Object Serialization

Objectives

- Learn how to persist object state using serialization
- Learn how to control the serialization process
- Understand object versioning

Serialization

- What if you want to save the state of an object between sessions?
  - User interface geometry
  - Session parameters (ip addresses, modem init strings, etc.)
  - Email address book in a mail reader program
  - Objects drawn by the user in your new jPhotoshop software

Serialization (cont'd)

- Serialization allows you to save object state to a stream
- Provides a simple persistence mechanism for objects
- Default mechanism saves the value of nonstatic, nontransient attributes
- Most Java API objects are serializable, but you should check if unsure
- Application-specific, i.e. more commonly used in some applications than others

Serialization (cont'd)

- Pros
  - Easy to store binary representation of objects
  - Serialization can be to any stream: file, network, byte[], etc.
  - Very flexible framework
  - Works well for *value objects* and classes not designed for inheritance
- Cons
  - Maybe too easy…
  - Default serialization mechanism does not perform well
  - Security problems
  - Versioning issues
  - Does not work well for classes designed for inheritance

The `Serializable` Interface

- To be serialized, an object’s class must implement the `java.io.Serializable` interface
- `Serializable` contains no methods (it’s a marker interface, like `Cloneable`)
- What good is a “marker” interface, anyway?

What Serialization Does

- Serialization is a “deep copy” operation…it will attempt to serialize all contained objects in turn (and all of those objects’ contained objects, and so on)
- If a contained object somewhere does not implement `Serializable`, a `java.io.NotSerializableException` will be thrown

Serialization File Format

- File format
  - File begins with a magic number AC ED
  - Version number, currently 00 05
  - Objects
- Objects have their ids
  - String 74, Class 72, Object 73
  - Used to prevent cycles in object graph
- String “Java” will be saved as :

Magic #		Version		Id	Size		“Java”			
AC	ED	00	05	74	00	04	4A	61	76	61

Storing Object Descriptions

- Saving arbitrary classes requires saving the class info itself
  - Name of a class
  - Serial version unique ID (fingerprint)
  - Description of method used
  - Description of data fields
- Constants in `ObjectStreamConstants`
  - `SC_WRITE_METHOD`
  - `SC_SERIALIZABLE`
  - `SC_EXTERNALIZABLE`

Storing Object Data

- Field descriptors
  - 1-byte type code
    - `J    long`
    - `Z    boolean`
    - `L    object`
    - `[`   array
    - Other primitives by first letter
  - 2-byte length
  - Field name
  - Class name (if an object)

`ObjectOutputStream`

- To write an object’s state to the stream, use the `ObjectOutputStream` class
  - Use `writeObject(Object o)` to save an object’s state to a stream
- Constructor takes any `OutputStream`
  - Most often wrapped around a `FileOutputStream`
  - `Remember the Decorator pattern?`

`ObjectInputStream`

- To read an object’s state from a stream, use the `ObjectInputStream` class
  - Use `readObject()` to read an object’s state from the stream
- Constructor takes any `InputStream`
  - Most often wrapped around a `FileInputStream`
- `readObject()` returns an `Object`; you must cast it to the appropriate type

Object Stream Exceptions

- Note which `Exceptions` have to be caught, and where
  - `IOException` for streams
  - `ClassNotFoundException` on `ObjectInputStream.readObject()`

Serialization Example

– Example: Serializing an Object

```
try {
    AnObject object = new AnObject();

    ObjectOutputStream out =
       new ObjectOutputStream(
          new FileOutputStream("filename.ser"));
    out.writeObject(object);
    out.close();
 }
 catch(IOException ex) {
    ex.printStackTrace();
 }
```

Deserialization Example

– Example: Deserializing an Object

```
try {
    ObjectInputStream in =
       new ObjectInputStream(
          new FileInputStream("filename.ser"));
    AnObject object = (AnObject)in.readObject();
    in.close();
 }
 catch(ClassNotFoundException ex) {
    ex.printStackTrace();
 }
 catch(IOException ex) {
    ex.printStackTrace();
 }
```

What Should not be Serialized

- What kinds of classes are not Serializable?
  - Streams
  - DB and network connections
  - Graphics contexts
  - Anything that can only be determined from current context
- If your class contains these types, you can still make your class serializable by marking them with the transient keyword:
  - `private transient FileInputStream file;`
- Transient fields are ignored during serialization

Custom Serialization

- But might doing this not mess up your class' state?
- You can control serialization beyond what is built in to the JDK
- Implement two methods in your class:

```
private void writeObject(ObjectOutputStream out)
private void readObject(ObjectInputStream in)
```

Custom Serialization (cont’d)

- Within custom methods you can use the default method of the stream, then do any needed additional processing

```
    private void readObject(ObjectInputStream in) {
            in.defaultReadObject();
            // Process calculated attributes here
            today = calendar.getTime();
    }
```

Ultimate Serialization Control

- Implement `Externalizable` (which extends `Serializable`)
- Defines two methods: `writeExternal()` and `readExternal()`
- These allow you to specify exactly how objects are stored to the underlying stream

Class Versioning

- Fingerprint is SHA computed 20 bytes
  - ~100% fingerprint changes if data is changed
  - Java uses only 8 bytes - still OK
  - Checks data and methods(!)
- Why worry?
  - If class layout is changed, original data may corrupt the memory
  - Class definition can be altered to hack into programs
  - This computation is expensive

Versioning

- Versioning is used for compatibility with older software
  - `serialver` tool in JDK
  - `static final long serialVersionID = 2121…21L;`
- Used instead of computing SHA fingerprint
- Dealing with different versions
  - Variable changes type
  - Variables added
  - Variables deleted

Object Versioning

- Each time an object is written, its serial version UID (unique identifier is written with it
- Attempting to deserialize an object saved with a different UID causes an InvalidClassException

Substituting Objects

- It is some time necessary to replace the object read from stream with one of your choosing
  - Singletons
  - Type safe-enumerations
- Classes may implement the `readResolve` method to return the "chosen" object

Singleton Example

```
public class ASingleton implements Serializable {
   private ASingleton theInstance;

   private ASingleton() {
   }

   public static ASingleton getInstance() {
      if (theInstance == null) {
         theInstance = new ASingleton();
      }
      return theInstance;
   }

   private Object readResolve() {
      return getInstance();
   }

   ...
}
```

Serialization Proxy

- Serializing a proxy instance prevents security problems from making instances
- Process
  - Define private static nested class representing enclosing classes state - the proxy
    - Implement `readResolve()` method to create and return an instance of the enclosing class
  - In the enclosing class
    - Implement `writeReplace()` method in enclosing class
    - Implement `readObject(ObjectInputStream)` to throw `InvalidObjectException` in enclosing class

Serialization Proxy Example

```
import java.io.InvalidObjectException;
import java.io.ObjectInputStream;
import java.io.Serializable;

public final class Cube implements Serializable {
    private double height;
    private double width;
    private double depth;

    public Cube(final double width, final double height, final double depth) {
        this.width = width;
        this.height = height;
        this.depth = depth;
    }

    public double getHeight() { return height; }

    public double getWidth() { return width; }

    public double getDepth() { return depth; }
```

Serialization Proxy Example

```java
    private static class SerializationProxy implements Serializable {
        private double x;
        private double y;
        private double z;

        SerializationProxy(final Cube cube) {
            y = cube.height;
            x = cube.width;
            z = cube.depth;
        }

        private Object readResolve() {
            return new Cube(x, y, z);
        }
    }

    private Object writeReplace() {
        return new SerializationProxy(this);
    }

    private void readObject(ObjectInputStream ois)
    throws InvalidObjectException {
        throw new InvalidObjectException("Proxy required");
    }
}
```