Advanced Object Serialization

(고급 객체 직렬화)

중앙대학교 컴퓨터공학과
자바 동호회
JSTORM
http://www.jstorm.pe.kr

Document Information

Document title:	고급 객체 직렬화
Document file name:	AdvancedSerialization_박준용_final.doc
Revision number:	<1.0>
Issued by:	<JSTORM>
Issue Date:	<2001/09/11 >
Status:	Final

Content Information

Audience	자바 중 , 고급
Abstract	이 문서는 객체 직렬화에서 고급 기술들을 설명한다
Reference	(http://developer.java.sun.com/developer/technicalArticles/ALT/serialization/?frontpage-jdc) 를 번역
Benchmark information	

Table of Contents

Advanced Object Serialization

(고급 객체 직렬화)

By John Zukowski

August 2001

원문은 다음 페이지에서 보실 수 있습니다.

http://developer.java.sun.com/developer/technicalArticles/ALT/serialization/?frontpage-jdc

이후 나오는 Serialization 을 직렬화로 번역하겠습니다. 가능한 번역이 매끄럽고 알아보기 쉽게 하도록 가능한 직렬화를 사용하고 직렬화를 쓰기가 불편한 곳에서는 영어를 그대로 쓰도록 하겠습니다.

전체 요약

대부분의 기술문서들은 직렬화에 대한 기본적인 것들(직렬화가 무엇인가와 사용법 등에 관한 것들) 을 다룬다. 실제로 객체 직렬화는 더 많은 기능을 가지고 있다. 이 기술문서에서는 직렬화의 기본적인 것들을 살펴보고 직렬화된 스트림 인증에 대한 설명 , ObjectInputStreamField 를 통한 serialzable 필드 다루기 , 객체 스트림 암호화 (encrypting) 에 대해 다룬다.

직렬화(Serialization)란?

객체 직렬화의 기본적인 개념은 객체를 바이트 스트림으로 쓰고 읽게 하는 것이다. 이러한 스트림들은 서블릿에서 세션 상태 저장이나 RMI(Remote Method Invocation) 호출에서 파라미터 전송 , 자바빈즈 컴포넌트에서 상태 정보 저장 , 네트웍을 통한 객체 전송 등 많은 작업들에 사용할 수 있다.

ObjectOutputStream 을 사용해서 객체를 저장하고 ObjectInputStream 으로 저장한 객체를 읽어들인다. 직렬화 과정은 모든 미처리 데이터를 객체로 만들고 그 객체를 참조하는 모든 객체를 저장하는 객체 트리를 다룬다. 객체를 다시 읽어

들일 때 원본 객체 트리 그래프는 재생성 된다. 하나의 객체에 대한 원형 레퍼런스(circular references) 와 다중 레퍼런스(multiple references) 같은 특별한 경우는 트리에서 다른 객체에 대한 참조가 마법적으로 보이지 않게 트리 그래프가 새로운 객체로 재생성 될 때 보존된다. (간단히 말하자면 직렬화 하기 전의 상태로 정확히 되돌려 준다는 말)

객체가 직렬화 처리 과정을 지원하려면 그 클래스는 Serializable 인터페이스를 implement 해야 한다. Serializable 인터페이스는 아무런 메소드도 가지지 않는다. 이 인터페이스는 클래스가 직렬화 가능하다는 것을 알려주는데에만 사용된다. 그러나 단순히 클래스 정의에 implement Serializable 을 추가하는 것만으로 이 클래스가 자동으로 직렬화 가능(Serializable) 해지지는 않는다. 또한 클래스의 모든 인스턴스 변수들은 Serializable 해야 한다. Serializable 하지 않은 클래스를 직렬화 하려고 한다면 예외가 발생한다. java.awt.Image 와 java.lang.Thread 같은 특정 플랫폼 구현 정보를 가진 클래스들은 Serializable 하지 않고 transient로 표시해야 한다.

transient Image image;

클래스를 직렬화 할 때 non-static 이거나 non-transient 인스턴스 데이터들만 저장된다. 클래스 정의는 저장되지 않는다. 이 인스턴스 데이터들은 객체를 역직렬화(deserialize) 할 때 이용가능해진다.

객체를 직렬화하는 기본적인 과정은 다음과 같다.

```
ObjectOutputStream oos =
       new ObjectOutputStream(anOutputStream);
Serializable serializableObject = ...
       oos.writeObject(serializableObject);
```

그 후 객체를 다시 읽어들이는 역직렬화는 그 반대로 수행하면 된다.

```
ObjectInputStream ois =
       new ObjectInputStream(anInputStream);
Object serializableObject = ois.readObject();
```

직렬화에서 자주 설명되는 또 다른 것은 기본적인 직렬화 행동을 변경하는 readObject 와 writeObject 메소드 오버라이딩 이다. 이 메소드를 오버라이딩함으로써 역직렬화 시에 다시 읽어들일 수 있도록 output 스트림에 추가적인 정보를 제공할 수 있다.

```
private void writeObject(ObjectOutputStream oos)
    throws IOException {
  oos.defaultWriteObject();
  // Write/save additional fields
  oos.writeObject(new java.util.Date());
}

// assumes "static java.util.Date aDate;" declared
private void readObject(ObjectInputStream ois)
    throws ClassNotFoundException, IOException {
  ois.defaultReadObject();
  // Read/initialize additional fields
  aDate = (java.util.Date)ois.readObject();
}
```

readObject 와 writeObject를 오버라이드 하는 가장 큰 이유는 Serializable 하지 않는 슈퍼클래스의 데이터들을 직렬화 하기 위해서이다.

그리고, 여기까지가 대부분의 직렬화 설명의 끝이다.

스트림 인증하기(Validating Streams)

데이터를 파일로 직렬화하거나 소켓으로 데이터를 전송할 때 제대로 받았는지 제대로 쓰여져 있는지 보장할 수 없다. 파일로 저장하는 경우 부정한 사용자가 에디터를 사용해서 값을 변경시킬 수 있다.

직렬화에서 클래스는 객체가 완전히 복원된 이후 인증을 수행하도록 ObjectInputValidation 인터페이스를 구현한 ObjectInputStream 파라미터를 readObject 에게 제공한다. 이 인터페이스는 `public void validateObject() throws InvalidObjectException` 라는 하나의 메소드로 구성된다. 구현시 클래스의 readObject 메소드는 `registerValidation(ObjectInputValidation, int)` 를 통해 ObjectInputStream을 인증자(validator) 에게 등록해야 한다. 이 메소드의 마지막 아규먼트는 인증자(validator) 우선순위로서 여러 개의 값을 가질 수 있다. (높은 값일수록 먼저 호출된다.)

```
private void readObject(ObjectInputStream ois)
    throws ClassNotFoundException, IOException {
  ...
  ois.registerValidation(validator, 0);
   ...
}
```

설명을 돕기 위해 다음 클래스는 인증을 사용한다. 두개의 인스턴스 변수중 어느 하나가 숫자 6이라면 읽기는 실패한다. 테스트 프로그램인 ValidationExample 은 코맨드 라인에서 두개의 인스턴스 변수의 값을 얻는다.

```
import java.io.*;

public class ValidationExample
    implements Serializable, ObjectInputValidation {
  private int x, y;
  public static void main(
  String args[]) throws Exception {

    if (args.length != 2) {
      System.err.println(
      "Please pass in two numbers");
      System.exit(-1);
    }

    // Initialize object
    ValidationExample ve = new ValidationExample();
    try {
      ve.x = Integer.parseInt(args[0]);
      ve.y = Integer.parseInt(args[1]);
    } catch (NumberFormatException e) {
      System.err.println(
      "Please pass in two numbers");
      System.exit(-1);
    }

    FileOutputStream fos =
     new FileOutputStream("val.ser");
    ObjectOutputStream oos =
     new ObjectOutputStream(fos);
    oos.writeObject(ve);
    oos.close();

    try {
      FileInputStream fis =
      new FileInputStream("val.ser");
      ObjectInputStream ois =
       new ObjectInputStream(fis);
      ValidationExample ve2 =
       (ValidationExample)ois.readObject();
      ois.close();
      System.out.println(ve2);
    } catch (InvalidObjectException invalid) {
      System.err.println(invalid.getMessage());
```

```
    }
  }

  public String toString() {
    return getClass().getName(
    ) + "[x=" + x + ",y=" + y + "]";
  }

  private void readObject(ObjectInputStream ois)
      throws ClassNotFoundException,
      IOException {
    ois.registerValidation(this, 0);
    ois.defaultReadObject();
  }

  public void validateObject()
      throws InvalidObjectException {
    if ((x == 6) || (y == 6)) {
      throw new InvalidObjectException(
        "6 is an invalid entry. Can't restore.");
    }
  }
}
```

ObjectStreamField 사용하기

객체가 직렬화될 때 어떤 필드가 스트림화 될지를 정의하는 두가지 방법이 있다. 기본적으로 non-static 과 non-transient 필드는 저장된다. 그러나 , 클래스에서 serialPersistentFields (private , static , final 인 경우)라 이름지어진 ObjectStreamField 객체들의 배열을 정의한다면 명시적으로 특정한 필드만을 저장하도록 선언할 수 있다. 배열내의 필드 순서는 쓴 순서대로 이다. 예를 들어 다음 클래스에서 username 과 counter 필드는 직렬화되고 password는 직렬화 되지 않는다.

```
public class MyClass implements Serializable {
  private String username;
  private int counter;
  private String password;

  private final static ObjectStreamField[]
    serialPersistentFields = {
      new ObjectStreamField(
      "username", String.class),
      new ObjectStreamField("counter", int.class)
    };
```

```
  ...
}
```

기본적으로 serialPersistentFields 설정을 제공할 때 readObject 와 writeObject의 최적화는 필요치 않다.

클래스 변경시에 serialPersistentFields는 유용하다. 하나의 클래스가 두개의 필드를 가진다고 생각해보자.

```
Point point;
Dimension dimension;
```

여기서 하나의 필드를 가진 클래스를 생각해보자.

Rectangle rectangle;

양 클래스 간 양방향(bidirection) 직렬화를 원한다면 두번째 클래스에서 point 와 dimension을 rectangle로 맵핑하는 serialPersistentFie lds를 생성할 수 있다.

```
private static final
ObjectStreamField[] serialPersistentFields = {
  new ObjectStreamField("point", Point.class),
  new ObjectStreamField("dimension", Dimension.class)
};
```

readObject와 writeObject에서 실제적인 맵핑을 한다.

```
private void readObject(ObjectInputStream ois)
    throws ClassNotFoundException, IOException {

  // Read version one types
  ObjectInputStream.GetField fields =
  ois.readFields();
  Point point = (Point)fields.get("point", null);
  Dimension dimension =
  (Dimension)fields.get("dimension", null);

  // Convert to version two type
  rectangle = new Rectangle(point, dimension);
}

private writeObject(ObjectOutputStream oos)
```

```
    throws IOException {

  // Convert to version one types
  ObjectOutputStream.PutFields fields =
   oos.putFields();
  fields.put("point", rectangle.getLocation());
  fields.put("dimension", rectangle.getSize());

  // Write version one types
  oos.writeFields();
}
```

두번째 클래스는 첫번째 클래스처럼 동일한 serialVersionUID를 가져야 한다. 변경하기 전에 첫번째 클래스에서 serialver 명령을 실행하라.

SerialPersistentFields를 사용해서 클래스를 변경시키는 완전한 예제는 다음을 참고하라.

Using Serialization and the Serializable Fields API

http://java.sun.com/j2se/1.4/docs/guide/serialization/examples/altimpl/index3.html

직렬화된 객체 암호화 하기 (Encrypting Serialized Object)

JCE(Java Cryptography Extension)는 암호화를 지원한다. 직렬화에서는 모든 스트림을 CipherOutputStream으로 암호화할 수 있고 각 객체를 SealedObject 클래스를 통해 Cipher로 봉인할 수 있다. 보통 CipherOutputStream을 사용해서 전체 스트림을 암호화 한다. 그러나 SealedObject 로 어떠한 Serializable 객체라도 봉인해서 서블릿 세션에 저장할 수 있다.

다음 예제는 객체를 암호화해서 디스크에 쓰는데 CipherOutputStream 을 사용하고 파일에서 복호화(암호화 된걸 푸는 일 암호화의 반대)하는데 CipherInputStream 를 사용한 후 객체를 다시 읽어들인다. 암호화된 스트림은 다른 필터링 스트림이랑 똑같이 동작한다. 파일과 객체 스트림사이에 스트림을 추가한 것 뿐이다. 소스코드에서 가장 중요한 부분은 암호화/복호화 키인 Cipher을 생성하는 부분이다.

```
import java.io.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import java.awt.*;

public class CipherExample {
```

```
// Password must be at least 8 characters
private static final String password =
"zukowski";

public static void main(String args[]
) throws Exception {
  Point point = new Point(100, 200);
  Dimension dim = new Dimension(300, 400);
  Rectangle rect = new Rectangle(point, dim);

  // Create Key
  byte key[] = password.getBytes();
  DESKeySpec desKeySpec = new DESKeySpec(key);
  SecretKeyFactory keyFactory =
  SecretKeyFactory.getInstance("DES");
  SecretKey secretKey =
  keyFactory.generateSecret(desKeySpec);

  // Create Cipher
  Cipher desCipher =
  Cipher.getInstance("DES/ECB/PKCS5Padding");
  desCipher.init(Cipher.ENCRYPT_MODE, secretKey);

  // Create stream
  FileOutputStream fos =
   new FileOutputStream("out.des");
  BufferedOutputStream bos =
   new BufferedOutputStream(fos);
  CipherOutputStream cos =
   new CipherOutputStream(bos, desCipher);
  ObjectOutputStream oos =
   new ObjectOutputStream(cos);

  // Write objects
  oos.writeObject(point);
  oos.writeObject(dim);
  oos.writeObject(rect);
  oos.flush();
  oos.close();

  // Change cipher mode
  desCipher.init(Cipher.DECRYPT_MODE, secretKey);

  // Create stream
  FileInputStream fis =
   new FileInputStream("out.des");
  BufferedInputStream bis =
  new BufferedInputStream(fis);
  CipherInputStream cis =
  new CipherInputStream(bis, desCipher);
```

```
      ObjectInputStream ois =
       new ObjectInputStream(cis);

      // Read objects
      Point point2 = (Point)ois.readObject();
      Dimension dim2 = (Dimension)ois.readObject();
      Rectangle rect2 = (Rectangle)ois.readObject();
      ois.close();

      // Compare original with what was read back
      int count = 0;
      if (point.equals(point2)) {
        System.out.println("Points are okay.");
        count++;
      }
      if (dim.equals(dim2)) {
        System.out.println("Dimensions are okay.");
        count++;
      }
      if (rect.equals(rect2)) {
        System.out.println("Rectangles are okay.");
        count++;
      }
      if (count != 3) {
        System.out.println(
        "Problem during encryption/decryption");
      }
    }
}
```

암호화하는 두번째 방법은 객체를 봉인(sealing)**하는 것이다**. SealedObject **의 생성자를 호출함으로써 객체를 봉인한다**. Serializable **객체를 봉인하기 위해 생성자에 파라미터로 넘기고 암호화를 위해** Cipher**를 사용한다**.

```
SealedObject sealedObject =
   new SealedObject(serializable, cipher);
```

객체 봉인을 해제할때는 세가지 getObject **메소드를 사용할 수 있다**.

- getObject(Cipher c)
- getObject(Key key)
- getObject(Key key, String provider)

이러한 봉인해제 메소드 각각은 ObjectInputStream에서 readObject 메소드처럼 동작한다. (리턴된 객체를 적당한 타입으로 형변환 해야 한다.) 어느 메소드를 사용할지는 상황에 따라 달라지지만 두번째 메소드(복호화기(decrypter , 복호화하는 놈)가 암호화 파라미터를 몰라도 되기 때문에)가 가장 많이 사용된다. 첫번째 메소드는 Cipher 객체가 어플리케이션 내에서 공유될 때 일반적으로 사용된다.

다음은 AWT Rectangle 객체를 봉인 하고 봉인해제 하는 예제이다.

```
import java.io.*;
import javax.crypto.*;
import javax.crypto.spec.*;
import java.awt.*;

public class SealedExample {
  // Password must be at least 8 characters
  private static final String password = "zukowski";

  public static void main(String args[]
  ) throws Exception {
    Point point = new Point(100, 200);
    Dimension dim = new Dimension(300, 400);
    Rectangle rect = new Rectangle(point, dim);

    // Create Key
    byte key[] = password.getBytes();
    DESKeySpec desKeySpec =
    new DESKeySpec(key);
    SecretKeyFactory keyFactory =
     SecretKeyFactory.getInstance("DES");
    SecretKey secretKey =
    keyFactory.generateSecret(desKeySpec);

    // Create Cipher
    Cipher desCipher =
    Cipher.getInstance("DES/ECB/PKCS5Padding");
    desCipher.init(Cipher.ENCRYPT_MODE, secretKey);

    // Seal object
    SealedObject sealedObject =
     new SealedObject(rect, desCipher);

    // Change cipher mode
    desCipher.init(Cipher.DECRYPT_MODE, secretKey);

    // Unseal object
    Rectangle rect2 =
    (Rectangle)sealedObject.getObject(secretKey);
```

```
    // Just print each out
    System.out.println(rect);
    System.out.println(rect2);
  }
}
```

다른 특징

직렬화에는 여기에서 설명한 것보다 더 많은 특징들이 있다.

Java Object Serialization Specification (http://java.sun.com/j2se/1.4/docs/guide/serialization/spec/serialTOC.doc.html) 이 자바2 플랫폼 1.2 릴리즈 이후로 변한 것이 없고 1.3 릴리즈(http://java.sun.com/j2se/1.4/docs/guide/serialization/relnotes.html)와 1.4 릴리즈(http://java.sun.com/j2se/1.4/docs/guide/serialization/relnotes14.html) 에서는 단순히 향상만 되었다.

직렬화에 대한 흥미가 없어지면서 처음 발표된 이후로 변경된 점이 없다. 가끔 Serializable 을 상속한 클래스를 상속받을 때 상속받은 클래스가 직렬화되지 않기를 원할 수 있다. 이러한 경우 readObject 와 writeObject 메소드에서 NotSerializableException 을 던져라(throw).

```
private void readObject(ObjectInputStream ois)
    throws ClassNotFoundException, IOException {
  throw new NotSerializableException();
}

private void writeObject(ObjectOutputStream ois)
    throws IOException {
  throw new NotSerializableException();
}
```

결론

이 기술문서에서 객체 직렬화에 대한 몇가지 특징들을 배웠다. 이러한 특징들이 모두 필요하지는 않겠지만, 시스템 설계시에 알고 있고 이해하고 있다면 도움이 될 것이다.

Resources

? Java Cryptography Extension (JCE)

(http://java.sun.com/products/jce/)

? Serialization Documentation

(http://java.sun.com/j2se/1.3/docs/guide/serialization/

? Sun Object Serialization FAQ

(http://java.sun.com/products/jdk/serialization/faq/)

? jGuru Object Serialization FAQ

(http://www.jguru.com/faq/home.jsp?topic=Serialization)