Software Development (CS2500)

Lecture 41: Serialization

M.R.C. van Dongen

February 2, 2011

Contents

1 Outline

In this lecture we shall study how to save objects. There are two ways to do this.

Hard way: Write dedicated save methods for all relevant classes.

Easy way: Implement `Serializable` in all relevant classes.

As part of this lecture we shall study `ObjectOutputStream` and `ObjectInputStream` objects. We shall use these objects to write objects to and read objects from files.

2 Saving State

Many real-world applications save state. For example commercial applications such as database systems.

One obvious reason why a program would need to save its state is that the computer which it's running on may not be able to stay on all the time. For example, there may be unforseeable power cuts, the computer may be relocated, ... To solve the problem, the program saves and restores its data.

- The program's data (state) is written to a secondary backup medium: a file on a harddisk, on a CD, on a DVD, on a tape, ...
- The program terminates.
- When it starts again, it reads its state from the backup medium.

Saving state is also useful for backups, data exchange, and communication.

2.1 Saving a Game

Let's assume you're writing an adventure game. The game involves `GameCharacter` objects: `Hobbit`, `Elf`, `Ork`, ... You want a save-restore option. This let's you play the game, save it, restore it, and continue where you stopped. You have two options:

1. Use serialisation. To save the game you write serialized objects to a file. (The file won't make sense to the human eye.) To restore the game you read in the serialized objects. This is the easier option.
2. Save and restore all information to and from a text file. This is complicated because you have to implement write/read methods for *all* classes in your application. Moreover, you may have to change these methods each time you change your classes.

3 Streams

Let's assume our game has three `GameCharacters`: `frodo`, a `Hobbit`, `legolas`, an `Elf`, and `gandalf`, a `Wizard`. The following four steps show how you save the objects' `GameCharacter` state in the serialization framework.

1. We create a `FileOuputStream` object. Using this object we can write low-level data to an external file.

   ```
   FileOutputStream outputStream = new FileOutputStream( "Game.ser" );
   ```
   Java

   Here `Game.ser` is the name of the file we're going to write the state to.

2. We create an `ObjectOuputStream`. This object is built on top of the `FileOuputStream` object. Using it we can write objects to the external file.

   ```
   ObjectOutputStream os = new ObjectOutputStream( outputStream );
   ```
   Java

   The reason why we need to do this is that we cannot directly write objects to the `FileOutputStream`.

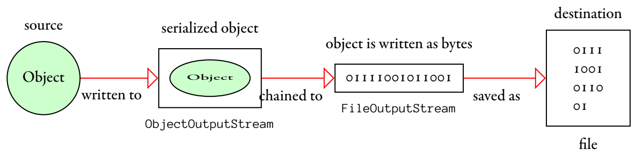

Figure 1: Writing an object to a file.

3. Using the `FileOuputStream` we write the `GameCharacter` objects:

   ```java
   os.writeObject( frodo );
   ```

   Saving the other `GameCharacter` objects is similar.

4. Finally, we close the `ObjectOutputStream`:

   ```java
   os.close( );
   ```

   By closing the `ObjectOutputStream` object we make sure the external file is propetly closed. You should alway close your `ObjectOutputStream` objects as soon as possible. Failing to do so may result in loss of data. For example, if there's a power cut and the `ObjectOutputStream` object isn't closed then you may lose data. However, closing the object before the power cut makes sure your data is properly saved.

3.1 How it Works

When we saved our object states using the `ObjetcOutputStream` object, several things went on. This section explains what went on and how this works.

`Java` *Streams* are for doing I/O. There are two kinds of streams: *connection* streams and *chain* streams. A *connection stream* connects a source to a destination. Possible source and destinations are files and network sockets. Connection streams are low-level. They are used to send/receive byte streams. *Chain streams* are built on top of other streams. Chain streams are for high-level communication. They use the underlying streams to do the communication. For example, they may translate an object to/from a byte sequence before passing on the bytes sequence to the undelying connection stream. Figure 1 depicts the process graphically.

3.2 What is the Object's State?

As part of the serialization process the object's state is saved as bytes. For objects with primitive-valued attributes this is easy: Just write the primitive values as bytes. How does it work for attributes which are object references?

- The values are references: memory locations. You can't save these. For example, you have no control over memory at deserialization time.

- In reality object reference values are serialized recursively.
- Recursive serialization works properly: if two variables reference the same object before serialization, then they also reference the same object after serialization.

4 Serialization

In `Java`, writing objects to `ObjectOutputStreams` is easy. All you have to do is make sure the object you write is an instance of a class which implements the `Serializable` interface. If you implement this interface, the methods that save and restore object state comes for free.

The following shows some of the game application.

```java
import java.io.*;
public class Game {
    public static void main( String[] args ) {
        ...
        try {
            FileOutputStream fileStream
                = new FileOutputStream( "Game.ser" );
            ObjectOutputStream output
                = new ObjectOutputStream( fileStream );
            output.writeObject( frodo );
            output.writeObject( legolas );
            output.writeObject( gandalf );
            output.close( );
        } catch (Exception exception) {
            handleException( exception );
        }
    }
    ...
}
```

The `Hobbit` class implements `Serializable`.

```java
import java.io.Serializable;
public class Hobbit implements GameCharacter,
                               Serializable {
    ...
}
```

The `Elf` and `Wizard` classes are implemented similarly.

Implementing `Serializable` is just a contract: you can write objects from the class, and you can read objects from the class. `Serializable` does not define any methods. So there's no need to `@Override` anything.

The `Serializable` contract is a contradiction in terms. For example, `Serializable` is an interface, yet there's no need for any `@Override`. So, what provides the implementation of `readObject( )` and `writeObject( )`. There's no real answer, it "just" works. Let's call it magic.

4.1 Not all Objects are `Serializable`

Some objects are not `Serializable`.

- There may be several reasons for this. For example, the implementor may have forgotten to implement `Serializable`.
- Perhaps it was a deliberate choice ....
- The most important reason why some objects aren't `Serializable` is that certain things cannot/should not be saved: network connections, file objects, passwords (security problem), other objects depending on specific *run-time dependent* experience.

Any attempt to serialize a non-`Serializable` object causes an exception. If an attribute should *not* be serialized you mark it `transient`. For example, in the following class the attribute `publicId` is specific to the `Chat` sessions. When a session is closed, the `publicId` is lost. When we're deserializing it is impossible to start a new session with a given `publicId`: we're (more than likely) getting a different `publicId`. This is why `serialized` is marked `transient`. There's simply no point in saving it.

```java
public class Chat implements Serializable {
    transient int publicId; // Not serialized.
    String userName;        // Serialized.
    ...
}
```

4.2 Tricks of the Serialization Trade

So what if you want to save a non-`Serializable` object? You extend the object's class and make the subclass `Serializable`. This only works if the class is extendable. (If it is not marked `final`.) Of course your program has to use the subclass throughout.

5 Deserialization

Deserialization works similar to serialization. This time, however, you *read* from an `Object`*`Input`*`Stream` object. The following demonstrates the main ideas.

```java
import java.io.*;
public class Game {
    public static void main( String[] args ) {
        ...
        try {
            FileInputStream fileStream
                = new FileInputStream( "Game.ser" );
            ObjectInputStream input
                = new ObjectInputStream( fileStream );
            frodo = (Hobbit)input.readObject( );
            legolas = (Elf)input.readObject( );
            gandalf = (Wizard)input.readObject( );
            input.close( );
        } catch (Exception e) {
            handleException( );
        }
        ...
    }
}
```

Notice that the method `readObject( )` returns an `Object` reference, so we have to cast it before assigning it to the destination object reference variable.

5.1 Order Matters

With serializing and deserializing the order of reads and writes matters. The serialized file is just a sequence of bytes. If the first n written bytes represent `frodo` then the first n bytes read also represent `frodo`. If the next m written bytes represent `legolas` then the next m read bytes also represent `legolas`. And so on.

The easiest way to overcome problems related to order is writing a single object.

The following demonstrates the main ideas. The implementation of the methods `saveGame( )` and `restoreGame( )` are provided in the next two listings. Before studying the next two listings try if you understand why implementing the method `saveGame( )` as an instance method is possible. Next see if you understand why implementing `restoreGame( )` as an instance method does not make much sense.

```java
import java.io.*;
public class Game implements Serializable {
    private final Hobbit hobbit = new Hobbit( );
    private final Elf elf = new Elf( );
    private final Ork ork = new Ork( );

    ...

    private void saveGame( ) {
        ⟨omitted⟩
    }

    private static Game restoreGame( ) {
        ⟨omitted⟩
    }
}
```

The following is the method `saveGame( )`.

```java
private void saveGame( ) {
    try {
        FileOutputStream fileStream
            = new FileOutputStream( "Game.ser" );
        ObjectOutputStream output
            = new ObjectOutputStream( fileStream );
        output.writeObject( this );
        output.close( );
    } catch (Exception exception) {
        handleException( exception );
    }
}
```

The following is the method `restorGame( )`.

```java
private static Game restoreGame( ) {
    Game game = null;
    try {
        FileInputStream fileStream
            = new FileInputStream( "Game.ser" );
        ObjectInputStream input
            = new ObjectInputStream( fileStream );
        game = (Game)input.readObject( );
        input.close( );
    } catch (Exception exception) {
        handleException( exception );
    }
    return game;
}
```

5.2 Tricks of the Deserialization Trade

So what if an attribute's object class is `transient` and this class is not extendable? Well, if the attribute is `transient`, then attribute becomes `null` after deserializing. It is crucial that the attribute gets a proper value. Depending on the attribute, there may be several ways to do this. If the attribute's value doesn't matter you can give it a default value. If the attribute's value does matter then you may be able to *compute* it. This is possible if your attribute's value depends on other values. (These values may be instance attributes of the object which is referenced by the attribute.)

- You save the other values at serialization time.
- You read them in at deserialization time.
- You use them to construct a new value for the attribute.

This cause problems if the `transient` attribute is `final`.

6 For Friday

Study the lecture notes, study Chapter 13, and study Pages 429–446 of Chapter 14.