Serialization

In computer science, in the context of data storage, serialization is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer, or transmitted across a network connection link) and reconstructed later in the same or another computer environment.

When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object. For many complex objects, such as those that make extensive use of references, this process is not straightforward. Serialization of object-oriented objects does not include any of their associated methods with which they were previously inextricably linked.

This process of serializing an object is also called marshaling an object. The opposite operation, extracting a data structure from a series of bytes, is deserialization (which is also called unmarshalling).

Uses

- a method of remote Procedure Calls, e.g., as in SOAP.
- a method for distributing objects, especially in component-based software engineering such as COM, CORBA, etc.
- a method for detecting changes in time-varying data.

For some of these features to be useful, architecture independence must be maintained.
For example, for maximal use of distribution, a computer running on a different hardware architecture should be able to reliably reconstruct a serialized data stream, regardless of endianness. This means that the

simpler and faster procedure of directly copying the memory layout of the data structure cannot work reliably for all architectures. Serializing the data structure in an architecture independent format means to prevent the problems of byte ordering, memory layout, or simply different ways of representing data structures in different programming languages.

Inherent to any serialization scheme is that, because the encoding of the data is by definition serial, extracting one part of the serialized data structure requires that the entire object be read from start to end, and reconstructed. In many applications this linearity is an asset, because it enables simple, common I/O interfaces to be utilized to hold and pass on the state of an object. In applications where higher performance is an issue, it can make sense to expend more effort to deal with a more complex, non-linear storage organization.

Even on a single machine, primitive pointer objects are too fragile to save because the objects to which they point may be reloaded to a different location in memory. To deal with this, the serialization process includes a step called unswizzling or pointer unswizzling, where direct pointer references are converted to references based on name or position . The deserialization process includes an inverse step called pointer swizzling.

Since both serializing and deserializing can be driven from common code (for example, the Serialize function in Microsoft Foundation Classes), it is possible for the common code to do both at the same time, and thus, 1) detect differences between the objects being serialized and their prior copies, and 2) provide the input for the next such detection. It is not necessary to actually build the prior copy because differences can be detected on the fly. The technique is called differential execution. It is useful in the programming of user interfaces whose contents are time-varying — graphical objects can be created, removed, altered, or made to handle input events without necessarily having to write separate code to do those things.

Consequences

Serialization, however, breaks the opacity of an abstract data type by potentially exposing private implementation details. Trivial implementations which serialize all data members may violate encapsulation.
To discourage competitors from making compatible products, publishers of proprietary software often keep the details of their programs' serialization formats a trade secret. Some deliberately obfuscate or even encrypt the serialized data. Yet, interoperability requires that applications be able to understand each other's serialization formats. Therefore, remote method call architectures such as CORBA define their serialization formats in detail.

Many institutions, such as archives and libraries, attempt to future proof their backup archives—in particular, database dumps—by storing them in some relatively human-readable serialized format.

Serialization formats

The Xerox Network Systems Courier technology in the early 1980s influenced the first widely adopted standard. Sun Microsystems published the External Data Representation (**XDR**) in 1987.
In the late 1990s, a push to provide an alternative to the standard serialization protocols started: XML was used to produce a human readable text-based encoding. Such an encoding can be useful for persistent objects that may be read and understood by humans, or communicated to other systems regardless of programming language. It has the disadvantage of losing the more compact, byte-stream-based encoding, but by this point larger storage and transmission capacities made file size less of a concern than in the early days of computing. Binary XML had been proposed as a compromise which was not readable by plain-text editors, but was more compact than regular XML. In the 2000s, XML was often used for

asynchronous transfer of structured data between client and server in Ajax web applications.

JSON is a more lightweight plain-text alternative to XML which is also commonly used for client-server communication in web applications. JSON is based on JavaScript syntax, but is supported in other programming languages as well.

Another alternative, YAML, is effectively a superset of JSON and includes features that make it more powerful for serialization, more "human friendly," and potentially more compact. These features include a notion of tagging data types, support for non-hierarchical data structures, the option to structure data with indentation, and multiple forms of scalar data quoting.

Another human-readable serialization format is the property list format used in NeXTSTEP, GNUstep, and Mac OS X Cocoa.

For large volume scientific datasets, such as satellite data and output of numerical climate, weather, or ocean models, specific binary serialization standards have been developed, e.g. HDF, netCDF and the older GRIB.

Programming language support

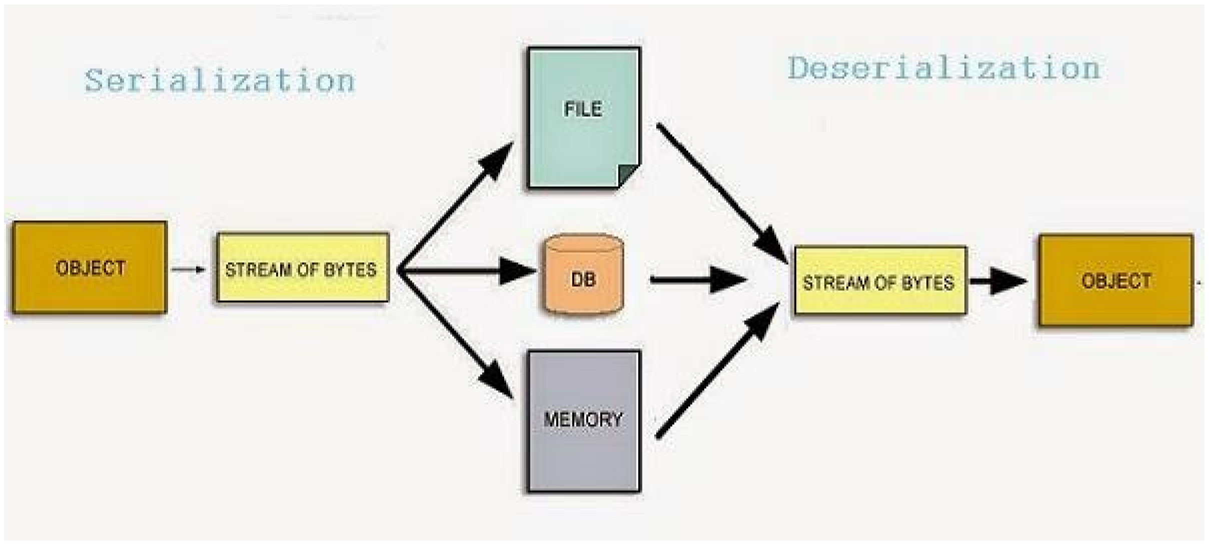
- Java
- CFML
- OCaml
- Perl
- C and C++
- Python
- PHP
- R
- REBOL
- Ruby

- Smalltalk

Java Serialization

To *serialize* an object means to convert its state(including its references) to a byte stream so that the byte stream can be reverted back into a copy of the object.

"Java serialization is the process which is used to serialize an object in java by storing the object's state into a file with extension .ser and recreating the object's state from that file."



When is serialization used?

Serialization is used when you want to persist the object. It is also used by RMI to pass objects between JVMs, either as arguments in a method invocation from a client to a server or as return values from a method invocation. In general, serialization is used when we want the object to exist beyond the lifetime of the JVM. It is mainly used in Hibernate, RMI, JPA, EJB, JMS technologies.

- To persist data for future use.
- To send data to a remote computer using such client/server Java technologies as RMI or socket programming.
- To "flatten" an object into an array of bytes in memory.

- To exchange data between applets and servlets.
- To store user sessions in Web applications.
- To activate/passivate enterprise javabeans.
- To send objects between the servers in a cluster.

Lets see couple of different scenarios (examples) where we use serialization:-

- **Banking example:** When the account holder tries to withdraw money from the server through ATM, the account holder information along with the withdrawal details will be serialized (marshaled/flattened to bytes) and sent to server where the details are deserialized (unmarshalled/rebuilt the bytes)and used to perform operations. This will reduce the network calls as we are serializing the whole object and sending it to the server and further requests for information from the client are not needed by the server.
- **Stock example:** Let's say an user wants the stock updates immediately when he requests it. To achieve this, everytime we have an update, we can serialize it and save it in a file. When a user requests the information, deserialize it from the file and provide the information. This way we don't need to make the user wait for the information until we hit the database, perform computations and get the result.

Fast facts:-

1. Transient and static fields are ignored in serialization. After de-serialization **transient fields and non-final static fields will be null. Final static fields** still have values since they are part of the class data.
2. **ObjectOutputStream.writeObject(obj)** and **ObjectInputStream.readObject()** are used in serialization and deserialization.

3. During serialization, you need to handle **IOException**; during deserialization, you need to handle **IOException** and **ClassNotFoundException**. So the deserialized class type must be in the classpath.
4. Uninitialized, non-serializable, non-transient instance fields are tolerated. When adding “private Thread th;”, no error in serializable. However, “private Thread threadClass = new Thread();” will cause an exception.
5. Serialization and deserialization can be used for copying and cloning objects. It is slower than regular **clone**, but can produce a deep copy very easily.
6. If I need to serialize a Serializable class Employee, but one of its superclasses is not Serializable, can the Employee class still be serialized and deserialized? The answer is **yes**, provided that the non-serializable superclass has a no-arg constructor, which is invoked at de-serialization to initialize that superclass.
7. You must be careful while modifying a class implementing **java.io.Serializable**. If a class does not contain a **serialVersionUID** field, its serialVersionUID will be automatically generated by the compiler. Different compilers, or different versions of the same compiler, will generate potentially different values.
8. Computation of serialVersionUID is based not only on fields, but also on other aspects of the class like implement clause, constructors, etc. So the best practice is to explicitly declare a serialVersionUID field to maintain backward compatibility. If you need to modify the serializable class substantially and expect it to be incompatible with previous versions, then you need to increment serialVersionUID to avoid mixing different versions.

The Java Serialization API provides a standard mechanism for developers to handle object serialization using the **Serializable** and **Externalizable** interface.

A Java object is *serializable* if its class or any of its superclasses implements either the **java.io.Serializable interface** or its sub interface, **java.io.Externalizable**.

Certain system-level classes such as Thread, OutputStream and its subclasses, and Socket are not serializable. If your serializable class contains such objects, it must mark them as "transient".

java.io.Serializable interface

Serializability of a class is enabled by the class implementing the java.io.Serializable interface.

Classes that do not implement this interface will not have any of their state serialized or deserialized.

All subtypes of a serializable class are themselves serializable.

The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.

To allow subtypes of non-serializable classes to be serialized, the subtype may assume responsibility for saving and restoring the state of the supertype's public, protected, and (if accessible) package fields.
The subtype may assume this responsibility only if the class it extends has an accessible no-arg constructor to initialize the class's state. It is an error to declare a class Serializable if this is not the case. The error will be detected at runtime.

During deserialization, the fields of non-serializable classes will be initialized using the public or protected no-arg constructor of the class. A no-arg constructor must be accessible to the subclass that is serializable. The fields of serializable subclasses will be restored from the stream.

When traversing a graph, an object may be encountered that does not support the Serializable interface. In this case the **NotSerializableException** will be thrown and will identify the class of the non-serializable object.

Serialization process is implemented by **ObjectInputStream** and **ObjectOutputStream**, so all we need is a wrapper over them to either save it to file or send it over the network.

Java Serialization Methods

We have seen that java serialization is automatic and all we need is implementing a Serializable interface. The implementation is present in the **ObjectInputStream** and **ObjectOutputStream classes**.

But what if we want to change the way we are saving data, for example we have some sensitive information in the object and before saving/retrieving we want to encrypt/decrypt it.

That's why there are four methods that we can provide in the class to change the serialization behavior. If these methods are present in the class, they are used for serialization purposes.

- **readObject(ObjectInputStream ois):** If this method is present in the class, ObjectInputStream readObject() method will use this method for reading the object from the stream.
- **writeObject(ObjectOutputStream oos):** If this method is present in the class, ObjectOutputStream writeObject() method will use this

method for writing the object to stream. One of the common usages is to obscure the object variables to maintain data integrity.

- **Object writeReplace():** If this method is present, then after the serialization process this method is called and the object returned is serialized to the stream.
- **Object readResolve():** If this method is present, then after the deserialization process, this method is called to return the final object to the caller program. One of the uses of this method is to implement Singleton pattern with Serialized classes.

Usually while implementing above methods, it's kept as **private** so that subclasses can't override them. They are meant for serialization purposes only and keeping them private avoids any security issue.

ObjectOutputStream class

The ObjectOutputStream class is used to write primitive data types and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams.

public ObjectOutputStream(OutputStream out) throws IOException {}
creates an ObjectOutputStream that writes to the specified OutputStream.

Classes that require special handling during the serialization and deserialization process must implement special methods with these exact signatures:

- **private void writeObject(java.io.ObjectOutputStream out) throws IOException**
- **private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException;**
- **private void readObjectNoData() throws ObjectStreamException;**

public final void writeObject(Object obj) throws IOException {}: -
The writeObject method is responsible for writing the state of the object for its particular class so that the corresponding readObject method can restore it.
The default mechanism for saving the Object's fields can be invoked by calling out.defaultWriteObject.
The method does not need to concern itself with the state belonging to its superclasses or subclasses.
State is saved by writing the individual fields to the ObjectOutputStream using the writeObject method or by using the methods for primitive data types supported by DataOutput.

private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException:- The readObject method is responsible for reading from the stream and restoring the classes' fields.
It may call in.defaultReadObject to invoke the default mechanism for restoring the object's non-static and non-transient fields.

The defaultReadObject method uses information in the stream to assign the fields of the object saved in the stream with the correspondingly named fields in the current object. This handles the case when the class has evolved to add new fields.
The method does not need to concern itself with the state belonging to its superclasses or subclasses. State is saved by writing the individual fields to the ObjectOutputStream using the writeObject method or by using the methods for primitive data types supported by DataOutput.

private void readObjectNoData() throws ObjectStreamException:-
The readObjectNoData method is responsible for initializing the state of the object for its particular class in the event that the serialization stream does not list the given class as a superclass of the object being deserialized.
This may occur in cases where the receiving party uses a different version

of the deserialized instance's class than the sending party, and the receiver's version extends classes that are not extended by the sender's version. This may also occur if the serialization stream has been tampered; hence, readObjectNoData is useful for initializing deserialized objects properly despite a "hostile" or incomplete source stream.

Serializable classes that need to designate an alternative object to be used when writing an object to the stream should implement this special method with the exact signature:

ANY-ACCESS-MODIFIER Object writeReplace() throws ObjectStreamException;

This writeReplace method is invoked by serialization if the method exists and it would be accessible from a method defined within the class of the object being serialized. Thus, the method can have private, protected and package-private access. Subclass access to this method follows java accessibility rules.

Classes that need to designate a replacement when an instance of it is read from the stream should implement this special method with the exact signature.

ANY-ACCESS-MODIFIER Object readResolve() throws ObjectStreamException;
This readResolve method follows the same invocation rules and accessibility rules as writeReplace.

Example

Create the following Java object called *Person*.

```
package de.vogella.java.serilization;
```

```
import java.io.Serializable;

public class Person implements Serializable {
  private String firstName;
  private String lastName;
  // stupid example for transient
  transient private Thread myThread;

  public Person(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.myThread = new Thread();
  }

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  @Override
  public String toString() {
    return "Person [firstName=" + firstName + ", lastName=" + lastName
        + "]";
  }

}
```

The following code example shows you how you can serializable and deserializable this object.

```
package de.vogella.java.serilization;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class Main {
  public static void main(String[] args) {
    String filename = "time.ser";
    Person p = new Person("Lars", "Vogel");

    // save the object to file
    FileOutputStream fos = null;
    ObjectOutputStream out = null;
    try {
      fos = new FileOutputStream(filename);
      out = new ObjectOutputStream(fos);
      out.writeObject(p);

      out.close();
    } catch (Exception ex) {
      ex.printStackTrace();
    }
    // read the object from file
    // save the object to file
    FileInputStream fis = null;
    ObjectInputStream in = null;
    try {
      fis = new FileInputStream(filename);
      in = new ObjectInputStream(fis);
      p = (Person) in.readObject();
      in.close();
    } catch (Exception ex) {
      ex.printStackTrace();
    }
    System.out.println(p);
  }
}
```

Java Serialization with Inheritance (IS-A Relationship)

If a class implements serializable then all its subclasses will also be serializable.

Let's see the example given below:

```
import java.io.Serializable;
class Person implements Serializable{
 int id;
 String name;
 Person(int id, String name) {
  this.id = id;
  this.name = name;
 }
}


class Student extends Person{
 String course;
 int fee;
 public Student(int id, String name, String course, int fee) {
  super(id,name);
  this.course=course;
  this.fee=fee;
 }
}
```

Now you can serialize the Student class object that extends the Person class which is Serializable. Parent class properties are inherited to subclasses so if the parent class is Serializable, subclasses would also be.

Java Serialization with Inheritance (**Serialize superclass state even though it's not implementing Serializable interface**)

Sometimes we need to extend a class that doesn't implement a Serializable interface. If we rely on the automatic serialization behavior and the superclass has some state, then they will not be converted to stream and hence not retrieved later on.

This is one place where **readObject**() and **writeObject**() methods really help.
By providing their implementation, we can save the superclass state to the stream and then retrieve it later on. Let's see this in action.

We can serialize superclass state even though it's not implementing a Serializable interface. This strategy comes handy when the superclass is a third-party class that we can't change.

SuperClass is a simple java bean but it's not implementing a Serializable interface.

```
package com.journaldev.serialization.inheritance;

public class SuperClass {

    private int id;
    private String value;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getValue() {
        return value;
    }
    public void setValue(String value) {
        this.value = value;
    }

}
```

```
package com.journaldev.serialization.inheritance;

import java.io.IOException;
import java.io.InvalidObjectException;
import java.io.ObjectInputStream;
import java.io.ObjectInputValidation;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class SubClass extends SuperClass implements Serializable,
ObjectInputValidation{

    private static final long serialVersionUID = -1322322139926390329L;

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
    @Override
    public String toString(){
        return
"SubClass{id="+getId()+",value="+getValue()+",name="+getName()+"}";
    }

    //adding helper method for serialization to save/initialize super class
state
    private void readObject(ObjectInputStream ois) throws
ClassNotFoundException, IOException{
        ois.defaultReadObject();

        //notice the order of read and write should be same
        setId(ois.readInt());
        setValue((String) ois.readObject());

    }

    private void writeObject(ObjectOutputStream oos) throws IOException{
        oos.defaultWriteObject();

        oos.writeInt(getId());
        oos.writeObject(getValue());
    }

    @Override
    public void validateObject() throws InvalidObjectException {
        //validate the object here
        if(name == null || "".equals(name)) throw new
InvalidObjectException("name can't be null or empty");
        if(getId() <=0) throw new InvalidObjectException("ID can't be
negative or zero");
    }

}
```

Notice that the order of writing and reading the extra data to the stream should be the same.

We can put some logic in reading and writing data to make it secure.

Also notice that the class is implementing an **ObjectInputValidation interface**. By implementing the **validateObject*()***method, we can put some business validations to make sure that the data integrity is not harmed.

Let’s write a test class and see if we can retrieve the superclass state from serialized data or not.

```
package com.journaldev.serialization.inheritance;

import java.io.IOException;

import com.journaldev.serialization.SerializationUtil;

public class InheritanceSerializationTest {

    public static void main(String[] args) {
        String fileName = "subclass.ser";

        SubClass subClass = new SubClass();
        subClass.setId(10);
        subClass.setValue("Data");
        subClass.setName("Pankaj");

        try {
            SerializationUtil.serialize(subClass, fileName);
        } catch (IOException e) {
            e.printStackTrace();
            return;
        }

        try {
            SubClass subNew = (SubClass)
SerializationUtil.deserialize(fileName);
            System.out.println("SubClass read = "+subNew);
        } catch (ClassNotFoundException | IOException e) {
            e.printStackTrace();
        }
    }

}
```

When we run above class, we get the following output.

```
SubClass read = SubClass{id=10,value=Data,name=Pankaj}
```

Java Serialization with Aggregation (HAS-A Relationship)

If a class has a reference to another class, all the references must be Serializable otherwise the serialization process will not be performed. In such cases, ***NotSerializableException*** is thrown at runtime.

```
class Address{
 String addressLine,city,state;
 public Address(String addressLine, String city, String state) {
  this.addressLine=addressLine;
  this.city=city;
  this.state=state;
 }
}


import java.io.Serializable;
public class Student implements Serializable{
 int id;
 String name;
 Address address;//HAS-A
 public Student(int id, String name) {
  this.id = id;
  this.name = name;
 }
}
```

Since Address is not Serializable, you can not serialize the instance of Student class.

Note: All the objects within an object must be Serializable.

Java Serialization with static data member

```
class Employee implements Serializable{
 int id;
 String name;
 static String company="SSS IT Pvt Ltd";//it won't be serialized
 public Student(int id, String name) {
  this.id = id;
  this.name = name;
 }
}
```

Java Serialization with array or collection

Rule: In case of array or collection, all the objects of array or collection must be serializable. If any object is not serializable, serialization will fail.

Serialization Proxy Pattern

Java Serialization comes with some serious pitfalls such as:-

- The class structure can’t be changed a lot without breaking the serialization process. So even though we don’t need some variables later on, we need to keep them just for backward compatibility.

- Serialization causes huge security risks, an attacker can change the stream sequence and cause harm to the system. For example, the user role is serialized and an attacker changes the stream value to make it admin and run malicious code.

Serialization Proxy pattern is a way to achieve greater security with Serialization. In this pattern, an inner private static class is used as a proxy class for serialization purposes. This class is designed in a way to maintain the state of the main class. This pattern is implemented by properly implementing *readResolve()*and *writeReplace()* methods.

Let us first write a class which implements serialization proxy pattern and then we will analyze it for better understanding.

```
package com.journaldev.serialization.proxy;

import java.io.InvalidObjectException;
import java.io.ObjectInputStream;
import java.io.Serializable;

public class Data implements Serializable{

    private static final long serialVersionUID = 2087368867376448459L;

    private String data;

    public Data(String d){
        this.data=d;
    }

    public String getData() {
        return data;
    }
    public void setData(String data) {
        this.data = data;
    }

    @Override
    public String toString(){
        return "Data{data="+data+"}";
    }

    //serialization proxy class
    private static class DataProxy implements Serializable{

        private static final long serialVersionUID = 8333905273185436744L;

        private String dataProxy;
        private static final String PREFIX = "ABC";
        private static final String SUFFIX = "DEFG";

        public DataProxy(Data d){
            //obscuring data for security
            this.dataProxy = PREFIX + d.data + SUFFIX;
        }

        private Object readResolve() throws InvalidObjectException {
            if(dataProxy.startsWith(PREFIX) &&
dataProxy.endsWith(SUFFIX)){
            return new Data(dataProxy.substring(3, dataProxy.length() -4));
            }else throw new InvalidObjectException("data corrupted");
        }

    }
    //replacing serialized object to DataProxy object
    private Object writeReplace(){
        return new DataProxy(this);
    }

    private void readObject(ObjectInputStream ois) throws
InvalidObjectException{
        throw new InvalidObjectException("Proxy is not used, something
fishy");
    }
}
```

Description:-

- Both Data and DataProxy class should implement Serializable interface.
- DataProxy should be able to maintain the state of the Data object.
- DataProxy is an inner private static class, so that other classes can't access it.
- DataProxy should have a single constructor that takes Data as an argument.
- The Data class should provide a **writeReplace**() method returning a DataProxy instance. So when a Data object is serialized, the returned stream is of DataProxy class. However DataProxy class is not visible outside, so it can't be used directly.
- The DataProxy class should implement a **readResolve**() method returning a Data object. So when the Data class is deserialized, internally DataProxy is deserialized and when its **readResolve**() method is called, we get a Data object.
- Finally implement readObject() method in Data class and throw InvalidObjectException to avoid hackers attacking trying to fabricate Data object stream and parse it.

Let’s write a small test to check whether implementation works or not.

```
package com.journaldev.serialization.proxy;

import java.io.IOException;

import com.journaldev.serialization.SerializationUtil;

public class SerializationProxyTest {

    public static void main(String[] args) {
        String fileName = "data.ser";

        Data data = new Data("Pankaj");

        try {
            SerializationUtil.serialize(data, fileName);
        } catch (IOException e) {
            e.printStackTrace();
        }

        try {
            Data newData = (Data) SerializationUtil.deserialize(fileName);
            System.out.println(newData);
        } catch (ClassNotFoundException | IOException e) {
            e.printStackTrace();
        }
    }

}
```

When we run above class, we get below output in the console.

```
Data{data=Pankaj}
```

If you open the **data.ser file**, you can see that DataProxy object is saved as a stream in the file.

Java Serialization and Singleton Design Pattern

Sometimes in distributed systems, we need to implement a Serializable interface in Singleton class so that we can store its state in the file system and retrieve it at a later point of time.

Here is a small singleton class that implements Serializable interface also.

```
package com.journaldev.singleton;

import java.io.Serializable;

public class SerializedSingleton implements Serializable{

    private static final long serialVersionUID = -7604766932017737115L;

    private SerializedSingleton(){}

    private static class SingletonHelper{
        private static final SerializedSingleton instance = new
SerializedSingleton();
    }

    public static SerializedSingleton getInstance(){
        return SingletonHelper.instance;
    }

}
```

The problem with the above serialized singleton class is that whenever we deserialize it, it will create a new instance of the class.

Let’s see it with a simple program.

```
package com.journaldev.singleton;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;

public class SingletonSerializedTest {

   public static void main(String[] args) throws FileNotFoundException,
IOException, ClassNotFoundException {
       SerializedSingleton instanceOne =
SerializedSingleton.getInstance();
       ObjectOutput out = new ObjectOutputStream(new
FileOutputStream(
               "filename.ser"));
       out.writeObject(instanceOne);
       out.close();

       //deserailize from file to object
       ObjectInput in = new ObjectInputStream(new FileInputStream(
               "filename.ser"));
       SerializedSingleton instanceTwo = (SerializedSingleton)
in.readObject();
       in.close();

       System.out.println("instanceOne
hashCode="+instanceOne.hashCode());
       System.out.println("instanceTwo
hashCode="+instanceTwo.hashCode());

   }
}
```

Output of the above program is:-

```
instanceOne hashCode=2011117821
instanceTwo hashCode=109647522
```

So it destroys the singleton pattern, to overcome this scenario all we need to do is provide the implementation of **readResolve() method.**

```
protected Object readResolve() {
    return getInstance();
}
```

After this you will notice that hashCode of both the instances are the same in the test program.

Write Object to File.

Once an object is converted to Stream, it can be saved to file or send over the network or used in socket connections. The object should implement a Serializable interface and we can use **java.io.ObjectOutputStream** to write an object to file or to any **OutputStream** object.

Let's create an Object that we will save into the file.

```
package com.journaldev.files;

import java.io.Serializable;

public class EmployeeObject implements Serializable {

    private static final long serialVersionUID = -299482035708790407L;

    private String name;
    private String gender;
    private int age;
    private String role;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getGender() {
        return gender;
    }
    public void setGender(String gender) {
        this.gender = gender;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getRole() {
        return role;
    }
    public void setRole(String role) {
        this.role = role;
    }

    @Override
    public String toString() {
        return "Employee:: Name=" + this.name + " Age=" + this.age + "
Gender=" + this.gender +
                " Role=" + this.role;
    }

}
```

Here is the program showing how to write an Object to file in java.

```
package com.journaldev.files;
```

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class ObjectToFile {

    public static void main(String[] args) {
        EmployeeObject emp = new EmployeeObject();

        emp.setAge(10);
        emp.setGender("Male");
        emp.setName("Pankaj");

        try {
            FileOutputStream fos = new
FileOutputStream("EmployeeObject.ser");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            //write object to file
            oos.writeObject(emp);
            System.out.println("Done");
            //closing resources
            oos.close();
            fos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }

    }

}
```

Here is the file contents:-

```
¨Ìsr#com.journaldev.files.EmployeeObject˚ÿ¨‹˜yyIageLgendertLjava/lang/S
tring;Lnameq~Lroleq~xp
tMaletPankajp
```

The file contents are garbled but that’s how it gets saved into file in java.

Class Refactoring with Serialization and serialVersionUID

Java Serialization permits some changes in the java class if they can be ignored. Some of the changes in class that will not affect the deserialization process are:-

- Adding new variables to the class
- Changing the variables from transient to non-transient, for serialization it’s like having a new field.
- Changing the variable from static to non-static, for serialization it’s like having a new field.

But for all these changes to work, the java class should have serialVersionUID defined for the class.

serialVersionUID

The serialization 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.

If the receiver has loaded a class for the object that has a different serialVersionUID than that of the corresponding sender's class, then deserialization will result in an InvalidClassException.

A serializable class can declare its own serialVersionUID explicitly by declaring a field named "serialVersionUID" that must be static, final, and of type long:
ANY-ACCESS-MODIFIER static final long serialVersionUID = 42L;

If a serializable class does not explicitly declare a serialVersionUID, then the serialization runtime will calculate a default serialVersionUID value for that class based on various aspects of the class, as described in the Java(TM) Object Serialization Specification.

However, it is *strongly recommended* that all serializable classes explicitly declare serialVersionUID values, since the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected **InvalidClassExceptions** during deserialization.

Therefore, to guarantee a consistent serialVersionUID value across different java compiler implementations, a serializable class must declare an explicit serialVersionUID value.

It is also strongly advised that explicit **serialVersionUID** declarations use the **private modifier** where possible, since such declarations apply only to the immediately declaring class--serialVersionUID fields are not useful as inherited members.

Array classes cannot declare an explicit serialVersionUID, so they always have the default computed value, but the requirement for matching serialVersionUID values is waived for array classes.

SerialVersionUID inside Serializable class in Java

When you declare a class as Serializable by implementing marker interface java.io.Serializable, Java runtime persists instances of that class into disk by using default Serialization mechanism, provided you have not customized the process using Externalizable interface.

During serialization, Java runtime creates a version number for a class, so that it can de-serialize it later. This version number is known as SerialVersionUID in Java.

If during deserialization, SerialVersionUID doesn't match then process will fail with InvalidClassException as Exception in thread "main" java.io.InvalidClassException, also printing class-name and respective SerialVersionUID.
Quick solution to fix this problem is copying SerialVersionUID and declaring them as private static final long constant in your class.

As I said, when we don't declare SerialVersionUID, as a static, final and long value in our class, the Serialization mechanism creates it for us.

This mechanism is sensitive to many details including fields in your class, their access modifier, the interface they implement and even different Compiler implementations, any change in class or using a different compiler may result in different SerialVersionUID, which may eventually stop reloading serialized data. It's too risky to rely on Java Serialization mechanism for generating this id, and that's why it's recommended to declare explicit SerialVersionUID in your Serializable class.

<u>Test cases For SerialVersionUID.</u>

Address.java :-serializable class with a serialVersionUID of 1L.

```
import java.io.Serializable;


public class Address implements Serializable{


        private static final long serialVersionUID = 1L;


        String street;
        String country;
        public void setStreet(String street){
            this.street = street;
        }

        public void setCountry(String country){
            this.country = country;
        }

        public String getStreet(){
            return this.street;
        }

        public String getCountry(){
            return this.country;
        }

        @Override
        public String toString() {
        return new StringBuffer(" Street : ")
        .append(this.street)
        .append(" Country : ")
        .append(this.country).toString();
        }
}
```

WriteObject.java:- A simple class to write / serialize the Address object into a file – “c:\\address.ser”.

```
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;

public class WriteObject{

    public static void main (String args[]) {

        Address address = new Address();
        address.setStreet("wall street");
        address.setCountry("united states");

        try{

          FileOutputStream fout = new
FileOutputStream("c:\\address.ser");
          ObjectOutputStream oos = new
ObjectOutputStream(fout);
          oos.writeObject(address);
          oos.close();
          System.out.println("Done");

        }catch(Exception ex){
              ex.printStackTrace();
```

```
            }
        }
}
```

ReadObject.java:- A simple class to read / deserialize the Address object from file – “c:\\address.ser”.

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;

public class ReadObject{

   public static void main (String args[]) {

           Address address;
       
           try{

                   FileInputStream fin = new
FileInputStream("c:\\address.ser");
                   ObjectInputStream ois = new
ObjectInputStream(fin);
                   address = (Address) ois.readObject();
                   ois.close();

                   System.out.println(address);
```

```

        }catch(Exception ex){
            ex.printStackTrace();
        }
    }
}
```

Result:-

Same serialVersionUID , there is no problem during the deserialization process

```
javac Address.java

javac WriteObject.java

javac ReadObject.java

java WriteObject

java ReadObject

Street : wall street Country : united states
```

Different serialVersionUID:- In Address.java, change the serialVersionUID to 2L (it was 1L), and compile it again.

```
javac Address.java

java ReadObject

java.io.InvalidClassException: Address; local class

incompatible:

stream classdesc serialVersionUID = 1, local class

serialVersionUID = 2

        ...
        at ReadObject.main(ReadObject.java:14)
```

The “`InvalidClassException`” will raise, because you write a serialization class with serialVersionUID “1L” but try to retrieve it back with updated serialization class, serialVersionUID “2L”.

Important fact, related to Java SerialVersionUID:-

- SerialVersionUID is used to version serialized data. You can only de-serialize a class if its SerialVersionUID matches with the serialized instance.
- When we don't declare SerialVersionUID in our class, Java runtime generates it for us, but that process is sensitive to many class meta data including number of fields, type of fields, access modifier of fields, interface implemented by class etc. You can find accurate information in Serialization documentation from Oracle.
- It's recommended to declare SerialVersionUID as a private static final long variable to avoid default mechanism. Some IDE like Eclipse also display warning if you miss it e.g. "The Serializable class Customer does not declare a static final SerialVersionUID field of type long". Though you can disable these warnings by going to Window > Preferences > Java > Compiler > Errors / Warnings > Potential Programming Problems, I suggest not to do that. Only case I see being careless is when restoring data is not needed. Here is how this error looks like in Eclipse IDE, all you need to do is accept the first quick fix.
- You can even use the serialver tool from JDK to generate Serial Version for classes in Java. It also has a GUI, which can be enabled by passing -show parameter.
- It's Serialization best practice in Java to explicitly declare SerialVersionUID, to avoid any issues during deserialization especially if you are running a client server application which relies on serialized data e.g. RMI.

- Always include it as a field, for example: “private static final long serialVersionUID = 7526472295622776147L; ” include this field even in the first version of the class, as a reminder of its importance.
- Do not change the value of this field in future versions, unless you are knowingly making changes to the class which will render it incompatible with old serialized objects. If needed, follow the above given guidelines.

The ObjectStreamClass Class

The ObjectStreamClass provides information about classes that are saved in a Serialization stream.
The descriptor provides the fully-qualified name of the class and its serialization version UID. A SerialVersionUID identifies the unique original class version for which this class is capable of writing streams and from which it can read.

```
package java.io;

public class ObjectStreamClass
{
   public static ObjectStreamClass lookup(Class cl);

        public static ObjectStreamClass lookupAny(Class cl);

   public String getName();
   public Class forClass();
   public ObjectStreamField[] getFields();
   public long getSerialVersionUID();

   public String toString();
}
```

The lookup method returns the ObjectStreamClass descriptor for the specified class in the virtual machine. If the class has defined serialVersionUID it is retrieved from the class. If the serialVersionUID is not

defined by the class, it is computed from the definition of the class in the virtual machine. *I*f the specified class is not serializable or externalizable, *null* is returned.
The lookupAny method behaves like the lookup method, except that it returns the descriptor for any class, regardless of whether it implements Serializable. The serialVersionUID of a class that does not implement Serializable is *0L.*
The getName method returns the name of the class, in the same format that is used by the Class.getName method.
The forClass method returns the Class in the local virtual machine if one was found by ObjectInputStream.resolveClass method. Otherwise, it returns *null*.
The getFields method returns an array of ObjectStreamField objects that represent the serializable fields of this class.
The getSerialVersionUID method returns the serialVersionUID of this class. If not specified by the class, the value returned is a hash computed from the class's name, interfaces, methods, and fields using the Secure Hash Algorithm (SHA) as defined by the National Institute of Standards.
The toString method returns a printable representation of the class descriptor including the name of the class and the serialVersionUID.

Externalization in Java

Externalization is nothing but it is serialization by implementing an **Externalizable interface** to persist and restore the object.

To externalize your object, you need to implement an **Externalizable interface** that **extends** the **Serializable interface**.

Here only the identity of the class is written in the serialization stream and it is the responsibility of the class to save and restore the contents of its instances which means you will have complete control of what to serialize and what not to serialize.

But with serialization the identity of all the classes, its superclasses, instance variables and then the contents for these items is written to the serialization stream.

But to externalize an object, you need a default public constructor.

Unlike Serializable interface, Externalizable interface is **not a marker interface** and it provides two methods - **writeExternal** and **readExternal**.

These methods are implemented by the class to give the class a complete control over the format and contents of the stream for an object and its supertypes.
These methods must explicitly coordinate with the supertype to save its state.
These methods supersede customized implementations of **writeObject** and **readObject** methods.

Java Externalizable Interface

public interface Externalizable extends Serializable

Only the identity of the class of an Externalizable instance is written in the serialization stream and it is the responsibility of the class to save and restore the contents of its instances.
The **writeExternal** and **readExternal** methods of the Externalizable interface are implemented by a class to give the class complete control over the format and contents of the stream for an object and its supertypes.

These methods must explicitly coordinate with the supertype to save its state. These methods supersede customized implementations of writeObject and readObject methods.

Object Serialization uses the Serializable and Externalizable interfaces. Object persistence mechanisms can use them as well. Each object to be stored is tested for the Externalizable interface.

If the object supports **Externalizable**, the **writeExternal** method is **called**.

If the object does not support Externalizable and does implement Serializable, the object is saved using ObjectOutputStream.

When an Externalizable object is reconstructed, an instance is created using the public no-arg constructor, then the **readExternal** method is called.
Serializable objects are restored by reading them from an **ObjectInputStream**.

An Externalizable instance can designate a substitution object via the **writeReplace** and **readResolve** methods documented in the Serializable interface.

public void writeExternal(ObjectOutput out) throws IOException:-
The object implements the writeExternal method to save its contents by calling the methods of Data Output for its primitive values or calling the writeObject method of ObjectOutput for objects, strings, and arrays.

How does serialization happen?

JVM first checks for the Externalizable interface and if the object supports Externalizable interface, then serializes the object using the **writeExternal** method.
If the object does not support Externalizable but implements Serializable, then the object is saved using ObjectOutputStream.

Now when an Externalizable object is reconstructed, an instance is created first using the public no-arg constructor, then the **readExternal** method is called.
Again if the object does not support Externalizable, then Serializable objects are restored by reading them from an ObjectInputStream.

public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException; The object implements the readExternal method to restore its contents by calling the methods of Data Input for primitive types and readObject for objects, strings and arrays.

If you notice the serialization process, it’s done automatically.

Sometimes we want to obscure the object data to maintain its integrity. We can do this by implementing java.io.Externalizable interface and providing implementation of ***writeExternal***() and ***readExternal***() methods to be used in the serialization process.

```
package com.journaldev.externalization;

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

public class Person implements Externalizable{

    private int id;
    private String name;
    private String gender;

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeInt(id);
        out.writeObject(name+"xyz");
        out.writeObject("abc"+gender);
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException,
            ClassNotFoundException {
        id=in.readInt();
        //read in the same order as written
        name=(String) in.readObject();
        if(!name.endsWith("xyz")) throw new IOException("corrupted data");
        name=name.substring(0, name.length()-3);
        gender=(String) in.readObject();
        if(!gender.startsWith("abc")) throw new IOException("corrupted
data");
        gender=gender.substring(3);
    }

    @Override
    public String toString(){
        return "Person{id="+id+",name="+name+",gender="+gender+"}";
    }
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

}
```

Notice that I have changed the field values before converting it to Stream and then while reading reversed the changes. In this way, we can maintain data integrity of some sorts. We can throw an exception if after reading the stream data, the integrity checks fail.

Let’s write a test program to see it in action.

```
package com.journaldev.externalization;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class ExternalizationTest {

    public static void main(String[] args) {

        String fileName = "person.ser";
        Person person = new Person();
        person.setId(1);
        person.setName("Pankaj");
        person.setGender("Male");
        try {
            FileOutputStream fos = new FileOutputStream(fileName);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(person);
            oos.close();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        FileInputStream fis;
        try {
            fis = new FileInputStream(fileName);
            ObjectInputStream ois = new ObjectInputStream(fis);
            Person p = (Person)ois.readObject();
            ois.close();
            System.out.println("Person Object Read="+p);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }

    }

}
```

When we run the above program, we get the following output.

```
Person Object Read=Person{id=1,name=Pankaj,gender=Male}
```

What will happen when an externalizable class extends a non externalizable superclass?

Then in this case, you need to persist the superclass fields also in the subclass that implements Externalizable interface.

Look at this example.

```
/**
 * The superclass does not implement externalizable
 */
class Automobile  {

    /*
     * Instead of making these members private and adding setter
     * and getter methods, I am just giving default access specifiers.
     * You can make them private members and add setters and getters.
     */
    String regNo;
    String mileage;

    /*
     * A public no-arg constructor
     */
    public Automobile() {}

    Automobile(String rn, String m) {
        regNo = rn;
        mileage = m;
    }
}

public class Car extends Automobile implements Externalizable {

    String name;
    int year;

    /*
     * mandatory public no-arg constructor
     */
    public Car() { super(); }
    Car(String n, int y) {
        name = n;
        year = y;
    }

    /**
     * Mandatory writeExernal method.
     */
    public void writeExternal(ObjectOutput out) throws IOException  {
        /*
         * Since the superclass does not implement the Serializable
interface
         * we explicitly do the saving.
         */
        out.writeObject(regNo);
        out.writeObject(mileage);

        //Now the subclass fields
        out.writeObject(name);
        out.writeInt(year);
    }

    /**
     * Mandatory readExternal method.
     */
    public void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException {
        /*
         * Since the superclass does not implement the Serializable
interface
         * we explicitly do the restoring
         */
        regNo = (String) in.readObject();
        mileage = (String) in.readObject();

        //Now the subclass fields
        name = (String) in.readObject();
        year = in.readInt();
    }

    /**
     * Prints out the fields. used for testing!
     */
    public String toString() {
        return("Reg No: " + regNo + "\n" + "Mileage: " + mileage +
                    "Name: " + name + "\n" + "Year: " + year );
    }
}
```

Here the Automobile class does not implement an Externalizable interface. So to persist the fields in the automobile class the **writeExternal** and **readExternal** methods of Car class are modified to save/restore the superclass fields first and then the subclass fields.

What if the superclass implements the Externalizable interface?

Well, in this case the superclass will also have the readExternal and writeExternal methods as in Car class and will persist the respective fields in these methods.

```
import java.io.*;

/**
 * The superclass implements externalizable
 */
class Automobile implements Externalizable {

    /*
     * Instead of making thse members private and adding setter
     * and getter methods, I am just giving default access specifier.
     * You can make them private members and add setters and getters.
     */
    String regNo;
    String mileage;

    /*
     * A public no-arg constructor
     */
    public Automobile() {}

    Automobile(String rn, String m) {
        regNo = rn;
        mileage = m;
    }

    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeObject(regNo);
        out.writeObject(mileage);
    }

    public void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException {
       regNo = (String)in.readObject();
       mileage = (String)in.readObject();
    }

}

public class Car extends Automobile implements Externalizable {

    String name;
    int year;

    /*
     * mandatory public no-arg constructor
     */
    public Car() { super(); }
```

```
   Car(String n, int y) {
        name = n;
        year = y;
   }

   /**
    * Mandatory writeExernal method.
    */
   public void writeExternal(ObjectOutput out) throws IOException  {
        // first we call the writeExternal of the superclass as to write
        // all the superclass data fields
        super.writeExternal(out);

        //Now the subclass fields
        out.writeObject(name);
        out.writeInt(year);
   }

   /**
    * Mandatory readExternal method.
    */
   public void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException {
        // first call the superclass external method
        super.readExternal(in);

        //Now the subclass fields
        name = (String) in.readObject();
        year = in.readInt();
   }

   /**
    * Prints out the fields. used for testing!
    */
   public String toString() {
      return("Reg No: " + regNo + "\n" + "Mileage: " + mileage +
                  "Name: " + name + "\n" + "Year: " + year );
   }
```

```
}
```

In this example since the Automobile class stores and restores its fields in its own writeExternal and readExternal methods, you don't need to save/restore the superclass fields in subclass but if you observe closely the writeExternal and readExternal methods of Car class closely, you will find that you still need to first call the super.xxxx() methods that confirms the statement the externalizable object must also coordinate with its supertype to save and restore its state.

The difference in sizes when you serialize using Serializable interface and serialize using Externalizable interface:-

When you serialize an object using serialization, the basic serialization mechanism stores all kinds of information in the file so that it can deserialize without any other assistance.

```
Length: 220
Magic: ACED
Version: 5
  OBJECT
    CLASSDESC
    Class Name: "SimpleClass"
    Class UID:  -D56EDC726B866EBL
    Class Desc Flags: SERIALIZABLE;
    Field Count: 4
    Field type: object
    Field name: "firstName"
    Class name: "Ljava/lang/String;"
    Field type: object
    Field name: "lastName"
    Class name: "Ljava/lang/String;"
        Field type: float
    Field name: "weight"
    Field type: object
    Field name: "location"
    Class name: "Ljava/awt/Point;"
    Annotation: ENDBLOCKDATA
    Superclass description: NULL
   STRING: "Brad"
   STRING: "Pitt"
   float: 180.5
   OBJECT
     CLASSDESC
     Class Name: "java.awt.Point"
     Class UID:  -654B758DCB8137DAL
     Class Desc Flags: SERIALIZABLE;
     Field Count: 2
     Field type: integer
     Field name: "x"
     Field type: integer
     Field name: "y"
     Annotation: ENDBLOCKDATA
     Superclass description: NULL
    integer: 49.345
    integer: 67.567
```

Now if you serialize the same by extending the Externalizable interface, the size will be reduced drastically and the information saved in the persistent store is also reduced a lot. Here is the result of serializing the same class, modified to be externalizable.

Notice that the actual data is not parseable externally any more--only your class knows the meaning of the data!

```
Length: 54
Magic: ACED
```

```
Version: 5
 OBJECT
  CLASSDESC
  Class Name: "SimpleClass"
  Class UID:  5CB3777417A3AB5BL
  Class Desc Flags: EXTERNALIZABLE;
  Field Count: 0
  Annotation
   ENDBLOCKDATA
  Superclass description
   NULL
 EXTERNALIZABLE:
  [70 00 04 4D 61 72 6B 00 05 44 61 76 69 73 43 3C
  80 00 00 00 00 01 00 00 00 01]
```

Limitations at Serialization with externalization.

Externalization efficiency comes at a price.

The default serialization mechanism adapts to application changes due to the fact that metadata is automatically extracted from the class definitions (*observe the format above and you will see that when the object is serialized by implementing Serializable interface, the class metadata(definitions) are written to the persistent store while when you serialize by implementing Externalizable interface, the class metadata is not written to the persistent store*).

Externalization on the other hand isn't very flexible and requires you to rewrite your marshaling and unmarshaling code whenever you change your class definitions.

As you know a default public **no-arg constructor** will be called when serializing the objects that implement the Externalizable interface.

Hence, Externalizable interface can't be implemented by Inner Classes in Java as all the constructors of an inner class in Java will always accept the instance of the enclosing class as a prepended parameter and therefore you can't have a no-arg constructor for an inner class.

Inner classes can achieve object serialization by only implementing a Serializable interface.

If you are subclassing your externalizable class, you have to invoke your superclass's implementation. So this causes overhead while you subclass your externalizable class.

Observe the examples above where the superclass writeExternal method is explicitly called in the subclass writeExternal method.

Methods in externalizable interfaces are public. So any malicious program can invoke which results into losing the prior serialized state.

Once your class is tagged with either Serializable or Externalizable, you can't change any evolved version of your class to the other format.

You alone are responsible for maintaining compatibility across versions. That means that if you want the flexibility to add fields in the future, you'd better have your own mechanism so that you can skip over additional information possibly added by those future versions.

When serialization by implementing a Serializable interface is serving your purpose, why should you go for externalization?

Answer:- Serializing by implementing Serializable interface has some issues.
Let's see one by one what they are.

1. Serialization is a recursive algorithm. What I mean to say here is, apart from the fields that are required, starting from a single object, until all the objects that can be reached from that object by following instance variables, are also serialized. This includes the superclass of the object until it reaches the "Object" class and the same way the superclass of the instance variables until it reaches the "Object" class of those variables. Basically all the objects that it can read. This leads to a lot of overheads. Say for example, you need only car type and license number but using serialization, you cannot stop there. All the information that includes description of the car, its parts, blah blah will be serialized. Obviously this slows down the performance.
2. Both serializing and deserializing require the serialization mechanism to discover information about the instance it is serializing. Using the default serialization mechanism, will use reflection to discover all the field values. Also the information about class description is added to the stream which includes the description of all the serializable superclasses, the description of the class and the instance data associated with the specific instance of the class. Lots of data and metadata and again performance issues.
3. You know that serialization needs serialVersionUID, a unique Id to identify the information persisted. If you don't explicitly set a serialVersionUID, serialization will compute the serialVersionUID by going through all the fields and methods. So based on the size of the class, again the serialization mechanism takes a respective amount of time to calculate the value. A third performance issue. Above three points confirm serialization has performance issues. Apart from performance issues,
4. When an object that implements Serializable interface, is serialized or deserialized, no constructor of the object is called and hence any initialization which is done in the constructor cannot be done.

Although there is an alternative of writing all initialization logic in a separate method and calling it in constructor and readObject methods so that when an object is created or deserialized, the initialization process can happen but it definitely is a messy approach.

The solution for all the above issues is ***Externalization***.

Tips for serialization

You can decide whether to implement Externalizable or Serializable on a class-by-class basis. Within the same application, some of your classes can be Serializable, and some can be Externalizable. This makes it easy to evolve your application in response to actual performance data and shifting requirements.
You can do the following thing:

- Make all your classes implement Serializable.
- Then make some of them, the ones you send often and for which serialization is inefficient, implement Externalizable instead.

To reduce memory size:-

- Write primitives or Strings directly. For example, instead of writing out a contained object, Point (in SimpleClass, we have a field of type Point), write out each of its integer coordinates separately. When you read them in, create a new Point from the two integers. This can be very significant in terms of size: an array of three Points takes 117 bytes; an array of 6 ints takes 51 bytes.
- Strings are special-cased and don't carry much of the object overhead; you will normally use them as is. However, the serialized representation of a String is UTF, which works great for ASCII characters, is neutral for most European characters, but causes a 50% increase in size for Japanese and other scripts. If you have significant strings of Asian text you better serialize a char array instead.

Read Object from File.

Above, we learned how to write an Object to File in java, here we will read the same file to create an object in java.

This process is called deserialization and we use **java.io.ObjectInputStream** to read Objects from files in java.

```
package com.journaldev.files;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class FileToObject {

    public static void main(String[] args) {
        try {
            FileInputStream is = new FileInputStream("EmployeeObject.ser");
            ObjectInputStream ois = new ObjectInputStream(is);
            EmployeeObject emp = (EmployeeObject) ois.readObject();
            ois.close();
            is.close();
            System.out.println(emp.toString());
        } catch (ClassNotFoundException | IOException e) {
            e.printStackTrace();
        }
    }

}
```

Output of the above program is:

```
Employee:: Name=Pankaj Age=10 Gender=Male Role=null
```

Java Transient Keyword

Java transient keyword is used in serialization. **If you define any data member as transient, it will not be serialized.**

Example:- I have declared a class as Student, it has three data members id, name and age. If you serialize the object, all the values will be serialized but I don't want to serialize one value, e.g. age then we can declare the age data member as transient.

Let's create a class with a transient variable.

```
import java.io.Serializable;
public class Student implements Serializable{
 int id;
 String name;
 transient int age;//Now it will not be serialized
 public Student(int id, String name,int age) {
  this.id = id;
  this.name = name;
  this.age=age;
 }
}
```

Now write the code to serialize the object.

```
import java.io.*;
class PersistExample{
 public static void main(String args[])throws Exception{
  Student s1 =new Student(211,"ravi",22);//creating object
  //writing object into file
  FileOutputStream f=new FileOutputStream("f.txt");
  ObjectOutputStream out=new ObjectOutputStream(f);
  out.writeObject(s1);
  out.flush();
  out.close();
  f.close();
  System.out.println("success");
 }
}
```

Output: success

Now write the code for deserialization.

```
import java.io.*;
class DePersist{
 public static void main(String args[])throws Exception{
  ObjectInputStream in=new ObjectInputStream(new
FileInputStream("f.txt"));
  Student s=(Student)in.readObject();
  System.out.println(s.id+" "+s.name+" "+s.age);
  in.close();
 }
}
```

Output: 211 ravi 0

As you can see, the printing age of the student returns 0 because the value of age was not serialized.

Deserialization

Deserialization is the process by which the object previously serialized is reconstructed back into its original form i.e. object instance. The input to the deserialization process is the stream of bytes which we get over the other end of the network OR we simply read it from file system/database.

What is written inside this stream of bytes?
To be very precise, this stream of bytes (or say serialized data) has all the information about the instance which was serialized by serialization process. This information includes **class's metadata**, **type information** of **instance fields** and **values of instance fields** as well.

This same information is needed when an object is reconstructed back to a new object instance.
While deserializing an object, the JVM reads its class **metadata** from the stream of bytes which specifies whether the class of an object implements either ‘Serializable’ or ‘Externalizable’ interface.

How does the JVM create the object without calling the constructor?
If an instance implements the serializable interface, then an instance of the class is created without invoking its constructor.

Let’s look at the bytecode of a simple program:-

```
public class SimpleProgram
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
    }
}

Byte code:

public class SimpleProgram extends java.lang.Object{
public SimpleProgram();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   getstatic   #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   3:   ldc #3; //String Hello World!
   5:   invokevirtual   #4; //Method
java/io/PrintStream.println:(Ljava/lang/String;)V
```

```
   8:   return
}
```

Descriptions:- In our first line at byteCode, we're going to push a value from the "local variable table" onto the stack.
In this case, we're really only pushing the implicit reference to "**this**" so it isn't the most exciting instruction. Second instruction is the main thing. **It actually invokes the constructor of super most class and in the above case it is Object.java.**
And once the constructor of super most class (i.e. Object in this case) has been called, rest of the code does specific instructions written in code.

Matching to above concept i.e. constructor of super most class, we have similar concept in deserialization.
In the deserialization process, **it is required that all the parent classes of instance should be Serializable; and if any superclass in hierarchy is not Serializable then it must have a default constructor.**

Now it makes sense. So, while deserializing the super most classes are searched first until any non-serializable class is found. If all super classes are serializable then JVM ends up reaching the Object class itself and creating an instance of the Object class first.
If in between searching the super classes, any class is found non-serializable then its default constructor will be used to allocate an instance in memory.

So till now we got the instance located in memory using one of superclass's default constructor.
Note that after this no constructor will be called for any class. After executing super class constructor, JVM reads the byte stream and uses the instance's metadata to set type information and other meta information of the instance.

After the blank instance is created, JVM first sets its static fields and then invokes the default **readObject**() method [if it’s not overridden, otherwise overridden method will be called] internally which is responsible for setting the values from byte stream to blank instance.

After the **readObject**() method is completed, the deserialization process is done and you are ready to work with a new deserialized instance.

Note:- If any superclass of instance to be deserialized in non-serializable and also does not have a default constructor then the **‘NotSerializableException‘** is thrown by JVM.

Also, before continuing with the object reconstruction, the JVM checks to see if the serialVersionUID mentioned in the byte stream matches the serialVersionUID of the class of that object. If it does not match then the ‘InvalidClassException‘ is thrown.

How to make a Java class Serializable?

Making a class Serializable in Java is very easy. Your Java class just needs to implement the java.io.Serializable **interface** and JVM will take care of serializing objects in default format.

Decision to making a Class Serializable should be taken concisely because though near term cost of making a Class Serializable is low, long term cost is substantial and it can potentially limit your ability to further modify and change its implementation because like any public API, serialized form of an object becomes part of public API and when you change structure of your class by implementing addition interface, adding or removing any field can potentially break default serialization, this can be minimized by using a

custom binary format but still requires lot of effort to ensure backward compatibility.
One example of How Serialization can put constraints on your ability to change class is SerialVersionUID.
If you don't explicitly declare SerialVersionUID then JVM generates it based upon the structure of the class which depends upon interfaces a class implements and several other factors which are subject to change.
Suppose you implement another interface than JVM will generate a different SerialVersionUID for new versions of class files and when you try to load an old object serialized by an old version of your program you will get **InvalidClassException**.

What is the difference between Serializable and Externalizable interfaces in Java?

Externalizable provides us with the **writeExternal**() and **readExternal**() method which gives us flexibility to control java serialization mechanism instead of relying on Java's default serialization.
Correct implementation of Externalizable interface can improve application performance drastically.

How many methods does Serializable have? If there is no method then what is the purpose of a Serializable interface?

Serializable interface exists in java.io package and forms core of java serialization mechanism. It doesn't have any method and is also called Marker Interface in Java.
When your class implements **java.io.Serializable interface** it becomes Serializable in Java and gives the compiler an indication that it uses Java Serialization mechanism to serialize this object.

What is serialVersionUID? What would happen if you don't define this?

SerialVersionUID is an ID which is stamped on an object when it gets serialized, usually the hashcode of the object, you can use tool serialver to see serialVersionUID of a serialized object .

SerialVersionUID is used for version control of objects. you can specify serialVersionUID in your class file also.

Consequence of not specifying serialVersionUID is that when you add or modify any field in class then the already serialized class will not be able to recover because serialVersionUID generated for new class and for old serialized objects will be different. Java serialization process relies on correct serialVersionUID for recovering state of serialized object and throws java.io.InvalidClassException in case of serialVersionUID mismatch.

While serializing you want some of the members not to serialize? How do you achieve it?

This is sometimes also asked as what is the use of transient variables, does transient and static variables get serialized or not etc.

So if you don't want any field to be part of an object's state then declare it either static or transient based on your need and it will not be included during the Java serialization process.

What will happen if one of the members in the class doesn't implement a Serializable interface?

One of the easy questions about the Serialization process in Java.

If you try to serialize an object of a class which implements Serializable, but the object **includes a reference to an non- Serializable class** then a **‘NotSerializableException’** will be thrown **at runtime.**

This is why I always put a Serializable Alert (comment section in my code) , one of the code comment best practices, to instruct developers to remember this fact while adding a new field in a Serializable class.

If a class is Serializable but its superclass is not, what will be the state of the instance variables inherited from superclass after deserialization?

Java serialization process only continues in object hierarchy till the class is Serializable i.e. implements Serializable interface in Java and values of the instance variables inherited from superclass will be initialized by calling constructor of Non-Serializable Superclass during deserialization process.

Once the constructor chaining starts it wouldn't be possible to stop that , hence even if classes higher in hierarchy implements Serializable interface , their constructor will be executed.

Can you Customize Serialization process or can you override the default Serialization process in Java?

The answer is yes you can. We all know that for serializing an object **ObjectOutputStream.writeObject** (saveThisobject) is invoked and for reading object **ObjectInputStream.readObject**() is invoked but there is one more thing which Java Virtual Machine provides you is to define these two method in your class.
If you define these two methods in your class then JVM will invoke these two methods instead of applying the default serialization mechanism.

You can customize behavior of object serialization and deserialization here by doing any kind of pre or post processing task.

Important point to note is making **these methods private** to avoid being inherited, overridden or overloaded.

Since only Java Virtual Machine can call private methods, the integrity of your class will remain and Java Serialization will work as normal.

Suppose superclass of a new class implements Serializable interface, how can you avoid a new class from being serialized?

If Superclass of a Class already implements Serializable interface in Java then it is already Serializable in Java, since you can not un-implement an interface it's not really possible to make it Non Serializable class but yes there is a way to avoid serialization of new class.

To avoid Java serialization you need to implement **writeObject**() and **readObject**() methods in your Class and need to throw **NotSerializableException** from those methods.

Which methods are used during the Serialization and DeSerialization process in Java?

This is a very common interview question in Serialization. Basically the interviewer is trying to know; Whether you are familiar with usage of **readObject**(), **writeObject**(), **readExternal**() and **writeExternal**() or not.

Java Serialization is done by java.io.ObjectOutputStream class. That class is a filter stream which is wrapped around a lower-level byte stream to handle the serialization mechanism. To store any object via serialization

mechanism we call **ObjectOutputStream.writeObject(saveThisobject)** and to deserialize that object we call **ObjectInputStream.readObject()** method.
Call to **writeObject()** method triggers serialization process in java. one important thing to note about the **readObject()** method is that it is used to read bytes from the persistence and to create objects from those bytes and it returns an Object which needs to be type cast to correct type.

Suppose you have a class which you serialized and stored in persistence and later modified that class to add a new field. What will happen if you deserialize the object already serialized?

It depends on whether the class has its own serialVersionUID or not. As we know, if we don't provide serialVersionUID in our code java compiler will generate it and normally it’s equal to hashCode of object.

By adding any new field there is chance that new serialVersionUID generated for that class version is not the same of already serialized object and in this case Java Serialization API will **throw java.io.InvalidClassException** and this is the reason it's recommended to have your own serialVersionUID in code and make sure to keep it same always for a single class.

What are the compatible changes and incompatible changes in Java Serialization Mechanism?

The real challenge lies with change in class structure by adding any field, method or removing any field or method is that with an already serialized object.
As per Java Serialization specification adding any field or method comes under compatible change and changing class hierarchy or

UN-implementing Serializable interfaces some under non compatible changes.

Can we transfer a Serialized object via network?

Yes you can transfer a Serialized object via network because Java serialized object remains in the form of bytes which can be transmitted via network. You can also store serialized objects in Disk or database as Blob.

Which kind of variables are not serialized during Java Serialization?

This question is sometimes asked differently but the purpose is the same whether Java developers know specifics about static and transient variables or not.
Since static variables belong to the class and not to an object, they are not part of the state of the object so they are not saved during the Java Serialization process.
As Java Serialization only persists the state of object and not object itself.

Transient variables are also not included in the Java serialization process and are not part of the object's serialized state.

After this question, sometimes the interviewer asks a follow-up question: if you don't store values of these variables then what would be the value of these variables once you deserialize and recreate those objects?

Some More On Serialization.

We all know what Serializable interface guarantees i.e. ability to serialize the classes. This interface recommends you to use serialVersionUID also.

Now, even if you use both in your classes, do you know what can break your design even now??

Let's identify the future changes in your class which will be compatible and others which will prove incompatible.

Incompatible Changes- Incompatible changes to classes are those changes for which the guarantee of interoperability cannot be maintained.

The incompatible changes that may occur while evolving a class are (considering default serialization or deserialization):-

1. **Deleting fields --** If a field is deleted in a class, the stream written will not contain its value. When the stream is read by an earlier class, the value of the field will be set to the default value because no value is available in the stream. However, this default value may adversely impair the ability of the earlier version to fulfill its contract.
2. **Moving classes up or down the hierarchy --** This cannot be allowed since the data in the stream appears in the wrong sequence.
3. **Changing a non-static field to static or a non-transient field to transient --** When relying on default serialization, this change is equivalent to deleting a field from the class. This version of the class will not write that data to the stream, so it will not be available to be read by earlier versions of the class. As when deleting a field, the field of the earlier version will be initialized to the default value, which can cause the class to fail in unexpected ways.
4. **Changing the declared type of a primitive field --** Each version of the class writes the data with its declared type. Earlier versions of the class attempting to read the field will fail because the type of the data in the stream does not match the type of the field.
5. Changing the writeObject or readObject method so that it no longer writes or reads the default field data or changing it so that it attempts to write it or read it when the previous version did not. The default field data must consistently either appear or not appear in the stream.

6. Changing a class from Serializable to Externalizable or vice-versa is an incompatible change since the stream will contain data that is incompatible with the implementation of the available class.
7. Changing a class from a non-enum type to an enum type or vice versa since the stream will contain data that is incompatible with the implementation of the available class.
8. Removing either Serializable or Externalizable is an incompatible change since when written it will no longer supply the fields needed by older versions of the class.
9. Adding the writeReplace or readResolve method to a class is incompatible if the behavior would produce an object that is incompatible with any older version of the class.

Compatible Changes-

1. Adding fields -- When the class being reconstituted has a field that does not occur in the stream, that field in the object will be initialized to the default value for its type. If class-specific initialization is needed, the class may provide a readObject method that can initialize the field to non default values.
2. Adding classes -- The stream will contain the type hierarchy of each object in the stream. Comparing this hierarchy in the stream with the current class can detect additional classes. Since there is no information in the stream from which to initialize the object, the class's fields will be initialized to the default values.
3. Removing classes -- Comparing the class hierarchy in the stream with that of the current class can detect that a class has been deleted. In this case, the fields and objects corresponding to that class are read from the stream. Primitive fields are discarded, but the objects referenced by the deleted class are created, since they may be referred to later in the stream. They will be garbage-collected when the stream is garbage-collected or reset.
4. Adding writeObject/readObject methods -- If the version reading the stream has these methods then readObject is expected, as usual, to

read the required data written to the stream by the default serialization. It should call defaultReadObject first before reading any optional data. The writeObject method is expected as usual to call defaultWriteObject to write the required data and then may write optional data.

5. Removing writeObject/readObject methods -- If the class reading the stream does not have these methods, the required data will be read by default serialization, and the optional data will be discarded.
6. Adding java.io.Serializable -- This is equivalent to adding types. There will be no values in the stream for this class so its fields will be initialized to default values. The support for subclassing nonserializable classes requires that the class's super type have a no-arg constructor and the class itself will be initialized to default values. If the no-arg constructor is not available, the InvalidClassException is thrown.
7. Changing the access to a field -- The access modifiers public, package, protected, and private have no effect on the ability of serialization to assign values to the fields.
8. Changing a field from static to non-static or transient to non transient -- When relying on default serialization to compute the serializable fields, this change is equivalent to adding a field to the class. The new field will be written to the stream but earlier classes will ignore the value since serialization will not assign values to static or transient fields.

Recommendations for readObject and writeObject.

1. Deserialization must be treated as any constructor : validate the object state at the end of deserializing -- this implies that readObject should almost always be implemented in Serializable classes, such that this validation is performed.
2. If constructors make defensive copies for mutable object fields, so must readObject.

Other points to keep remember.

1. Use javadoc @serial tag to denote Serializable fields.
2. The .ser extension is conventionally used for files representing serialized objects.
3. No static or transient fields undergo default serialization.
4. Extendable classes should not be Serializable, unless necessary.
5. Inner classes should rarely, if ever, implement Serializable.
6. Container classes should usually follow the style of Hashtable, which implements Serializable by storing keys and values, as opposed to a large hash table data structure.

How to do deep cloning using in memory serialization in java?

Answer:- We all know that the easiest way of deep cloning (with some performance overhead) is Serialization.

It involves serializing the object to bytes and from bytes to object again. I will suggest you to use in memory deep cloning whenever it is the only need and you don’t need to persist the object for future use. In this post, I will suggest one mechanism of in memory deep cloning for reference.

In the demonstration program, I have created a demo class named SerializableClass. This has three variables i.e. first name, last name and permissions. I will add a deepCopy() instance level method to this class. Whenever invoked on an instanceof SerializableClass, it will return an exact clone/ copy of that instance.

For deep cloning, we have to first serialize and then deserialize.

For serialization, I have used ByteArrayOutputStream and ObjectOutputStream.

For deserialization, I have used ByteArrayInputStream and ObjectInputStream.

```
package serializationTest;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class SerializableClass implements Serializable{

    private static final long serialVersionUID = 1L;

    private String firstName = null;
    private String lastName = null;

    @SuppressWarnings("serial")
    private List permissions = new ArrayList()
                                    {
                                        {
                                            add("ADMIN");
                                            add("USER");
                                        }
                                    };

    public SerializableClass(final String fName, final String lName)
    {
        //validateNameParts(fName);
        //validateNameParts(lName);
        this.firstName = fName;
        this.lastName = lName;
    }

    public SerializableClass deepCopy() throws Exception
```

```
   {
      //Serialization of object
      ByteArrayOutputStream bos = new ByteArrayOutputStream();
      ObjectOutputStream out = new ObjectOutputStream(bos);
      out.writeObject(this);

      //De-serialization of object
      ByteArrayInputStream bis = new
ByteArrayInputStream(bos.toByteArray());
      ObjectInputStream in = new ObjectInputStream(bis);
      SerializableClass copied = (SerializableClass) in.readObject();

      //Verify that object is not corrupt

      //validateNameParts(fName);
      //validateNameParts(lName);

      return copied;
   }

   public String getFirstName() {
      return firstName;
   }

   public void setFirstName(String firstName) {
      this.firstName = firstName;
   }

   public String getLastName() {
      return lastName;
   }

   public void setLastName(String lastName) {
      this.lastName = lastName;
   }

   @Override
   public String toString() {
        return new StringBuilder().append(getFirstName()+",")
                            .append(getLastName()+",")
                            .append(permissions)
                            .toString();
    }
}
```

```
package serializationTest;

public class ImMemoryTest {

    public static void main(String[] args) throws Exception {
        //Create instance of serializable object
        SerializableClass myClass = new
SerializableClass("Lokesh","Gupta");
        //Verify the content
        System.out.println(myClass);

        //Now create a deep copy of it
        SerializableClass deepCopiedInstance = myClass.deepCopy();
       //Again verify the content
        System.out.println(deepCopiedInstance);
    }
}
Output:

Lokesh,Gupta,[ADMIN, USER]
Lokesh,Gupta,[ADMIN, USER]
```

How not to serialize some fields in a serializable object?

Answer:- Sometimes you don't want to serialize/store all the fields in the object. Say some fields you want to hide to preserve the privacy or some fields you may want to read only from master data, then you don't serialize them. To do this, you just need to declare a field as a transient field.

Also the static fields are not serialized. Actually there is no point in serialization static fields as static fields do not represent object state but represent class state and it can be modified by any other object.

Let's assume that you have serialized a static field and its value and before deserialization of the object, the static field value is changed by some other object. Now the static field value that is serialized/stored is no more valid. Hence it make no point in serializing the static field.

Apart from declaring the field as transient, there is another tricky way of controlling what fields can be serialized and what fields cannot be.

This is by overriding the **writeObject()** method while serialization and inside this method you are responsible for writing out the appropriate fields. When you do this, you may have to override the **readObject()** method as well.

This sounds similar to using **Externalizable** where you will write **writeExternal()** and **readExternal()** methods but anyways let's not take this route as this is not a neat route.

Why should certain objects not be serialized?

Answer:- As you know, the Object class does not implement a Serializable interface and hence any object by default is not serializable.
To make an object serializable, the respective class should explicitly implement a Serializable interface.

However certain system classes defined by java like "Thread", "OutputStream", "Socket" are not serializable. Why so?

Let's take a step back - now what is the use of serializing the Thread running in System1 JVM using System1 memory and then deserializing it in System2 and trying to run in System2 JVM. Makes no sense right! Hence these classes are not serializable.

What if you want to serialize an object containing an instance of Thread?

Answer:- Simple. Declare the Thread instance as transient.

```
public class SerialClass implements Serializable, Runnable {
   transient private Thread myThread;

   public PersistentAnimation(int animationSpeed) {
               ...
               ...
   }
}
```

Performance Issues/Improvement with Serialization:-

The default way of implementing the serialization (by implementing the Serializable interface) has performance glitches.

Say you have to write an object 10000 times in a flat file through serialization, this will take much more (almost double) the time than it takes to write the same object 10000 times to console.

To overcome this issue, it's always better to write your custom protocol instead of going for the default option.

Also always note to close the streams (object output and input streams). The object references are cached in the output stream and if the stream is not closed, the system may not garbage collect the objects written to a stream and this will result in poor performance.

Does Serialization always have performance issues?
Nope... Let me give you a situation where it is used for performance improvement.
Let's assume you have an application that should display a map and pointing to different areas in the map should highlight those areas in different colors.
Since all these are images, when you point to each location, loading an image each time will slow the application heavily.

To resolve this issue, serialization can be used.
So here since the images won't change, you can serialize the image object and the respective points on the map (x and y coordinates) and deserialize it as and when necessary. This improves the performance greatly.

What happens to inner classes? We forgot all about it.

Answer:-Yes, you can serialize inner classes by implementing the Serializable interface but it has some problems.

Inner classes (declared in a non-static context) will always contain implicit references to their enclosing classes and these references are always non-transient.

So, during the object serialization process of inner classes, the enclosing classes will also be serialized.

Now the problem is that the synthetic fields generated by Java compilers to implement inner classes are pretty much implementation dependent and hence we may face compatibility issues while deserializing on a different platform having a .class file generated by a different Java compiler.

The default serialVersionUID may also be different in such cases.

Not only this, the names assigned to the local and anonymous inner classes are also implementation dependent.

Thus, we see that object serialization of inner classes may pose some unavoidable compatibility issues and hence the serialization of inner classes is strongly discouraged.

Serialization of Enum Constants

Enum constants are serialized differently than ordinary serializable or externalizable objects. The serialized form of an enum constant consists solely of its name; field values of the constant are not present in the form.

To serialize an enum constant, **ObjectOutputStream** writes the value returned by the enum constant's name method.

To deserialize an enum constant, **ObjectInputStream** reads the constant name from the stream; the deserialized constant is then obtained by calling the **java.lang.Enum.valueOf method**, passing the constant's enum type along with the received constant name as arguments.

Like other serializable or externalizable objects, enum constants can function as the targets of back references appearing subsequently in the serialization stream.

The process by which enum constants are serialized cannot be customized: any class-specific writeObject, readObject, readObjectNoData,

writeReplace, and readResolve methods defined by enum types are ignored during serialization and deserialization.
Similarly, any serialPersistentFields or serialVersionUID field declarations are also ignored--all enum types have a fixed serialVersionUID of 0L.

Documenting serializable fields and data for enum types is unnecessary, since there is no variation in the type of data sent.