Versioning of Serializable Objects

CHAPTER 5

Topics:

- Overview
- Goals
- Assumptions
- Who's Responsible for Versioning of Streams
- Compatible Java Type Evolution
- Type Changes Affecting Serialization

5.1 Overview

When Java objects use serialization to save state in files, or as blobs in databases, the potential arises that the version of a class reading the data is different than the version that wrote the data.

Versioning raises some fundamental questions about the identity of a class, including what constitutes a compatible change. A ***compatible change*** is a change that does not affect the contract between the class and its callers.

This section describes the goals, assumptions, and a solution that attempts to address this problem by restricting the kinds of changes allowed and by carefully choosing the mechanisms.

The proposed solution provides a mechanism for "automatic" handling of classes that evolve by adding fields and adding classes. Serialization will handle versioning without class-specific methods to be implemented for each version. The stream format can be traversed without invoking class-specific methods.

5.2 Goals

The goals are to:

- Support bidirectional communication between different versions of a class operating in different virtual machines by:

    - Defining a mechanism that allows Java classes to read streams written by older versions of the same class.
    - Defining a mechanism that allows Java classes to write streams intended to be read by older versions of the same class.
- Provide default serialization for persistence and for RMI.
- Perform well and produce compact streams in simple cases, so that RMI can use serialization.
- Be able to identify and load classes that match the exact class used to write the stream.
- Keep the overhead low for nonversioned classes.
- Use a stream format that allows the traversal of the stream without having to invoke methods specific to the objects saved in the stream.

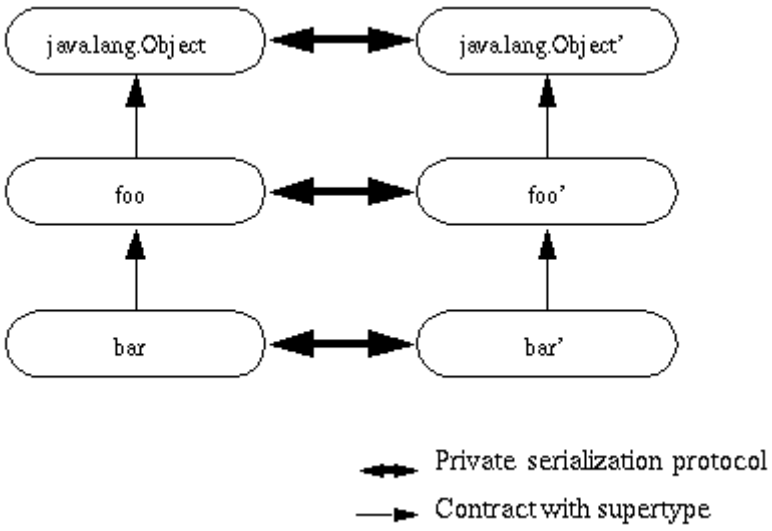
5.3 Assumptions

The assumptions are that:

- Versioning will only apply to serializable classes since it must control the stream format to achieve it goals. Externalizable classes will be responsible for their own versioning which is tied to the external format.
- All data and objects must be read from, or skipped in, the stream in the same order as they were written.
- Classes evolve individually as well as in concert with supertypes and subtypes.
- Classes are identified by name. Two classes with the same name may be different versions or completely different classes that can be distinguished only by comparing their interfaces or by comparing hashes of the interfaces.
- Default serialization will not perform any type conversions.
- The stream format only needs to support a linear sequence of type changes, not arbitrary branching of a type.

5.4 Who's Responsible for Versioning of Streams

In the evolution of classes, it is the responsibility of the evolved (later version) class to maintain the contract established by the nonevolved class. This takes two forms. First, the evolved class must not break the existing assumptions about the interface provided by the original version, so that the evolved class can be used in place of the original. Secondly, when communicating with the original (or previous) versions, the evolved class must provide sufficient and equivalent information to allow the earlier version to continue to satisfy the nonevolved contract.



For the purposes of the discussion here, each class implements and extends the interface or contract defined by its supertype. New versions of a class, for example `foo'`, must continue to satisfy the contract for `foo` and may extend the interface or modify its implementation.

Communication between objects via serialization is not part of the contract defined by these interfaces. Serialization is a private protocol between the implementations. It is the responsibility of the implementations to communicate sufficiently to allow each implementation to continue to satisfy the contract expected by its clients.

5.5 Compatible Java Type Evolution

The Java Language Specification discusses binary compatibility of Java classes as those classes evolve. Most of the flexibility of binary compatibility comes from the use of late binding of symbolic references for the names of classes, interfaces, fields, methods, and so on.

The following are the principle aspects of the design for versioning of serialized object streams.

- The default serialization mechanism will use a symbolic model for binding the fields in the stream to the fields in the corresponding class in the virtual machine.
- Each class referenced in the stream will uniquely identify itself, its supertype, and the types and names of each serializable field written to the stream. The fields are ordered with the primitive types first sorted by field name, followed by the object fields sorted by field name.
- Two types of data may occur in the stream for each class: required data (corresponding directly to the serializable fields of the object); and optional data (consisting of an arbitrary sequence of primitives and objects). The stream format defines how the required and optional data occur in the stream so that the whole class, the required, or the optional parts can be skipped if necessary.
    - The required data consists of the fields of the object in the order defined by the class descriptor.
    - The optional data is written to the stream and does not correspond directly to fields of the class. The class itself is responsible for the length, types, and versioning of this optional information.
- If defined for a class, the `writeObject`/`readObject` methods supersede the default mechanism to write/read the state of the class. These methods write and read the optional

data for a class. The required data is written by calling `defaultWriteObject` and read by calling `defaultReadObject`.
- The stream format of each class is identified by the use of a Stream Unique Identifier (SUID). By default, this is the hash of the class. All later versions of the class must declare the Stream Unique Identifier (SUID) that they are compatible with. This guards against classes with the same name that might inadvertently be identified as being versions of a single class.
- Subtypes of `ObjectOutputStream` and `ObjectInputStream` may include their own information identifying the class using the `annotateClass` method; for example, `MarshalOutputStream` embeds the URL of the class.

5.6 Type Changes Affecting Serialization

With these concepts, we can now describe how the design will cope with the different cases of an evolving class. The cases are described in terms of a stream written by some version of a class. When the stream is read back by the same version of the class, there is no loss of information or functionality. The stream is the only source of information about the original class. Its class descriptions, while a subset of the original class description, are sufficient to match up the data in the stream with the version of the class being reconstituted.

The descriptions are from the perspective of the stream being read in order to reconstitute either an earlier or later version of the class. In the parlance of RPC systems, this is a "receiver makes right" system. The writer writes its data in the most suitable form and the receiver must interpret that information to extract the parts it needs and to fill in the parts that are not available.

5.6.1 Incompatible Changes

Incompatible changes to classes are those changes for which the guarantee of interoperability cannot be maintained. The incompatible changes that may occur while evolving a class are:

- Deleting fields - If a field is deleted in a class, the stream written will not contain its value. When the stream is read by an earlier class, the value of the field will be set to the default value because no value is available in the stream. However, this default value may adversely impair the ability of the earlier version to fulfill its contract.
- Moving classes up or down the hierarchy - This cannot be allowed since the data in the stream appears in the wrong sequence.
- Changing a nonstatic field to static or a nontransient field to transient - When relying on default serialization, this change is equivalent to deleting a field from the class. This version of the class will not write that data to the stream, so it will not be available to be read by earlier versions of the class. As when deleting a field, the field of the earlier version will be initialized to the default value, which can cause the class to fail in unexpected ways.
- Changing the declared type of a primitive field - Each version of the class writes the data with its declared type. Earlier versions of the class attempting to read the field will fail because the type of the data in the stream does not match the type of the field.
- Changing the `writeObject` or `readObject` method so that it no longer writes or reads the default field data or changing it so that it attempts to write it or read it when the previous version did not. The default field data must consistently either appear or not appear in the stream.

- Changing a class from `Serializable` to `Externalizable` or vice versa is an incompatible change since the stream will contain data that is incompatible with the implementation of the available class.
- Changing a class from a non-enum type to an enum type or vice versa since the stream will contain data that is incompatible with the implementation of the available class.
- Removing either `Serializable` or `Externalizable` is an incompatible change since when written it will no longer supply the fields needed by older versions of the class.
- Adding the `writeReplace` or `readResolve` method to a class is incompatible if the behavior would produce an object that is incompatible with any older version of the class.

5.6.2 Compatible Changes

The compatible changes to a class are handled as follows:

- Adding fields - When the class being reconstituted has a field that does not occur in the stream, that field in the object will be initialized to the default value for its type. If class-specific initialization is needed, the class may provide a readObject method that can initialize the field to nondefault values.
- Adding classes - The stream will contain the type hierarchy of each object in the stream. Comparing this hierarchy in the stream with the current class can detect additional classes. Since there is no information in the stream from which to initialize the object, the class's fields will be initialized to the default values.
- Removing classes - Comparing the class hierarchy in the stream with that of the current class can detect that a class has been deleted. In this case, the fields and objects corresponding to that class are read from the stream. Primitive fields are discarded, but the objects referenced by the deleted class are created, since they may be referred to later in the stream. They will be garbage-collected when the stream is garbage-collected or reset.
- Adding `writeObject`/`readObject` methods - If the version reading the stream has these methods then `readObject` is expected, as usual, to read the required data written to the stream by the default serialization. It should call `defaultReadObject` first before reading any optional data. The `writeObject` method is expected as usual to call `defaultWriteObject` to write the required data and then may write optional data.
- Removing `writeObject`/`readObject` methods - If the class reading the stream does not have these methods, the required data will be read by default serialization, and the optional data will be discarded.
- Adding `java.io.Serializable` - This is equivalent to adding types. There will be no values in the stream for this class so its fields will be initialized to default values. The support for subclassing nonserializable classes requires that the class's supertype have a no-arg constructor and the class itself will be initialized to default values. If the no-arg constructor is not available, the `InvalidClassException` is thrown.
- Changing the access to a field - The access modifiers public, package, protected, and private have no effect on the ability of serialization to assign values to the fields.
- Changing a field from static to nonstatic or transient to nontransient - When relying on default serialization to compute the serializable fields, this change is equivalent to adding a field to the class. The new field will be written to the stream but earlier classes will ignore the value since serialization will not assign values to static or transient fields.

CONTENTS | PREV | NEXT
