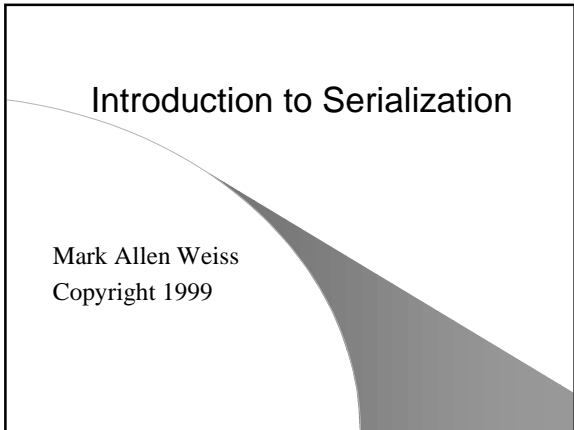


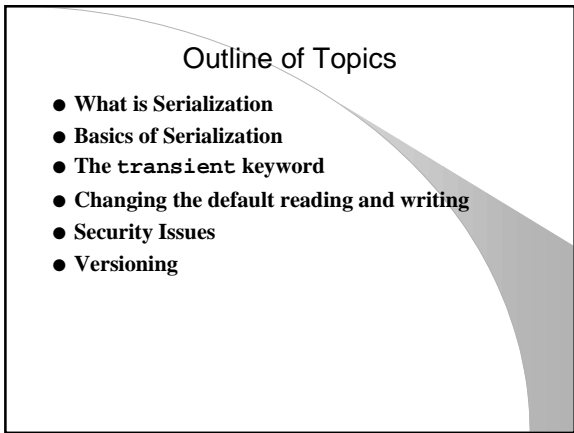
Introduction to Serialization

Mark Allen Weiss
Copyright 1999



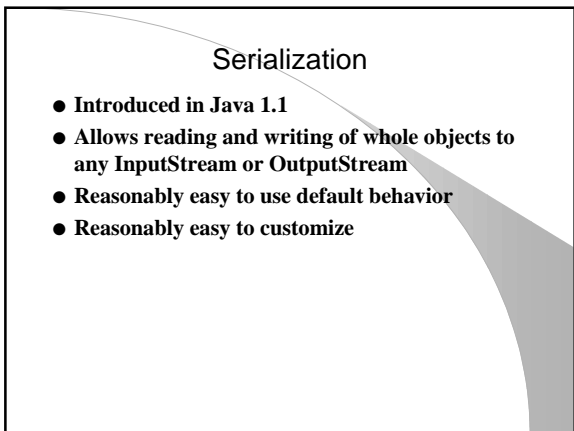
Outline of Topics

- What is Serialization
- Basics of Serialization
- The `transient` keyword
- Changing the default reading and writing
- Security Issues
- Versioning



Serialization

- Introduced in Java 1.1
- Allows reading and writing of whole objects to any `InputStream` or `OutputStream`
- Reasonably easy to use default behavior
- Reasonably easy to customize



Why Serialization?

- **Persistence: Storing objects in files and databases (persistence)**
- **Marshalling: Passing whole objects between VMs over the network**

Serializable Interface

- **Only objects that implement the `Serializable` interface can be serialized**
- **This interface is part of Java 1.1**
- **`String`, `Integer`, `Vector`, `Hashtable` and other typical classes have been retrofitted in Java 1.1 to be `Serializable`**
- **Easy to implement this interface: it has no methods!**

Why Isn't Everything Serializable?

- **If it was, there could be information leaks when objects are written.**
- **We also need to be careful when reading a serialized object (sketch of the details provided later)**

Writing an Object

- Create an `ObjectOutputStream` from any `OutputStream`
- Use its `writeObject` method:
`void writeObject(Object obj);`
- Remember: `obj` must be serializable.
- Need to handle `IOException`, `NotSerializableException`, `InvalidClassException`. All are `IOExceptions`.

Reading an Object

- Create an `ObjectInputStream` from any `InputStream`
- Use its `readObject` method:
`Object readObject();`
- Returned type must be downcast to actual type.
- Need to handle `IOException`, `ClassNotFoundException`, `OptionalDataException`. All but `ClassNotFoundException` are `IOExceptions`.

Example of Writing

```
FileOutputStream f;  
  
ObjectOutputStream out = new ObjectOutputStream( f );  
Person p = new Person( ... );  
...  
try {  
    out.writeObject( p );  
}  
catch( NotSerializableException e ) {  
    // Oops: Reservation is not serializable  
}  
catch( IOException e ) {  
    // Oops: Various I/O problems  
}
```

Example of Reading

```
FileInputStream f;  
  
ObjectInputStream out = new ObjectInStream( f );  
Person p;  
...  
try {  
    p = (Person) in.readObject( );  
}  
catch( ClassNotFoundException e ) {  
    // Oops: Reservation class not loadable on this VM  
}  
catch( IOException e ) {  
    // Oops: Various I/O problems  
}
```

What Gets Written

- **ObjectStreamClass** object for the object's class
 - Includes 64-bit `serialVersionUID` (versioning)
 - Names and types of fields
 - `ObjectStreamClass` of super class
- **Non-transient, non-static fields**
 - Base class data written first (if base class not serializable, zero-parameter constructor used)
 - Fields that are object references are followed; if they can be serialized, they are; otherwise, if null ok; otherwise exception is thrown.

Several References to Same Object

- When written, each object gets a serial number (hence, serialization)
- Each reference is then either null or the appropriate serial number
- Long chains of references are followed automatically using a depth-first search; saves you lots of manual labor
- When objects are restored, everything looks like it was.

What Gets Read

- `ObjectStreamClass` object for the object's class
 - Includes 64-bit `serialVersionUID` (versioning)
 - Names and types of fields
 - `ObjectStreamClass` of super class
- Object fields read in next
- If base class is nonserializable, inherited portion is initialized with no-parameter constructor.

Class Versioning

- 64 bit `serialVersionUID` is stored; it is based on the class members
- If the class changes in any significant way, the `serialVersionUID` changes too
- When object is read from an object stream, system ensures that version UIDs of `ObjectStreamClass` and local class are the same
- If not, exception is thrown

Version UIDs

- Computed using SHA (only 64 of the 160 bits are retained); can obtain it by running `serialver`, which is part of the JDK.
- Will change if you add or delete members or change signatures (even from public to private)
- You can declare your opinion that a class is compatible by adding a `private static final long` member named `serialVersionUID`.

Version UIDs (continued)

- If UIDs match, attempt to load class will continue
- During the read, new fields (those not written by the write) will get default values (null for references, etc.)
- Removed fields will be ignored
- If VM isn't happy, an exception is thrown. May happen if field type has changed

Security Problems

- The constructor for the `Serializable` object is NOT called.
 - Some fields may be null, which might not be what was expected based on constructors
- `Serializable` object is read in without checks. Consider a serializable `Date` class.
 - Can write the `Date` out to a file
 - Can edit the file, and generate an illegal date
 - When it is read in, fields are copied, blindly
 - There's no check, and `Date` is now inconsistent
 - Don't even need files: can use `ByteStreams`.

Customizing `readObject`

- Can have `readObject` do extra work by implementing (in the serialized object's class)

```
private void readObject( ObjectInputStream in )  
    throws IOException, ClassNotFoundException;
```
- `readObject` (and only `readObject`) can call an `ObjectInputStream` method to do the normal default read, and can then add code to do error checks or other additional work. The method call would be
 - `in.defaultReadObject();`
- Note: `readObject` must be private

Example: Making Date Safer

```
public class Date implements Serializable
{
    ...
    private boolean isValid( )
        { /* implementation omitted */ }

    private void readObject( ObjectInputStream in )
        throws IOException, ClassNotFoundException
    {
        in.defaultReadObject( );
        if( !isValid( ) ) throw new IOException( );
    }
}
```

Customizing writeObject

- Not surprisingly, can do similar things with writeObject
 - Can have private writeObject method (throws IOException only)
 - Private writeObject method can call out.defaultWriteObject()
- This is useful for writing output in a special way. For instance, can write a transient field in an encrypted format. Of course, need to implement readObject to match.

readObject and writeObject

```
public class Secure implements Serializable {
    transient private String password;
    private String name;

    private void writeObject( ObjectOutputStream out )
        throws IOException {
        out.defaultWriteObject( );
        out.writeUTF( NSA.encrypt( password ) );
    }

    private void readObject( ObjectInputStream is )
        throws IOException, ClassNotFoundException {
        in.defaultReadObject( );
        password = NSA.decrypt( in.readUTF( ) );
    }
}
```

Dealing With Changing Fields

- Uses nested class
`ObjectInputStream.GetField`
- Call `ObjectInputStream.readFields` to get a collection of name/value pairs
 - unfortunately, cannot enumerate over collection
 - must know exact field name in stream format
- Various `get` methods extract objects
 - Two parameters: name of field, and value to return if field is not in the stream (but is in current class)
 - if field not in the stream AND field is not in the current class, you get an exception
 - Must explicitly assign values to all class fields, or they will get zero for primitives, null for references

Using `ObjectInputStream.GetField`

```
class Person {
    static final long serialVersionUID = ...;
    ...
    private void readObject( ObjectInputStream in )
        throws IOException, ClassNotFoundException {
        ObjectInputStream.GetField gf = in.readFields( );

        fullName = (String) gf.get( "fullName", null );
        if( fullName == null )
            fullName = gf.get( "lastName", null )
                + ", " + gf.get( "firstName", null );
        age = gf.get( "age", 0 );
    }

    private String fullName;    // the new version
    // private String lastName; // the old version
    // private String firstName; // the old version
}
```

Externalization

- `Serializable` is nice, but has some overhead.
May not be appropriate in all instances.
- The `Externalizable` interface extends `Serializable`. Must implement
 - `public void writeExternal(ObjectOutput out)`
throws `IOException`;
 - `public void readExternal(ObjectInput in)`
throws `IOException`, `ClassNotFoundException`;
- Your object has complete control over how it is written but you deal with base classes, and serializing object references.
- Object and all super classes must have 0-param constructor

Externalizable Date Class

```
class Date implements Externalizable
{
    public Date( )
    { this( 1, 1, 2000 ); }

    public Date( int m, int d, int y )
    { month = m; date = d; year = y; }

    public void writeExternal( ObjectOutputStream out ) throws IOException
    { out.writeInt( month ); out.writeInt( date ); out.writeInt( year ); }

    public void readExternal( ObjectInputStream in )
    throws IOException, ClassNotFoundException
    { month = in.readInt( ); date = in.readInt( ); year = in.readInt( ); }

    private int month;
    private int date;
    private int year;
}
```

Summary

- **Serialization is easy to use; it tracks down and serializes all objects reachable from the basic object being serialized**
- **Usually have to implement private readObject to do security check**
- **May need to watch out for class version changes and set the serialVersionUID**
- **Externalization is hard to use unless class is trivial, but can be faster and more compact**
