**Serialization (Java 1.1)**

Saving state of an object to a file is **Serialization**. (Process of converting an object from Java supported form to file supported form or network supported form is **Serialization** in Heap memory).

Process of reading state of an object to a file is **Deserialization.** (Process of converting an object from file supported form or network supported form to from Java supported form is **Deserialization)**.

**How to Serialize**

1. Create **FileOutputStream** (Writes binary data to file).
2. Create an **ObjectOutputStream** to write the above **FileOutputStream** (Converts object to binary data).
3. Then the object can be written to a file.

**How to Deserialize**

1. Create **FileInputStream** (File data to binary data).
2. Create **ObjectInputStream** (Binary data to Object).
3. Then Object is obtained.

**Packages required**

1. Java.IO

**Ex**

class Dog **implements Serializable** {

int i = 10;

int j = 20;

}

class SerialiseDemo{

public static void main(String[] args) **throws Exception**{

//Serialisation

Dog d1 = new Dog();

FileOutputStream fos = new FileOutputStream("file.ser");

//Java can take any file extension. (anyFile.anyExtension)

ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeObject(d1);

//DeSerialisation

FileInputStream fis = new FileInputStream("file.ser");

ObjectInputStream ois = new ObjectInputStream(fis);

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

System.out.println(d2.i +” ”+ d2.j); }

}

**Note:**

1. If Dog class does not implement Serializable, there won’t be any compilation error. But it will throw NotSerializableException.
2. main () must throw or handle the required checked exception.
3. Serializable interface is a Marker interface (does not have ant methods).
4. An object can be Serializable if it implements Serializable interface.

**Transient**

1. This modifier is applicable only for **Variables.**
2. If any data that should not be stored to meet security constraints, that variable should be made transient.
3. At the time of Serialization JVM will check if any variable is **Transient**. If there are any, JVM will ignore the original value of transient variable and will save the default value.

Ex.

class Dog **implements Serializable** {

int i = 10;

transient int j = 20;

}

Stored value: i = 10, j = 0.

1. **Static vs Transient:**

If **transient** variables are made **static**, they belong to class and hence there will not be Serialization. It is useless to make it transient

Ex. class Dog implements Serializable {

int i = 10;

transient static int j = 20;

}

Stored value i = **10**; j = **20;**

1. **Final vs transient**

All final variables are replaced by its **value** in runtime. Since variables are replaced by values, there will be no effect of transient keyword on these variables.

Ex.

class Dog **implements Serializable** {

transient final int i = 10;

int j = 20;

}

Stored value: i = 10, j = 20.

|  |  |
| --- | --- |
| Input | Output |
| int i = 10;  int j = 20; | 10..20 |
| transient int i = 10;  int j = 20; | 0…20 |
| transient static int i = 10;  transient int j = 20; | 10…20 |
| transient int I = 10;  transient final int j = 20; | 0…20 |

**Serializing Multiple Objects**

1. Any number of objects can be Serialized into a file but order of De-Serialization should be same as Serialization
2. If order of Serialization is unknown then **instanceOf** is used.

**Ex**

FileInputStream fis = new FileInputStream(“file.ser”);

ObjectInputStream ois = new ObjectInputStream(fis);

Object obj = ois.readObject();

if (obj instanceOf Dog){

Dog d1 = (Dog) obj;

}

if (obj instanceOf Cat){

Cat c1 = (Cat) obj; }

**Object Graph**

When Serializing an object, set of all objects which are reachable from that object will be serialized automatically.

This is **Object Graph**.

To run successfully, all underlying objects should implement **Serializable**

class Dog **implements Serializable** {

Cat c = new Cat();

}

class Cat **implements Serializable** {

Rat r = new Rat();

}

class Rat **implements Serializable** {

int j = 20;

}

**CUSTOMIZED SERIALIZATION**

In default Serialization, there is chance of Data loss due to transient variable. To recover this Customized Serialization is required

class Account **implements Serializable** {

String username = “name”;

transient String *pwd* = “password”;

}

Class CustomSerializationDemo{

p s v main(String[] args){

Account acc = new Account();

//Serialize

//DeSerialize

System.out.println(acc.username + “ … “+ acc.*pwd*);

}

}

Output: name … null

**Implementation of Customised Serialization**

1. During Serialization - private void writeObject(ObjectOutputStream oos) throws Exception.
2. During De-Serialization - private void readObject(ObjectInputStream ois) throws Exception.
3. These 2 are **call-back** method ie JVM automatically calls them.
4. Use the same syntax because JVM identifies the same syntax.
5. **The above 2 methods should be in the Object which is being Serialized.**

class Account implements Serializable{

String userName = "User";

transient String password = "pass";

private void writeObject(ObjectOutputStream oos) throws Exception{

oos.defaultWriteObject(); // Default Serialization

String encryptedPwd = encrypt(password); // Prepare encrypted password

oos.writeObject(encryptedPwd); // write the encrypted password

}

private void readObject(ObjectInputStream ois) throws Exception{

ois.defaltReadObject();

String encryptedPwd = (String) ois.readObject(); // Read encrypted password

password = decrypt(encryptedPwd); // Decrypt the password and assign it to the Class variable

}

}

class CustomisedSerialization{

public static void main(String[] args){

Account acc = new Account();

//Serialization

FileOutputStream fos = new FileOutputStream("file.ser");

ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeObject(acc);

//Deserialization

FileInputStream fis = new FileInputStream("file.ser");

ObjectInputStream ois = new ObjectInputStream(fis);

ois.readObject(acc);

}

}

**SERIALIZATION AND INHERITANCE**

1. Parent implements Serializable and the Child **does not:**

class Animal implements Serializable{

int i = 10;

}

class Dog extends Animal{

int j = 20

}

public class SerializationDemo{

public static void main(String[] args){

Dog d1 = new Dog();

FOS fos = new FOS ("file.ser");

OOS oos = new OOS(fos);

oos.writeObject(d1);

FIS fis = new FIS("file.ser");

OIS ois = new OIS(fis);

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

S.o.p(d2.i + "..." + d2.j);

}

}

Output – 10…20

If Parent implements Serializable, automatically Child implements Serializable (no need to mention explicitly). Hence Dog **can** be serialized.

**Note:**

* Object Class does not implement Serializable.
* GenericServlet implements Serializable.

1. Parent does not implement Serializable but Child implements

class Animal {

int i = 10;

Animal(){

S.o.p(“Animal Constructor”);}

}

class Dog extends Animal implements Serializable {

int j = 20;

Dog(){

S.o.p(“Dog Constructor”)

}

}

public class SerializationDemo{

public static void main(String[] args){

Dog d1 = new Dog();

d1.i = 888;

d1.j = 999;

FOS fos = new FOS ("file.ser");

OOS oos = new OOS(fos);

oos.writeObject(d1);

FIS fis = new FIS("file.ser");

OIS ois = new OIS(fis);

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

S.o.p(d2.i + "..." + d2.j);

}

}

Output:

1. Animal Constructor
2. Dog Constructor
3. Animal Constructor
4. 10…999

Flow:

1. Parent constructor is called (Animal).
2. Then Child constructor is called (Dog).
3. Variable i and j are updated to 888 and 999.
4. When oos.writeObject() (ie Serialization), JVM will check if there are any instance variables inheriting from non-Serializable parent or not.
5. If there are any, JVM ignores original values and saves default value into the file. Hence i = 0 and j = 999 stored in file.
6. During de-serialization, JVM checks if there are any non-serializable parent. If there are any, JVM will execute **Instance Control Flow** in every non Serializable Parent. This will assign i=10 and no argument constructor will be executed in Parent,
7. If there is no “No Argument” constructor, we get InvalidClassException.

**Instance Control Flow**

1. Identification of Instance member
2. Execution of instance variable assignments and instance blocks
3. Execution of constructors.

**EXTERNALIZATION (Java 1.1)**

In Serialization, complete Object must be saved. There is no choice of saving partial objects. If there are 2000 variables and only 2 must be serialized, there are no freedom to do this. Hence decreasing the performance of the system. Also, everything is takes care by JVM and programmer does not have any control.

In externalization, programmer has control over the persistence. Partial object can be saved which increases the performance of the system.

Externalizable is Child interface of **Serializable.**

class Animal implements Externalizable { /\* ... \*/ }

**Methods in Externalizable**

1. writeExternal()
2. readExternal()
3. writeExternal():
   1. During Serialization
   2. To write required properties to the file
4. readExternal () :
   1. During deserialization
   2. To read required properties from the file.
   3. During deserialization, file does not contain complete object. Hence JVM will createthe object and call readExternal().The object is created using public no-argument constructor. So, there should be no-argument constructor should be present.
   4. If there is no “no-arg constructor” **InvalidClassException** is seen at **runtime.**
   5. Transient keyword will not play any role or it is not required in Externalizable

class ExternalizableDemo implements Externalizable {

String s;

int i;

int j;

public ExternalizableDemo(){

S.o.p("Public no arg constructor");

}

public ExternalizableDemo(String s, int i, int j){

this.s = s; this.i=i, this.j=j;

}

public void writeExternal(ObjectOutputStream out) throws IOException{

out.writeObject(s);

out.writeInt(i);

}

public void readExternal(ObjectInputStream in) throws IOException{

s = (String) in.readObject();

i = in.readInt();

}

public static void main (String[] args){

ExternalizableDemo exDemo = new ExternalizableDemo ("name123", "100", "200");

FileOutputStream fos = new FileOutputStream("file.ser");

ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeObject(d1);

FileInputStream fis = new FileInputStream("file.ser");

ObjectInputStream ois = new ObjectInputStream(fis);

ExternalizableDemo ex = (ExternalizableDemo) ois.readExternal();

S.o.p(ex.s + "..." + ex.i + "..." + ex.j);

}

}

Program Flow:

1. During serialization JVM will check whether the object is implementing Serializable or Externalizable.
2. Executes writeExternal() which has required variables to be serialized.
3. During de-serialization, JVM will create the object using no-arg constructor and call readExternal () and assigns the values. The variables which are not serialized will be assigned will null value.

Output:

Public no arg constructor

name123...100

|  |  |
| --- | --- |
| **Serialization** | **Externalization** |
| Meant for default Serialization | For customized Serialization |
| JVM takes care of everything and programmer doesnot have any control | Programmer takes care of everything |
| Allows only to save total object and partial saving is not allowed | Either total or partial objects can be saved. |
| Performance is low. | Performance is high |
| Serializable is Marker interface (does not contain any method) | Externalizable Contains 2 methods (writeExternal, readExternal) |
| Best when complete object must be saved | Best when part of object must be saved |
| Serializable implemented class not required to contain public no arg constructor | Externalizable implemented class must contain no-arg constructor or else InvalidClassException |
| transient keyword can be used | transient keyword will not play any role or not required. |

**SerialVersionUID**

During Serialization, with every object sender side JVM will send one unique ID. During deserialization, the receiver side JVM will compare the object unique id will local class unique id. This unique id saved with the object Is called **SerialVersionUID**.

If the ID does not match it throws InvalidClassException.

Problems with default serialVersionUI generated by JVM:

1. Both sender and receiver should use same version of JVM. If not, there will be a mismatch in ID.
2. Both sender and receiver should use same .class file version.
3. Chance of performance problem while generating ID.

Hence never let JVM to generate the ID.

Declare UID as

private static final long serialVersionUID = 1L ;