Network Working Group
Request for Comments: 2713
Category: Informational

V. Ryan
S. Seligman
R. Lee
Sun Microsystems, Inc.

October 1999

Schema for Representing Java(tm) Objects in an LDAP Directory

Status of this Memo

Copyright Notice

Abstract

This document defines the schema for representing Java(tm) objects in an LDAP directory [LDAPv3]. It defines schema elements to represent a Java serialized object [Serial], a Java marshalled object [RMI], a Java remote object [RMI], and a JNDI reference [JNDI].

1. Introduction

This document assumes that the reader has a general knowledge of the Java programming language [Java]. For brevity we use the term "Java object" in place of "object in the Java programming language" throughout this text.

Traditionally, LDAP directories have been used to store data. Users and programmers think of the directory as a hierarchy of directory entries, each containing a set of attributes. You look up an entry from the directory and extract the attribute(s) of interest. For example, you can look up a person’s telephone number from the directory. Alternatively, you can search the directory for entries with a particular set of attributes. For example, you can search for all persons in the directory with the surname "Smith".

For applications written in the Java programming language, a kind of data that is typically shared are Java objects themselves. For such applications, it makes sense to be able to use the directory as a repository for Java objects. The directory provides a centrally administered, and possibly replicated, service for use by Java applications distributed across the network.

For example, an application server might use the directory for "registering" objects representing the services that it manages, so that a client can later search the directory to locate those services as it needs.

The motivation for this document is to define a common way for applications to store and retrieve Java objects from the directory. Using this common schema, any Java application that needs to read or store Java objects in the directory can do so in an interoperable way.

2 Representation of Java Objects

This document defines schema elements to represent three types of Java objects: a Java serialized object, a Java marshalled object, and a JNDI reference. A Java remote object is stored as either a Java marshalled object or a JNDI reference.

2.1 Common Representations

A Java object is stored in the LDAP directory by using the object class javaObject. This is the base class from which other Java object related classes derive: javaSerializedObject, javaMarshalledObject, and javaNamingReference. javaObject is an abstract object class, which means that a javaObject cannot exist by itself in the directory; only auxiliary or structural subclasses of it can exist in the directory.

The object class javaContainer represents a directory entry dedicated to storing a Java object. It is a structural object class. In cases where a subclass of javaObject is mixed in with another structural object class, javaContainer is not required.

The definitions for the object classes javaObject and javaContainer are presented in Section 4.

The javaObject class has one mandatory attribute (javaClassName) and four optional attributes (javaClassNames, javaCodebase, javaDoc, description). javaClassName is a single valued attribute that is used to store the fully qualified name of the object's Java class (for example, "java.lang.String"). This may be the object's most derived class's name, but does not have to be; that of a superclass or interface in some cases might be most appropriate. This attribute is intended for storing the name of the object's "distinguished" class, that is, the class or interface with which the object should be identified.

javaClassNames is a multivalued attribute that is used to store the fully qualified names of the object's Java classes and interfaces (for example, "java.lang.Byte"). Like all multivalued attributes, the javaClassNames attribute's values are unordered and so no one value is more "distinguished" than the others. This attribute is intended for storing an object's class and interface names and those of its ancestor classes and interfaces, although the list of values does not have to be complete. If the javaClassNames attribute is present, it should include the value of javaClassName.

For example, suppose an object is stored in the directory with a javaClassName attribute of "java.io.FilePermission", and a javaClassNames attribute of {"java.security.Permission", "java.io.FilePermission", "java.security.Guard", "java.io.Serializable"}. An application searching a directory for Java objects might use javaClassName to produce a summary of the names and types of Java objects in that directory. Another application might use the javaClassNames attribute to find, for example, all java.security.Permission objects.

javaCodebase is a multivalued attribute that is used to store the location(s) of the object's class definition. javaDoc is used to store a pointer (URL) to the Java documentation for the class. description is used to store a textual description of a Java object and is defined in [v3Schema]. The definitions of these attributes are presented in Section 3.

2.2 Serialized Objects

To "serialize" an object means to convert its state into a byte stream in such a way that the byte stream can be converted back into a copy of the object. A Java object is "serializable" if its class or any of its superclasses implements either the java.io.Serializable interface or its subinterface java.io.Externalizable. "Deserialization" is the process of converting the serialized form of an object back into a copy of the object. When an object is serialized, the entire tree of objects rooted at the object is also serialized. When it is deserialized, the tree is reconstructed. For example, suppose a serializable Book object contains (a serializable field of) an array of Page objects. When a Book object is serialized, so is the array of Page objects.

The Java platform specifies a default algorithm by which serializable objects are serialized. A Java class can also override this default serialization with its own algorithm. [Serial] describes object serialization in detail.

When an object is serialized, information that identifies its class is recorded in the serialized stream. However, the class's definition ("class file") itself is not recorded. It is the responsibility of the system that is deserializing the object to determine the mechanism to use for locating and loading the associated class definitions. For example, the Java application might include in its classpath a JAR file containing the class definitions of the serialized object, or load the class definitions using information from the directory, as explained below.

2.2.1 Representation in the Directory

A serialized object is represented in the directory by the attributes javaClassName, javaClassNames, javaCodebase, and javaSerializedData, as defined in Section 3. The mandatory attribute, javaSerializedData, contains the serialized form of the object. Although the serialized form already contains the class name, the mandatory javaClassName attribute also records the class name of the serialized object so that applications can determined class information without having to first deserialize the object. The optional javaClassNames attribute is used to record additional class information about the serialized object. The optional javaCodebase attribute is used to record the locations of the class definitions needed to deserialize the serialized object.

A directory entry that contains a serialized object is represented by the object class javaSerializedObject, which is a subclass of javaObject. javaSerializedObject is an auxiliary object class, which means that it needs to be mixed in with a structural object class. javaSerializedObject's definition is given in Section 4.

2.3 Marshalled Objects

To "marshal" an object means to record its state and codebase(s) in such a way that when the marshalled object is "unmarshalled," a copy of the original object is obtained, possibly by automatically loading the class definitions of the object. You can marshal any object that is serializable or remote (that is, implements the java.rmi.Remote interface). Marshalling is like serialization, except marshalling also records codebases. Marshalling is different from serialization in that marshalling treats remote objects specially. If an object is a java.rmi.Remote object, marshalling records the remote object's "stub" (see Section 2.5), instead of the remote object itself. Like serialization, when an object is marshalled, the entire tree of objects rooted at the object is marshalled. When it is unmarshalled, the tree is reconstructed.

A "marshalled" object is the represented by the java.rmi.MarshalledObject class. Here's an example of how to create MarshalledObjects for serializable and remote objects:

```
java.io.Serializable sobj = ...;
java.rmi.MarshalledObject mobj1 =
    new java.rmi.MarshalledObject(sobj);

java.rmi.Remote robj = ...;
java.rmi.MarshalledObject mobj2 =
    new java.rmi.MarshalledObject(robj);
```

Then, to retrieve the original objects from the MarshalledObjects, do as follows:

```
java.io.Serializable sobj = (java.io.Serializable) mobj1.get();
java.io.Remote rstub = (java.io.Remote) mobj2.get();
```

MarshalledObject is available only on the Java 2 Platform, Standard Edition, v1.2, and higher releases.

2.3.1 Representation in the Directory

A marshalled object is represented in the directory by the attributes javaClassName, javaClassNames, and javaSerializedData, as defined in Section 3. The mandatory attribute, javaSerializedData, contains the serialized form of the marshalled object (that is, the serialized form of a MarshalledObject instance). The mandatory javaClassName attribute records the distinguished class name of the object before it has been marshalled. The optional javaClassNames attribute is used to record additional class information about the object before it has been marshalled.

A directory entry that contains a marshalled object is represented by the object class javaMarshalledObject, which is a subclass of javaObject. javaMarshalledObject is an auxiliary object class, which means that it needs to be mixed in with a structural object class. javaMarshalledObject's definition is given in Section 4.

As evident in this description, a javaMarshalledObject differs from a javaSerializedObject only in the interpretation of the javaClassName and javaClassNames attributes.

2.4 JNDI References

Java Naming and Directory Interface(tm) (JNDI) is a directory access API specified in the Java programming language [JNDI]. It provides an object-oriented view of the directory, allowing Java objects to be added to and retrieved from the directory without requiring the client to manage data representation issues.

JNDI defines the notion of a "reference" for use when an object cannot be stored in the directory directly, or when it is inappropriate or undesirable to do so. An object with an associated reference is stored in the directory indirectly, by storing its reference instead.

2.4.1 Contents of a Reference

A JNDI reference is a Java object of class javax.naming.Reference. It consists of class information about the object being referenced and an ordered list of addresses. An address is a Java object of class javax.naming.RefAddr. Each address contains information on how to construct the object.

A common use for JNDI references is to represent connections to a network service such as a database, directory, or file system. Each address may then identify a "communications endpoint" for that service, containing information on how to contact the service. Multiple addresses may arise for various reasons, such as replication or the object offering interfaces over more than one communication mechanism.

A reference also contains information to assist in the creation of an instance of the object to which the reference refers. It contains the Java class name of that object, and the class name and location of the object factory to be used to create the object. The procedures for creating an object given its reference and the reverse are described in [JNDI].

2.4.2 Representation in the Directory

A JNDI reference is stored in the directory by using the attributes javaClassName, javaClassNames, javaCodebase, javaReferenceAddress, and javaFactory, defined in Section 3. These attributes store information corresponding to the contents of a reference described above. javaReferenceAddress is a multivalued optional attribute for storing reference addresses. javaFactory is the optional attribute for storing the object factory’s fully qualified class name. The mandatory javaClassName attribute is used to store the name of the distinguished class of the object. The optional javaClassNames

attribute is used to record additional class and interface names. The optional javaCodebase attribute is used to store the locations of the object factory's and the object's class definitions.

A directory entry containing a JNDI reference is represented by the object class javaNamingReference, which is a subclass of javaObject. javaNamingReference is an auxiliary object class, which means that it needs to be mixed in with a structural object class. javaNamingReference's definition is given in Section 4.

2.5 Remote Objects

The Java Remote Method Invocation (RMI) system [RMI] is a mechanism that enables an object on one Java virtual machine to invoke methods on an object in another Java virtual machine. Any object whose methods can be invoked in this way must implement the java.rmi.Remote interface. When such an object is invoked, its arguments are marshalled and sent from the local virtual machine to the remote one, where the arguments are unmarshalled and used. When the method terminates, the results are marshalled from the remote machine and sent to the caller's virtual machine.

To make a remote object accessible to other virtual machines, a program typically registers it with the RMI registry. The program supplies to the RMI registry the string name of the remote object and the remote object itself. When a program wants to access a remote object, it supplies the object's string name to the RMI registry on the same machine as the remote object. The RMI registry returns to the caller a reference (called "stub") to the remote object. When the program receives the stub for the remote object, it can invoke methods on the remote object (through the stub). A program can also obtain references to remote objects as a result of remote calls to other remote objects or from other naming services. For example, the program can look up a reference to a remote object from an LDAP server that supports the schema defined in this document.

The string name accepted by the RMI registry has the syntax "rmi://hostname:port/remoteObjectName", where "hostname" and "port" identify the machine and port on which the RMI registry is running, respectively, and "remoteObjectName" is the string name of the remote object. "hostname", "port", and the prefix, "rmi:", are optional. If "hostname" is not specified, it defaults to the local host. If "port" is not specified, it defaults to 1099. If "remoteObjectName" is not specified, then the object being named is the RMI registry itself. See [RMI] for details.

RMI can be supported using different protocols: the Java Remote Method Protocol (JRMP) and the Internet Inter-ORB Protocol (IIOP). The JRMP is a specialized protocol designed for RMI; the IIOP is the standard protocol for communication between CORBA objects [CORBA]. RMI over IIOP allows Java remote objects to communicate with CORBA objects which might be written in a non-Java programming language [RMI-IIOP].

2.5.1 Representation in the Directory

Remote objects that use the IIOP are represented in the directory as CORBA object references [CORBA-LDAP]. Remote objects that use the JRMP are represented in the directory in one of two ways: as a marshalled object, or as a JNDI reference.

A marshalled object records the codebases of the remote object's stub and any serializable or remote objects that it references, and replaces remote objects with their stubs. To store a Remote object as a marshalled object (java.rmi.MarshalledObject), you first create a java.rmi.MarshalledObject instance for it.

```
java.rmi.Remote robj = ...;
java.rmi.MarshalledObject mobj =
    new java.rmi.MarshalledObject(robj);
```

You can then store the MarshalledObject instance as a javaMarshalledObject. The javaClassName attribute should contain the fully qualified name of the distinguished class of the remote object. The javaClassNames attribute should contain the names of the classes and interfaces of the remote object. To read the remote object back from the directory, first deserialize the contents of the javaSerializedData to get a MarshalledObject (mobj), then retrieve it from the MarshalledObject as follows:

```
java.rmi.Remote robj = (java.rmi.Remote)mobj.get();
```

This returns the remote stub, which you can then use to invoke remote methods.

MarshalledObject is available only on the Java 2 Platform, Standard Edition, v1.2 and higher releases. Therefore, a remote object stored as a MarshalledObject can only be read by clients using the the Java 2 Platform, Standard Edition, v1.2 or higher releases.

To store a remote object as a JNDI reference, you first create a javax.naming.Reference object instance for it using the remote object's string name as it has been, or will be, recorded with the RMI registry, with the additional restriction that the "rmi:" prefix must be present. Here's an example:

```
javax.naming.Reference ref = new javax.naming.Reference(
  obj.getClass().getName(),
  new javax.naming.StringRefAddr("URL",
      "rmi://rserver/AppRemoteObjectX"));
```

You then store the javax.naming.Reference instance as a javaNamingReference. The advantage of using a JNDI reference is that this can be done without a reference to the remote object. In fact, the remote object does not have to exist at the time that this recording in the directory is made. The remote object needs to exist and be bound with the RMI registry when the object is looked up from the directory.

2.6 Serialized Objects Vs. Marshalled Objects Vs. References

The object classes defined in this document store different aspects of the Java objects.

A javaSerializedObject or a serializable object stored as a javaMarshalledObject represents the object itself, while a javaNamingReference or a remote object stored as a javaMarshalledObject represents a "pointer" to the object.

When storing a serializable object in the directory, you have a choice of storing it as a javaSerializedObject or a javaMarshalledObject. The javaSerializedObject object class provides the basic way in which to store serializable objects. When you create an LDAP entry using the javaSerializableObject object class, you must explicitly set the javaCodebase attribute if you want readers of that entry to know where to load the class definitions of the object. When you create an LDAP entry using the javaMarshalledObject object class, you use the MarshalledObject class. The MarshalledObject class uses the RMI infrastructure available on the Java platform to automate how codebase information is gathered and recorded, thus freeing you from having to set the javaCodebase attribute. On the other hand, the javaCodebase attribute is human-readable and can be updated easily by using text-based tools without having to change other parts of the entry. This allows you, for instance, to move the class definitions to another location and then update the javaCodebase attribute to reflect the move without having to update the serialized object itself.

A javaNamingReference provides a way of recording address information about an object which itself is not directly stored in the directory. A remote object stored as a javaMarshalledObject also records address information (the object's "stub") of an object which itself is not directory stored in the directory. In other words, you can think of these as compact representations of the information required to access the object.

A javaNamingReference typically consists of a small number of human-readable strings. Standard text-based tools for directory administration may therefore be used to add, read, or modify reference entries -- if so desired -- quite easily. Serialized and marshalled objects are not intended to be read or manipulated directly by humans.

3 Attribute Type Definitions

The following attribute types are defined in this document:

javaClassName
javaClassNames
javaCodebase
javaSerializedData
javaFactory
javaReferenceAddress
javaDoc

3.1 javaClassName

This attribute stores the fully qualified name of the Java object's "distinguished" class or interface (for example, "java.lang.String"). It is a single-valued attribute. This attribute's syntax is 'Directory String' and its case is significant.

```
( 1.3.6.1.4.1.42.2.27.4.1.6
  NAME 'javaClassName'
  DESC 'Fully qualified name of distinguished Java class or
        interface'
  EQUALITY caseExactMatch
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
  SINGLE-VALUE
)
```

3.2 javaCodebase

This attribute stores the Java class definition's locations. It specifies the locations from which to load the class definition for the class specified by the javaClassName attribute. Each value of the attribute contains an ordered list of URLs, separated by spaces. For example, a value of "url1 url2 url3" means that the three (possibly interdependent) URLs (url1, url2, and url3) form the codebase for loading in the Java class definition.

If the javaCodebase attribute contains more than one value, each value is an independent codebase. That is, there is no relationship between the URLs in one value and those in another; each value can be viewed as an alternate source for loading the Java class definition. See [Java] for information regarding class loading.

This attribute's syntax is 'IA5 String' and its case is significant.

```
( 1.3.6.1.4.1.42.2.27.4.1.7
  NAME 'javaCodebase'
  DESC 'URL(s) specifying the location of class definition'
  EQUALITY caseExactIA5Match
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.26
)
```

3.3 javaClassNames

This attribute stores the Java object's fully qualified class or interface names (for example, "java.lang.String"). It is a multivalued attribute. When more than one value is present, each is the name of a class or interface, or ancestor class or interface, of this object.

This attribute's syntax is 'Directory String' and its case is significant.

```
( 1.3.6.1.4.1.42.2.27.4.1.13
  NAME 'javaClassNames'
  DESC 'Fully qualified Java class or interface name'
  EQUALITY caseExactMatch
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
)
```

3.4 javaSerializedData

This attribute stores the serialized form of a Java object. The serialized form is described in [Serial].

This attribute's syntax is 'Octet String'.

```
( 1.3.6.1.4.1.42.2.27.4.1.8
  NAME 'javaSerializedData
  DESC 'Serialized form of a Java object'
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.40
  SINGLE-VALUE
)
```

3.5 javaFactory

This attribute stores the fully qualified class name of the object factory (for example, "com.wiz.jndi.WizObjectFactory") that can be used to create an instance of the object identified by the javaClassName attribute.

This attribute's syntax is 'Directory String' and its case is significant.

```
( 1.3.6.1.4.1.42.2.27.4.1.10
  NAME 'javaFactory'
  DESC 'Fully qualified Java class name of a JNDI object factory'
  EQUALITY caseExactMatch
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
  SINGLE-VALUE
)
```

3.6 javaReferenceAddress

This attribute represents the sequence of addresses of a JNDI reference. Each of its values represents one address, a Java object of type javax.naming.RefAddr. Its value is a concatenation of the address type and address contents, preceded by a sequence number (the order of addresses in a JNDI reference is significant). For example:

```
#0#TypeA#ValA
#1#TypeB#ValB
#2#TypeC##rO0ABXNyABpq...
```

In more detail, the value is encoded as follows:

The delimiter is the first character of the value. For readability the character '#' is recommended when it is not otherwise used anywhere in the value, but any character may be used subject to restrictions given below.

The first delimiter is followed by the sequence number. The sequence number of an address is its position in the JNDI reference, with the first address being numbered 0. It is represented by its shortest string form, in decimal notation.

The sequence number is followed by a delimiter, then by the address type, and then by another delimiter. If the address is of Java class javax.naming.StringRefAddr, then this delimiter is followed by the value of the address contents (which is a string). Otherwise, this delimiter is followed immediately by another delimiter, and then by the Base64 encoding of the serialized form of the entire address.

The delimiter may be any character other than a digit or a character contained in the address type. In addition, if the address contents is a string, the delimiter may not be the first character of that string.

This attribute's syntax is 'Directory String' and its case is significant. It can contain multiple values.

```
( 1.3.6.1.4.1.42.2.27.4.1.11
  NAME 'javaReferenceAddress'
  DESC 'Addresses associated with a JNDI Reference'
  EQUALITY caseExactMatch
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
)
```

3.7 javaDoc

This attribute stores a pointer to the Java documentation for the class. It's value is a URL. For example, the following URL points to the specification of the java.lang.String class: http://java.sun.com/products/jdk/1.2/docs/api/java/lang/String.html

This attribute's syntax is 'IA5 String' and its case is significant.

```
( 1.3.6.1.4.1.42.2.27.4.1.12
  NAME 'javaDoc'
  DESC 'The Java documentation for the class'
  EQUALITY caseExactIA5Match
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.26
)
```

4 Object Class Definitions

The following object classes are defined in this document:

javaContainer
javaObject
javaSerializedObject
javaMarshalledObject
javaNamingReference

4.1 javaContainer

This structural object class represents a container for a Java object.

```
( 1.3.6.1.4.1.42.2.27.4.2.1
  NAME 'javaContainer'
  DESC 'Container for a Java object'
  SUP top
  STRUCTURAL
  MUST ( cn )
)
```

4.2 javaObject

This abstract object class represents a Java object. A javaObject cannot exist in the directory; only auxiliary or structural subclasses of it can exist in the directory.

```
( 1.3.6.1.4.1.42.2.27.4.2.4
  NAME 'javaObject'
  DESC 'Java object representation'
  SUP top
  ABSTRACT
  MUST ( javaClassName )
  MAY ( javaClassNames $
        javaCodebase $
        javaDoc $
        description )
)
```

4.3 javaSerializedObject

This auxiliary object class represents a Java serialized object. It must be mixed in with a structural object class.

```
( 1.3.6.1.4.1.42.2.27.4.2.5
  NAME 'javaSerializedObject'
  DESC 'Java serialized object'
  SUP javaObject
  AUXILIARY
  MUST ( javaSerializedData )
)
```

4.4 javaMarshalledObject

This auxiliary object class represents a Java marshalled object. It must be mixed in with a structural object class.

```
( 1.3.6.1.4.1.42.2.27.4.2.8
  NAME 'javaMarshalledObject'
  DESC 'Java marshalled object'
  SUP javaObject
  AUXILIARY
  MUST ( javaSerializedData )
)
```

4.5 javaNamingReference

This auxiliary object class represents a JNDI reference. It must be mixed in with a structural object class.

```
( 1.3.6.1.4.1.42.2.27.4.2.7
  NAME 'javaNamingReference'
  DESC 'JNDI reference'
  SUP javaObject
  AUXILIARY
  MAY ( javaReferenceAddress $
        javaFactory )
)
```

5. Security Considerations

Serializing an object and storing it into the directory enables (a copy of) the object to be examined and used outside the environment in which it was originally created. The directory entry containing the serialized object could be read and modified within the constraints imposed by the access control mechanisms of the directory. If an object contains sensitive information or information that could be misused outside of the context in which it was created, the object should not be stored in the directory. For more details on security issues relating to serialization in general, see [Serial].

6. Acknowledgements

We would like to thank Joseph Fialli, Peter Jones, Roger Riggs, Bob Scheifler, and Ann Wollrath of Sun Microsystems for their comments and suggestions.

7. References

[CORBA] The Object Management Group, "Common Object Request Broker Architecture Specification 2.0," http://www.omg.org

[CORBA-LDAP] Ryan, V., Lee, R. and S. Seligman, "Schema for Representing CORBA Object References in an LDAP Directory", RFC 2714, October 1999.

[Java] Ken Arnold and James Gosling, "The Java(tm) Programming Language," Second Edition, ISBN 0-201-31006-6.

[JNDI] Java Software, Sun Microsystems, Inc., "The Java(tm) Naming and Directory Interface (tm) Specification," February 1998. http://java.sun.com/products/jndi/

[LDAPv3] Wahl, M., Howes, T. and S. Kille, "Lightweight Directory Access Protocol (v3)", RFC 2251, December 1997.

[RMI] Java Software, Sun Microsystems, Inc., "Remote Method Invocation," November 1998. http://java.sun.com/products/jdk/1.2/docs/guide/rmi

[RMI-IIOP] IBM and Java Software, Sun Microsystems, Inc., "RMI over IIOP", June 1999.
http://java.sun.com/products/rmi-iiop/

[Serial] Java Software, Sun Microsystems, Inc., "Object Serialization Specification," November 1998.
http://java.sun.com/products/jdk/1.2/docs/guide/serialization

[v3Schema] Wahl, M., "A Summary of the X.500(96) User Schema for use with LDAPv3", RFC 2256, December 1997.

8. Authors' Addresses

Vincent Ryan
Sun Microsystems, Inc.
Mail Stop EDUB03
901 San Antonio Road
Palo Alto, CA 94303
USA

Phone: +353 1 819 9151
EMail: vincent.ryan@ireland.sun.com

Scott Seligman
Sun Microsystems, Inc.
Mail Stop UCUP02-209
901 San Antonio Road
Palo Alto, CA 94303
USA

Phone: +1 408 863 3222
EMail: scott.seligman@eng.sun.com

Rosanna Lee
Sun Microsystems, Inc.
Mail Stop UCUP02-206
901 San Antonio Road
Palo Alto, CA 94303
USA

Phone: +1 408 863 3221
EMail: rosanna.lee@eng.sun.com

Appendix - LDAP Schema

```
  -- Attribute types --

  ( 1.3.6.1.4.1.42.2.27.4.1.6
    NAME 'javaClassName'
    DESC 'Fully qualified name of distinguished Java class or interface'
    EQUALITY caseExactMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
    SINGLE-VALUE
  )

  ( 1.3.6.1.4.1.42.2.27.4.1.7
    NAME 'javaCodebase'
    DESC 'URL(s) specifying the location of class definition'
    EQUALITY caseExactIA5Match
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.26
  )

  ( 1.3.6.1.4.1.42.2.27.4.1.8
    NAME 'javaSerializedData'
    DESC 'Serialized form of a Java object'
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.40
    SINGLE-VALUE
  )

  ( 1.3.6.1.4.1.42.2.27.4.1.10
    NAME 'javaFactory'
    DESC 'Fully qualified Java class name of a JNDI object factory'
    EQUALITY caseExactMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
    SINGLE-VALUE
  )

  ( 1.3.6.1.4.1.42.2.27.4.1.11
    NAME 'javaReferenceAddress'
    DESC 'Addresses associated with a JNDI Reference'
    EQUALITY caseExactMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
  )

  ( 1.3.6.1.4.1.42.2.27.4.1.12
    NAME 'javaDoc'
    DESC 'The Java documentation for the class'
    EQUALITY caseExactIA5Match
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.26
  )
```

```
  ( 1.3.6.1.4.1.42.2.27.4.1.13
    NAME 'javaClassNames'
    DESC 'Fully qualified Java class or interface name'
    EQUALITY caseExactMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
  )

  -- from RFC-2256 --

  ( 2.5.4.13
    NAME 'description'
    EQUALITY caseIgnoreMatch
    SUBSTR caseIgnoreSubstringsMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.15{1024}
  )

  -- Object classes --

  ( 1.3.6.1.4.1.42.2.27.4.2.1
    NAME 'javaContainer'
    DESC 'Container for a Java object'
    SUP top
    STRUCTURAL
    MUST ( cn )
  )

  ( 1.3.6.1.4.1.42.2.27.4.2.4
    NAME 'javaObject'
    DESC 'Java object representation'
    SUP top
    ABSTRACT
    MUST ( javaClassName )
    MAY ( javaClassNames $ javaCodebase $ javaDoc $ description )
  )

  ( 1.3.6.1.4.1.42.2.27.4.2.5
    NAME 'javaSerializedObject'
    DESC 'Java serialized object'
    SUP javaObject
    AUXILIARY
    MUST ( javaSerializedData )
  )
```

```
  ( 1.3.6.1.4.1.42.2.27.4.2.7
    NAME 'javaNamingReference'
    DESC 'JNDI reference'
    SUP javaObject
    AUXILIARY
    MAY ( javaReferenceAddress $ javaFactory )
  )

  ( 1.3.6.1.4.1.42.2.27.4.2.8
    NAME 'javaMarshalledObject'
    DESC 'Java marshalled object'
    SUP javaObject
    AUXILIARY
    MUST ( javaSerializedData )
  )

  -- Matching rule from ISO X.520 --

  ( 2.5.13.5
    NAME 'caseExactMatch'
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
  )
```

Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.