Externalization in Java

Before going into what externalization is, you need to have some knowledge on what [serialization](http://www.jusfortechies.com/java/core-java/serialization.php)is because externalization is nothing but serialization but an alternative for it and Externalizable interface extends Serializable interface. Check [Serialization](http://www.jusfortechies.com/java/core-java/serialization.php)article for information on serialization. Just as an overview, Serialization is the process of converting an object's state (including its references) to a sequence of bytes, as well as the process of rebuilding those bytes into a live object at some future time. Serialization can be achieved by an object by implementing Serializable interface or Externalizable interface.

*Well, when serialization by implementing Serializable interface is serving your purpose, why should you go for externalization?*

Good question! Serializing by implementing Serializable interface has some issues. Lets see one by one what they are.

* 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 super class of the object until it reaches the "Object" class and the same way the super class of the instance variables until it reaches the "Object" class of those variables. Basically all the objects that it can read. This leads to lot of overheads. Say for example, you need only car type and licence number but using serialization, you cannot stop there. All the information that includes description of car, its parts, blah blah will be serialized. Obviously this slows down the performance.
* 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 descption 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 issue.
* You know that serialization needs serialVersionUID, a unique Id to identify the information persisted. If you dont explicitly set a serialiVersionUID, serialization will compute the serialiVersionUID by going through all the fields and methods. So based on the size of the class, again serialization mechanism takes respective amount of time to calculate the value. A third performance issue.  
    
  Above three points confirm serialization has performance issues. Apart from performance issues,

When an object that implements Serializable interface, is serialized or de-serialized, 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 call 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***. Cool. Here enters the actual topic.

*So what is externalization?*

Externalization is nothing but serialization but by implementing Externalizable interface to persist and restore the object. To externalize your object, you need to implement Externalizable interface that extends 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.

How serialization happens? JVM first checks for the Externalizable interface and if object supports Externalizable interface, then serializes the object using writeExternal method. If the object does not support Externalizable but implement 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.

Lets see a simple example.

import java.io.\*;

public class Car 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 {

out.writeObject(name);

out.writeInt(year);

}

/\*\*

\* Mandatory readExternal method.

\*/

public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {

name = (String) in.readObject();

year = in.readInt();

}

/\*\*

\* Prints out the fields. used for testing!

\*/

public String toString() {

return("Name: " + name + "\n" + "Year: " + year);

}

}

import java.io.\*;

public class ExternExample {

public static void main(String args[]) {

// create a Car object

Car car = new Car("Mitsubishi", 2009);

Car newCar = null;

//serialize the car

try {

FileOutputStream fo = new FileOutputStream("tmp");

ObjectOutputStream so = new ObjectOutputStream(fo);

so.writeObject(car);

so.flush();

} catch (Exception e) {

System.out.println(e);

System.exit(1);

}

// de-serialize the Car

try {

FileInputStream fi = new FileInputStream("tmp");

ObjectInputStream si = new ObjectInputStream(fi);

newCar = (Car) si.readObject();

}

catch (Exception e) {

System.out.println(e);

System.exit(1);

}

/\*

\* Print out the original and new car information

\*/

System.out.println("The original car is ");

System.out.println(car);

System.out.println("The new car is ");

System.out.println(newCar);

}

}

In this example, class Car implements Externalizable interface which means that car object is ready for serialization. This class have two public methods - "writeExternal" and "readExternal". Unlike Serializable interface which will serialize all the variables in the object with just by implementing the interface, here you have to explicitly mention what fields or variables you want to serialize and the same is done in "writeExternal" and "readExternal" methods. So in the "ExternExample" class, when you write the "Car" object to the OutputStream, the "writeExternal" method is called and the data is persisted. The same applies to "readExternal" method in the Car object i.e., when you read the "Car" object from the ObjectInputStream, "readExternal" method is called.

*What will happen when an externalizable class extends a non externalizable super class?*

Then in this case, you need to persist the super class fields also in the sub class that implements Externalizable interface. Look at this example.

/\*\*

\* The superclass does not implement externalizable

\*/

class Automobile {

/\*

\* 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 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 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 super class fields first and then the sub class fields.

*Sounds good! What if the super class implements the Externalizable interface?*

Well, in this case the super class 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 dont need to save/restore the superclass fields in sub class 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.

*Lets see the difference in sizes when you serialize using Serializable interface and serialize using Externalizable interface*

Let's take a simple case, an object of type SimpleClass with just few fields - firstName, lastName, weight and location, containing data {"Brad", "Pitt", 180.5, {49.345, 67.567}}. When you serialize this object that is about 24 bytes by implementing Serializable interface, it turns into 220 bytes (approx). As it turns out, the basic serialization mechanism stores all kinds of information in the file so that it can deserialize without any other assistance. Look at the format below when the object is serialized and you will understand why it is turned out to 200 bytes.

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 Externalizable interface, the size will be reduced drastically and the information saved in the persistant 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]

*Well, externalization has its own limitations*

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 marshalling and demarshalling code whenever you change your class definitions.

As you know a default public no-arg constructor will be called when serializing the objects that implements 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 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 interface are public. So any malicious program can invoke which results into loosing 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.

*So much of it. Here are some final 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.

Complete Example on Externalizable

**package** com.ddlab.rnd.core;

**import** java.io.Externalizable;

**import** java.io.IOException;

**import** java.io.ObjectInput;

**import** java.io.ObjectOutput;

**public** **class** Person1 **implements** Externalizable {

**private** **int** id;

**private** String firstName;

**private** String lastName;

**private** **int** age;

**private** **float** salary;

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**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;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

**public** **float** getSalary() {

**return** salary;

}

**public** **void** setSalary(**float** salary) {

**this**.salary = salary;

}

@Override

**public** **void** writeExternal(ObjectOutput out) **throws** IOException {

out.writeInt(**this**.id);

out.writeObject(**this**.firstName);

out.writeObject(**this**.lastName);

out.writeInt(**this**.age);

out.writeFloat(**this**.salary);

}

@Override

**public** **void** readExternal(ObjectInput in) **throws** IOException,

ClassNotFoundException {

**this**.id = in.readInt();

**this**.firstName = (String) in.readObject();

**this**.lastName = (String) in.readObject();

**this**.age = in.readInt();

**this**.salary = in.readFloat();

}

@Override

**public** String toString() {

**return** "Person [id=" + id + ", firstName=" + firstName + ", lastName="

+ lastName + ", age=" + age + ", salary=" + salary + "]";

}

}

package com.ddlab.rnd.test;

import java.io.File;

import java.io.FileInputStream;

import java.io.ObjectInputStream;

import com.ddlab.rnd.core.Person1;

public class Test3 {

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

Person1 person = new Person1();

person.setId(11);

person.setFirstName("Deb, Kumar");

person.setLastName("Mishra");

person.setAge(23);

person.setSalary(1234f);

File file = new File("data/t2.ser");

// ObjectOutputStream out = new ObjectOutputStream( new FileOutputStream(file));

// out.writeObject(person);

// out.flush();

// out.close();

ObjectInputStream oin = new ObjectInputStream( new FileInputStream(file));

Person1 p = (Person1) oin.readObject();

System.out.println(p);

}

}