

Serialization



Serialization Overview

- Allows objects to be saved to and restored from streams
 - includes files and sockets
- Goals
 - be simple to use
 - be usable for all objects without customization
 - not fully achieved, have to implement Serializable
 - be extensible
 - to customize what is written and read and how
 - support marshaling and unmarshaling for RMI
 - support persistence
 - saving the state of a collection of objects and restoring that state in a future session
 - maintain security
 - through ability to customize what is written



Serialization Contents

- Full class name
 - includes package name
- Class fingerprint
 - a hashcode derived from the following class elements
 - interfaces
 - field names and access modifiers
 - methods names, access modifiers, and signatures
 - changes to these, other than order, change the fingerprint
- Field values
 - only those that are not static or transient
 - from highest superclass to the class of the object
 - serialized and deserialized in an order that does not depend on the order in which they appear in the class definition
 - can read objects that were saved before the field order was changed



Serialization of Graph Structures

- When serializing one object, all objects reachable from it are also serialized
- If an attempt is made to serialize an object that has already been serialized, a “handle” to the previously serialized object is written
- Solves problem of dealing with
 - multiple references to the same object
 - circular references
- When an object is deserialized, all objects accessible from it are also deserialized



Serializable SalesEntry

```
import java.io.*;

public class SalesEntry implements Serializable {
    private String name;
    private Date date;
    private float sales;
    private transient int baseSalary;

    SalesEntry(String name,
                Date date,
                float sales,
                int baseSalary) {
        this.name = name;
        this.date = date;
        this.sales = sales;
        this.baseSalary = baseSalary;
    }
}
```



SalesEntry With Serialization (Cont'd)

- **Serializing**

```
SalesEntry entry =
    new SalesEntry("Mark Volkmann",
                   new Date(),
                   1234.56f,
                   100000);

FileOutputStream fos = new
    FileOutputStream("sales.jos");

ObjectOutput oo = new ObjectOutputStream(fos);
oo.writeObject(entry);
oo.close(); // can also oo.flush() since it's buffered
```

jos stands for Java
Object Serialization.
No special extension
is required.

ObjectOutput is an interface that
ObjectOutputStream implements.

ObjectInput is an interface that
ObjectInputStream implements.

- **Deserializing**

```
FileInputStream fis = new FileInputStream("sales.jos");
ObjectInput oi = new ObjectInputStream(fis);
try {
    SalesEntry entry = (SalesEntry) oi.readObject();
} catch (ClassNotFoundException e) {
    System.err.println(e);
}
oi.close();
```



Security Concerns

- In JDK 1.1 classes are not serializable by default
 - useful when programmers forget to consider security
 - classes must declare themselves to be serializable with `implements Serializable`
- Reasons to customize serialization
 - only write and read non-sensitive fields
 - excluding passwords, credit card numbers, file handles, ...
 - encrypt and decrypt sensitive fields
 - another way of doing this is to insert encrypting and decrypting filters between the object and file streams to encrypt everything
 - add compression
 - verify that data being deserialized has not been made invalid after it left Java's control (ex. checksum)
- Three ways to customize serialization
 - mark sensitive fields as transient
 - implement `Serializable` and override `readObject()` and `writeObject()`
 - implement `Externalizable` and override `writeExternal()` and `readExternal()`



readObject() & writeObject()

(only the changes to SalesEntry are shown)

- MyEncrypter is a fictitious class that
 - encrypts primitive values into Strings
 - decrypts those Strings back to primitive values

```
public class SalesEntry implements Serializable {  
  
    private void writeObject(ObjectOutputStream out) throws IOException {  
        // Write all fields that are not transient or static,  
        // including those in base classes.  
        out.defaultWriteObject();  
  
        // Write transient field baseSalary as an encrypted String.  
        out.writeUTF(MyEncrypter.encryptInt(baseSalary));  
    }  
  
    private void readObject(ObjectInputStream in) throws IOException {  
        // Read all fields that are not transient or static  
        // including those in base classes.  
        in.defaultReadObject();  
  
        // Read transient field baseSalary from an encrypted String.  
        baseSalary = MyEncrypter.decryptInt(in.readUTF());  
    }  
}
```



writeExternal() & readExternal()

(only the changes to SalesEntry are shown)

taking full responsibility
for serializing all
required fields

extends Serializable

```
public class SalesEntry implements Externalizable {  
  
    // A public no-arg constructor is required by  
    // Externalizable.readExternal.  
    public SalesEntry() {  
    }  
  
    // Method in the Externalizable interface.  
    public void writeExternal(ObjectOutput out) throws IOException {  
        // Can't use ObjectOutputStream.defaultWriteObject().  
        // If base class fields require serialization  
        // then code to perform that must be supplied here.  
        out.writeUTF(name);  
        out.writeObject(date); // use for objects and arrays  
        out.writeFloat(sales);  
        out.writeUTF(MyEncrypter.encryptInt(baseSalary));  
    }  
  
    // Method in the Externalizable interface.  
    public void readExternal(ObjectInput in)  
    throws IOException, ClassNotFoundException {  
        // Can't use ObjectInputStream.defaultReadObject().  
        // If base class fields require deserialization  
        // then code to perform that must be supplied here.  
        name = in.readUTF();  
        date = (Date) in.readObject(); // use for objects and arrays  
        sales = in.readFloat();  
        baseSalary = MyEncrypter.decryptInt(in.readUTF());  
    }  
}
```



Treating Deserializaton as a Transaction

- The problem
 - attempting to deserialize an object that contains references to other objects results in many objects being deserialized
 - an exception could be throw during deserialization of any object
 - ex. ClassNotFoundException if fingerprint has changed
 - if all of the objects in the graph are not deserialized, the objects that are deserialized may not be usable
 - those objects may require cleanup
- The solution
 - invoke registerValidation(ObjectInputValidation) on the ObjectInputStream
 - must pass an object that implements the ObjectInputValidation interface
 - it requests that validateObject () of the object passed to registerValidation be called
 - after the graph is deserialized
 - before the main object being deserialized is returned from readObject ()
 - validateObject can determine whether cleanup is needed and perform it
 - throws InvalidObjectException otherwise

