. **What is significance of transient variables?**

Serialization is not applicable on transient variables (it helps in saving time and space during Serialization process), we **must mark all rarely used variables as transient**. We can initialize transient variables during deSerialization by customizing deSerialization process.

1. **What will happen if one the member of class does not implement Serializable interface ?**

This is classy which will check your in depth knowledge of Serialization concepts. If any of the member does not implement Serializable than  NotSerializableException is thrown.

**Full Program/SourceCode** to show that if any of the member does not implement Serializable than  NotSerializableException is thrown>

|  |
| --- |
| **package** SerDeser10memberNotSer;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.io.ObjectOutput;  **import** java.io.ObjectOutputStream;  **import** java.io.OutputStream;  **import** java.io.Serializable;  **class** MyClass {}  /\*\* Copyright (c), AnkitMittal [JavaMadeSoEasy.com](http://javamadesoeasy.com/) \*/  /\*Author : AnkitMittal  Copyright- contents must not be reproduced in any form\*/  **class** Employee **implements** Serializable {    **private** **static** **final** **long** *serialVersionUID* = 1L;  **private** Integer id;  **private** MyClass myClass ;    **public** Employee(Integer id) {  **this**.id = id;            myClass=**new** MyClass();     }     @Override  **public** String toString() {  **return** "Employee [id=" + id + "]";     }  }  **public** **class** SerializeDeser {  **public** **static** **void** main(String[] args) {            Employee object1 = **new** Employee(8);  **try** {                   OutputStream fout = **new** FileOutputStream("ser.txt");                   ObjectOutput oout = **new** ObjectOutputStream(fout);                   System.*out*.println("Serialization process has started, serializing objects...");                   oout.writeObject(object1);                   System.*out*.println("Object Serialization completed.");                   fout.close();          oout.close();              } **catch** (IOException  e) {                   e.printStackTrace();            }     }  }  /\*OUTPUT  Serialization process has started, serializing objects...  java.io.NotSerializableException: SerDeser10memberNotSer.MyClass     at java.io.ObjectOutputStream.writeObject0(Unknown Source)     at java.io.ObjectOutputStream.defaultWriteFields(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 SerDeser10memberNotSer.SerializeConstructorCheck.main(SerializeConstructorCheck.java:42)  \*/ |

If we note output, myClass didn’t implemented Serializable interface that’s why Serialization process has thrown NotSerializableException.

**How to avoid** NotSerializableException?

We got to ensure that during Serialization all the members of class implements Serializable.

1. **What will happen if we have used List, Set and Map as member of class?**

This which will check your in depth knowledge of Serialization and Java Api’s. ArrayList, HashSet and HashMap implements Serializable interface, so if we will use them as member of class they will get Serialized and DeSerialized as well.

1. **Is constructor of class called during DeSerialization process?**

This which will check your in depth knowledge of Serialization and constructor chaining concepts. It depends on whether our object has implemented Serializable or Externalizable.

If **Serializable** has been implemented - constructor is **not called** during DeSerialization process.

But, if **Externalizable** has been implemented - constructor **is called** during DeSerialization process.

|  |
| --- |
| ObjectInput oin=**new** ObjectInputStream(fin);                   System.*out*.println("\nDeSerialization process has started, displaying employee objects...");                   Employee emp=(Employee)oin.readObject();                   System.*out*.println(emp);                   fin.close();         oin.close();         System.*out*.println("Object DeSerialization completed.");                } **catch** (IOException | ClassNotFoundException  e) {                   e.printStackTrace();            }     }  }  /\*OUTPUT  1-arg constructor called  Serialization process has started, serializing employee objects...  Object Serialization completed.  DeSerialization process has started, displaying employee objects...  **No-arg constructor called**  Employee [id=8]  Object DeSerialization completed.  \*/ |

If, we note output, **constructor** is **called** during DeSerialization process.

1. **Are primitive types part of serialization process?**

**Yes**, [primitive types are part of serialization process](http://www.javamadesoeasy.com/2015/02/are-primitive-types-part-of.html). Interviewer tends to check your basic java concepts over here.

1. **Is constructor of super class called during DeSerialization process of subclass ?**

Again your basic java concepts will be tested over here. It is depends on whether our superclass has implemented Serializable or not.

If superclass **has implemented Serializable** - constructor **is not called** during DeSerialization process.

If superclass has **not implemented Serializable** - constructor **is called** during DeSerialization process.

It is depends on whether our superclass has implemented Serializable or not.

If superclass **has implemented Serializable** - constructor **is not called** during DeSerialization process.

If superclass has **not implemented Serializable** - constructor **is called** during DeSerialization process.

**Full Program/SourceCode** to show that If superclass has implemented Serializable - constructor is not called during DeSerialization process.

|  |
| --- |
| **package** SerDeser9SuperConsCheck;  **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.io.ObjectInput;  **import** java.io.ObjectInputStream;  **import** java.io.ObjectOutput;  **import** java.io.ObjectOutputStream;  **import** java.io.OutputStream;  **import** java.io.Serializable;  /\*Author : AnkitMittal  Copyright- contents must not be reproduced in any form\*/  **class** Super **implements** Serializable{  **private** **static** **final** **long** *serialVersionUID* = 1L;  **public** Super(){            System.*out*.println("No-arg constructor of Super class");     }  }  **class** Sub **extends** Super {  //it automatically implements Serializable (because it's subclass implements Serializable).    **private** **static** **final** **long** *serialVersionUID* = 1L;  **private** Integer id;    **public** Sub(){            System.*out*.println("No-arg constructor of sub class");     }    **public** Sub(Integer id) {            System.*out*.println("1-arg constructor sub class");  **this**.id = id;     }     @Override  **public** String toString() {  **return** "Employee [id=" + id + "]";     }  }  **public** **class** SerializeDeser {  **public** **static** **void** main(String[] args) {            Sub object1 = **new** Sub(8);  **try** {                   OutputStream fout = **new** FileOutputStream("ser.txt");                   ObjectOutput oout = **new** ObjectOutputStream(fout);                   System.*out*.println("Serialization process has started, serializing objects...");                   oout.writeObject(object1);                   fout.close();          oout.close();                   System.*out*.println("Object Serialization completed.");                     //DeSerialization process >                       InputStream fin=**new** FileInputStream("ser.txt");                   ObjectInput oin=**new** ObjectInputStream(fin);                   System.*out*.println("\nDeSerialization process has started, displaying objects...");                   Sub subObj=(Sub)oin.readObject();                   System.*out*.println(subObj);                   fin.close();         oin.close();         System.*out*.println("Object DeSerialization completed.");                } **catch** (IOException | ClassNotFoundException  e) {                   e.printStackTrace();            }     }  }  /\*OUTPUT  No-arg constructor of Super class  1-arg constructor sub class  Serialization process has started, serializing objects...  Object Serialization completed.  DeSerialization process has started, displaying objects...  Employee [id=8]  Object DeSerialization completed.  \*/ |

If we note output, superclass **has implemented Serializable** and its constructor **is not called** during DeSerialization process.

**Full Program/SourceCode** to show that If superclass has not implemented Serializable - constructor is called during DeSerialization process.

**>**

|  |
| --- |
| **package** SerDeser9SuperConsCheck;  **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.io.ObjectInput;  **import** java.io.ObjectInputStream;  **import** java.io.ObjectOutput;  **import** java.io.ObjectOutputStream;  **import** java.io.OutputStream;  **import** java.io.Serializable;  /\*Author : AnkitMittal  Copyright- contents must not be reproduced in any form\*/  **class** Super {  **public** Super(){            System.*out*.println("No-arg constructor of Super class");     }  }  **class** Sub **extends** Super **implements** Serializable{    //it automatically implements Serializable (because it's subclass implements Serializable).    **private** **static** **final** **long** *serialVersionUID* = 1L;  **private** Integer id;    **public** Sub(){            System.*out*.println("No-arg constructor of sub class");     }    **public** Sub(Integer id) {            System.*out*.println("1-arg constructor sub class");  **this**.id = id;     }     @Override  **public** String toString() {  **return** "Employee [id=" + id + "]";     }  }  **public** **class** SerializeDeser {  **public** **static** **void** main(String[] args) {            Sub object1 = **new** Sub(8);  **try** {                   OutputStream fout = **new** FileOutputStream("ser.txt");                   ObjectOutput oout = **new** ObjectOutputStream(fout);                   System.*out*.println("Serialization process has started, serializing objects...");                   oout.writeObject(object1);                   fout.close();          oout.close();                   System.*out*.println("Object Serialization completed.");                     //DeSerialization process >                       InputStream fin=**new** FileInputStream("ser.txt");                   ObjectInput oin=**new** ObjectInputStream(fin);                   System.*out*.println("\nDeSerialization process has started, displaying objects...");                   Sub subObj=(Sub)oin.readObject();                   System.*out*.println(subObj);                   fin.close();         oin.close();         System.*out*.println("Object DeSerialization completed.");                } **catch** (IOException | ClassNotFoundException  e) {                   e.printStackTrace();            }     }  }  /\*OUTPUT  No-arg constructor of Super class  1-arg constructor sub class  Serialization process has started, serializing objects...  Object Serialization completed.  DeSerialization process has started, displaying objects...  No-arg constructor of Super class  Employee [id=8]  Object DeSerialization completed.  \*/ |

If we note output, superclass **has not implemented Serializable** and its constructor **is called** during DeSerialization process.

1. **18. What values will int and Integer will be initialized to during DeSerialization process if they were not part of Serialization?**

[int will be initialized to 0 and Integer will be initialized to null during DeSerialization](http://www.javamadesoeasy.com/2015/02/what-values-will-int-and-integer-will.html) (if they were not part of Serialization process).

1. **How you can avoid Deserialization process creating another instance of Singleton class ?**

This is another classy and very important which will check your in depth knowledge of Serialization and Singleton concepts. I’ll prefer you must understand this concept in detail. We can simply use **readResove()** method to return same instance of class, rather than creating a new one.

Defining readResolve() method ensures that we don't break singleton pattern during DeSerialization process.

|  |
| --- |
| **private** Object readResolve() **throws** ObjectStreamException {  **return** *INSTANCE*;   } |

Also define readObject() method, rather than creating new instance, assign current object to INSTANCE like done below :

|  |
| --- |
| **private** **void** **readObject**(ObjectInputStream ois) **throws** IOException,ClassNotFoundException{         ois.defaultReadObject();  **synchronized** (SingletonClass.**class**) {  **if** (*INSTANCE* == **null**) {  ***INSTANCE* = this;**          }         }   } |

1. **20. Can you Serialize Singleton class such that object returned by Deserialization process  is in same state as it was during Serialization time (regardless of any change made to it after Serialization)  ?**

It’s another very important which will be important in testing your Serialization and Singleton related concepts, you must try to understand the concept and in detail.

**YES**, we can Serialize Singleton class such that object returned by Deserialization process is in same state as it was during Serialization time (regardless of any change made to it after Serialization)

Defining readResolve() method ensures that we don't break singleton pattern during DeSerialization process.

|  |
| --- |
| **private** Object readResolve() **throws** ObjectStreamException {  **return** *INSTANCE*;   } |

Also define readObject() method, rather than creating new instance, assign current object to INSTANCE like done below :

|  |
| --- |
| **private** **void** **readObject**(ObjectInputStream ois) **throws** IOException,ClassNotFoundException{         ois.defaultReadObject();  **synchronized** (SingletonClass.**class**) {  **if** (*INSTANCE* == **null**) {  ***INSTANCE* = this;**          }         }   } |

1. **Purpose of serializing Singleton class OR  purpose of saving singleton state?**

Let’s take example of our laptop, daily eod we need to shut it down, but rather than shutting it down hibernate (save state of  laptop) is better option because it enables us to resume at same point where we leaved it, like wise serializing singleton OR saving state of Singleton can be very handy.

1. **How can subclass avoid Serialization if its superClass has implemented Serialization interface ?**

If superClass has implemented Serializable that means subclass is also Serializable (**as subclass always inherits all features from its parent class**), for avoiding Serialization in sub-class we can **define writeObject()** method and **throw NotSerializableException()** from there as done below.

|  |
| --- |
| **private void writeObject(ObjectOutputStream os) throws NotSerializableException {**  **throw new NotSerializableException("This class cannot be Serialized");**  **}** |

1. **Find output of following code :**

|  |
| --- |
| **package** serDeser6ListSetMap;  **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.io.ObjectInput;  **import** java.io.ObjectInputStream;  **import** java.io.ObjectOutput;  **import** java.io.ObjectOutputStream;  **import** java.io.OutputStream;  **import** java.io.Serializable;  **import** java.util.ArrayList;  **import** java.util.HashMap;  **import** java.util.HashSet;  **import** java.util.List;  **import** java.util.Map;  **import** java.util.Set;  /\*Author : AnkitMittal  Copyright- contents must not be reproduced in any form\*/  **class** MyClass **implements** Serializable {    **private** **static** **final** **long** *serialVersionUID* = 1L;  **private** List<Integer> list;  **private** Set<Integer> set;  **private** Map<Integer,Integer> map;    **public** MyClass(List<Integer> list, Set<Integer> set,                   Map<Integer, Integer> map) {  **super**();  **this**.list = list;  **this**.set = set;  **this**.map = map;     }     @Override  **public** String toString() {  **return** "MyClass [list=" + list + ", set=" + set + ", map=" + map + "]";     }    }  **public** **class** SerializeEmployee {  **public** **static** **void** main(String[] args) {            List<Integer> list=**new** ArrayList<Integer>();            list.add(2);            list.add(3);            Set<Integer> set=**new** HashSet<Integer>();            set.add(4);            set.add(5);            Map<Integer, Integer> map=**new** HashMap<Integer,Integer>();            map.put(6, 34);            map.put(7, 35);            MyClass object1 = **new** MyClass(list,set,map);  **try** {                   OutputStream fout = **new** FileOutputStream("ser.txt");                   ObjectOutput oout = **new** ObjectOutputStream(fout);                   System.*out*.println("Serialization process has started, serializing objects...");                   oout.writeObject(object1);                   fout.close();          oout.close();          System.*out*.println("Object Serialization completed.");                     //DeSerialization process >                       InputStream fin=**new** FileInputStream("ser.txt");                   ObjectInput oin=**new** ObjectInputStream(fin);                   System.*out*.println("\nDeSerialization process has started, displaying objects...");                   MyClass object=(MyClass)oin.readObject();                   System.*out*.println(object);                   fin.close();         oin.close();         System.*out*.println("Object DeSerialization completed.");              } **catch** (IOException | ClassNotFoundException  e) {                   e.printStackTrace();            }     }  } |

Here intention of interviewer will be to find out whether you know that list, set and map can be serialized or not.

/\*OUTPUT

Serialization process has started, serializing objects...

Object Serialization completed.

DeSerialization process has started, dispalying objects...

MyClass [list=[2, 3], set=[4, 5], map={6=34, 7=35}]

Object DeSerialization completed.

\*/

1. **Find output of following code  :**

|  |
| --- |
| **package** SerDeser10memberNotSer;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.io.ObjectOutput;  **import** java.io.ObjectOutputStream;  **import** java.io.OutputStream;  **import** java.io.Serializable;  **class** MyClass {}  /\*Author : AnkitMittal  Copyright- contents must not be reproduced in any form\*/  **class** Employee **implements** Serializable {    **private** **static** **final** **long** *serialVersionUID* = 1L;  **private** Integer id;  **private** MyClass myClass ;    **public** Employee(Integer id) {  **this**.id = id;            myClass=**new** MyClass();     }     @Override  **public** String toString() {  **return** "Employee [id=" + id + "]";     }  }  **public** **class** SerializeDeser {  **public** **static** **void** main(String[] args) {            Employee object1 = **new** Employee(8);  **try** {                   OutputStream fout = **new** FileOutputStream("ser.txt");                   ObjectOutput oout = **new** ObjectOutputStream(fout);                   System.*out*.println("Serialization process has started, serializing objects...");                   oout.writeObject(object1);                   System.*out*.println("Object Serialization completed.");                   fout.close();          oout.close();              } **catch** (IOException  e) {                   e.printStackTrace();            }     }  } |

Here intention of interviewer will be to find out whether you know that if any of the member does not implement Serializable than  NotSerializableException is thrown.

/\*OUTPUT

Serialization process has started, serializing objects...

java.io.NotSerializableException: SerDeser10memberNotSer.MyClass

   at java.io.ObjectOutputStream.writeObject0(Unknown Source)

   at java.io.ObjectOutputStream.defaultWriteFields(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 SerDeser10memberNotSer.SerializeConstructorCheck.main(SerializeConstructorCheck.java:42)

\*/