

Jini™ Lookup Attribute Schema Specification

The Jini™ system is a Java™ platform-centric distributed system designed around the goals of simplicity, flexibility, and federation. The Jini Discovery protocol is used by entities that wish to start participating in a Jini tsystem. This document specifies the schema for attributes of registrants in the lookup service.



THE NETWORK IS THE COMPUTER™

901 San Antonio Road
Palo Alto, CA 94303 USA
415 960-1300
fax 415 969-9131

Revision 1.0
January 25, 1999

Copyright © 1999 Sun Microsystems, Inc.
901 San Antonio Road, Palo Alto, CA 94303 USA

All rights reserved. Copyright in this document is owned by Sun Microsystems, Inc.

Sun Microsystems, Inc. has patent and other intellectual property rights relating to implementations of the technology described in this Specification ("Sun IPR"). Your limited right to use this Specification does not grant you any right or license to Sun IPR. A limited license to Sun IPR is available from Sun under a separate Community Source License.

THIS SPECIFICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY YOU AS A RESULT OF USING THE SPECIFICATION.

THIS SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE SPECIFICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE SPECIFICATIONS AT ANY TIME, IN ITS SOLE DISCRETION. SUN IS UNDER NO OBLIGATION TO PRODUCE FURTHER VERSIONS OF THE SPECIFICATION OR ANY PRODUCT OR TECHNOLOGY BASED UPON THE SPECIFICATION. NOR IS SUN UNDER ANY OBLIGATION TO LICENSE THE SPECIFICATION OR ANY ASSOCIATED TECHNOLOGY, NOW OR IN THE FUTURE, FOR PRODUCTIVE OR OTHER USE.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-1(a).

TRADEMARKS

Sun, the Sun logo, Sun Microsystems, Jini, JavaSpaces, JavaSoft, JavaBeans, JDK, Java, HotJava, HotJava Views, Visual Java, Solaris, NEO, Joe, Netra, NFS, ONC, ONC+, OpenWindows, PC-NFS, EmbeddedJava, PersonalJava, SNM, SunNet Manager, Solaris sunburst design, Solstice, SunCore, SolarNet, SunWeb, Sun Workstation, The Network Is The Computer, ToolTalk, Ultra, Ultracomputing, Ultrasever, Where The Network Is Going, Sun WorkShop, XView, Java WorkShop, the Java Coffee Cup logo, and Visual Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Contents



1. Introduction	1
1.1 Overview	1
1.1.1 Terminology	2
1.2 Design Issues	2
1.3 Dependencies	3
1.4 Comments	4
2. Human Access To Attributes	5
2.1 Providing a Single View of an Attribute's Value	5
3. JavaBeans Components and Design Patterns	7
3.1 Allowing Display and Modification of Attributes	7
3.1.1 Using JavaBeans Components with Entry Classes	7
3.2 Associating JavaBeans Components with Entry Classes	9
3.3 Supporting Interfaces and Classes	11
4. Generic Attribute Classes	13
4.1 Indicating User Modifiability	13



4.2 Basic Service Information	14
4.3 More Specific Information	15
4.4 Naming a Service	16
4.5 Adding a Comment To a Service	17
4.6 Physical Location	17
4.7 Status Information	20
4.8 Serialized Forms	20

1.1 Overview

The Jini™ Lookup service provides facilities for services to advertise their availability, and for would-be clients to obtain references to those services based on the attributes they provide. The mechanism it provides for registering and querying based on attributes is centered around the Java™ platform type system, and is based on the notion of an “entry”.

An entry is a class that contains a number of public fields of object type. Services provide concrete values for each of these fields; each value acts as an attribute. Entries thus provide aggregation of attributes into sets; a service may provide several entries when registering itself in the lookup service, which means that attributes on each service are provided in a set of sets.

The purpose of this document is to provide a framework in which services and their would-be clients can interoperate. This framework takes two parts.

- ◆ We describe a set of common predefined entries that span much of the basic functionality that is needed both by services registering themselves and entities that are searching for services.
- ◆ Since we cannot anticipate all of the future needs of clients of the lookup service, we provide a set of guidelines and design patterns for extending, using and imitating this set in ways that are consistent and predictable. We also construct some examples that illustrate the use of these patterns.

1.1.1 Terminology

Throughout this document, we will use the following terms in consistent ways:

- ◆ service - a service that has registered, or will register, itself with the lookup service
- ◆ client - an entity that performs queries on the lookup service, in order to find particular services

1.2 Design Issues

Several factors influence and constrain the design of the lookup service schema.

Matching Cannot Always Be Automated

No matter how much information it has at its disposal, a client of the lookup service will not always be able to find a single unique match without assistance, when it performs a lookup. In many instances, we expect that more than one service will match a particular query. Accordingly, both the lookup service and the attribute schema are geared towards reducing the number of matches that are returned on a given lookup to a minimum, and not necessarily to just one.

Attributes Are Mostly Static

We have designed the schema for the lookup service with the assumption that most attributes will not need to be changed frequently. For example, we do not expect attributes to change more often than once every minute or so. This decision is based on our expectation that clients that need to make a choice of service based on more frequently-updated attributes will be able to talk to whatever small set of services the lookup service returns for a query, and on our belief that the benefit of updating attributes frequently at the lookup service is outweighed by the cost in network traffic and processing.

Humans Need to Understand Most Attributes

A corollary of the idea that matching cannot always be automated is that humans - whether they be users or administrators of services - must be able to understand and interpret attributes. This has several implications:

- ◆ We must provide a mechanism to deal with localization of attributes
- ◆ Multiple-valued attributes must provide a way for humans to see only one value (see Chapter 2)

We will cover human accessibility of attributes in a later chapter.

Attributes Can Be Changed by Services or Humans, But Not Both

For any given attribute class, we expect that attributes within that class will all be set or modified either by the service, or via human intervention, but not both. What do we mean by this? A service is unlikely to be able to determine that it has been moved from one room to another, for example, so we would not expect the fields of a “location” attribute class to be changed by the service itself. Similarly, we do not expect that a human operator will need to change the name of the vendor of a particular service.

This idea has implications for our approach to ensuring that the values of attributes are valid.

Attributes Must Interoperate with JavaBeans™ Components

The JavaBeans™ specification provides a number of facilities relating to the localized display, and modification, of properties, and has been widely adopted. It is to our advantage to provide a familiar set of mechanisms for manipulating attributes in these ways.

1.3 Dependencies

This document relies on the following other specifications:

- ◆ Jini™ Entry Specification
- ◆ Jini™ Entry Utilities Specification
- ◆ JavaBeans™ Specification

1.4 *Comments*

Please direct comments to `jini-comments@java.sun.com`.

2.1 Providing a Single View of an Attribute's Value

Consider the following entry class:

```
public class Foo implements net.jini.core.entry.Entry {
    public Bar baz;
}

public class Bar {
    int quux;
    boolean zot;
}
```

A visual search tool is going to have a difficult time rendering the value of an instance of class `Bar` in a manner that is comprehensible to humans. Accordingly, to avoid such situations, entry class implementors should use the following guidelines when designing a class that is to act as a value for an attribute:

- ◆ Provide a property editor class of the appropriate type, as described in section 9.2 of the JavaBeans Specification.
- ◆ Extend the `java.awt.Component` class; this allows a value to be represented by a JavaBeans component or some other “active” object.

- ◆ Either provide a non-default implementation of the `java.lang.Object.toString` method, or inherit directly or indirectly from a class that does so (since the default implementation of `Object.toString` is not useful).

One of the above guidelines should be followed for all attribute value classes. Authors of entry classes should assume that any attribute value that does not satisfy one of these guidelines will be ignored by some or all user interfaces.

3.1 Allowing Display and Modification of Attributes

We use JavaBeans components to provide a layer of abstraction on top of the individual classes that implement the `net.jini.core.entry.Entry` interface, and this provides us with several benefits:

- ◆ This approach uses an existing standard, and thus reduces the amount of unfamiliar material for programmers
- ◆ JavaBeans components provide mechanisms for localized display of attribute values and descriptions
- ◆ Modification of attributes is also handled, via property editors

3.1.1 Using JavaBeans Components with Entry Classes

Many, if not most, entry classes should have a bean class associated with them. Our use of JavaBeans components provides a familiar mechanism for authors of browse/search tools to represent information about a service's attributes, such as its icons and appropriately localized descriptions of the meanings and values of its attributes. JavaBeans components also play a role in permitting administrators of a service to modify some of its attributes, as they can manipulate the values of its attributes using standard JavaBeans component mechanisms.

For example, obtaining a `java.beans.BeanDescriptor` for a JavaBeans component that is linked to a “location” entry object for a particular service allows a programmer to obtain an icon that gives a visual indication of what that entry class is for, along with a short textual description of the class and the values of the individual attributes in the location object. It also permits an administrative tool to view and change certain fields in the location, such as the floor number.

3.2 *Associating JavaBeans Components with Entry Classes*

The pattern for establishing a link between an entry object and an instance of its JavaBeans component is simple enough, as this example illustrates:

```
package org.example.foo;

import java.io.Serializable;
import net.jini.lookup.entry.EntryBean;
import net.jini.entry.AbstractEntry;

public class Size {
    public int value;
}

public class Cavenewt extends AbstractEntry {
    public Cavenewt() {
    }
    public Cavenewt(Size anvilSize) {
        this.anvilSize = anvilSize;
    }
    public Size anvilSize;
}

public class CavenewtBean implements EntryBean, Serializable {
    protected Cavenewt assoc;
    public CavenewtBean() {
        super();
        assoc = new Cavenewt();
    }
    public void setAnvilSize(Size x) {
        assoc.anvilSize = x;
    }
    public Size getAnvilSize() {
        return assoc.anvilSize;
    }
    public void makeLink(Entry obj) {
        assoc = (Cavenewt) obj;
    }
    public Entry followLink() {
        return assoc;
    }
}
```

From the above, the pattern should be relatively clear:

- ◆ The name of a JavaBeans component is derived by taking the fully-qualified entry class name and appending the string `Bean`; for example, the name of the JavaBeans component associated with the entry class `foo.bar.Baz` is `foo.bar.BazBean`. This implies that an entry class and its associated JavaBeans component must reside in the same package.
- ◆ The class has both a public no-arg constructor and a public constructor that takes each public object field of the class and its superclasses as parameter. The former constructs an empty instance of the class, and the latter initializes each field of the new instance to the given parameter.
- ◆ The class implements the `net.jini.core.entry.Entry` interface, preferably by extending the `net.jini.entry.AbstractEntry` class, and the JavaBeans component implements the `net.jini.lookup.entry.EntryBean` interface.
- ◆ There is a one-to-one link between a JavaBeans component and a particular entry object. The `makeLink` method establishes this link, and will throw an exception if the association is with an entry class of the wrong type. The `followLink` method returns the entry object associated with a particular JavaBeans component.
- ◆ The no-arg public constructor for a JavaBeans component creates and makes a link to an empty entry object.
- ◆ For each public object field `foo` in an entry class, there exist both a `setFoo` and a `getFoo` method in the associated JavaBeans component. The `setFoo` method takes a single argument of the same type as the `foo` field in the associated entry, and sets the value of that field to its argument. The `getFoo` method returns the value of that field.

3.3 *Supporting Interfaces and Classes*

The following classes and interfaces provide facilities for handling entry classes and their associated JavaBeans components.

```
package net.jini.lookup.entry;

public class EntryBeans {
    public static EntryBean createBean(Entry e)
        throws ClassNotFoundException, java.io.IOException;

    public static Class getBeanClass(Class c)
        throws ClassNotFoundException;
}

public interface EntryBean {
    void makeLink(Entry e);
    Entry followLink();
}
```

The `EntryBeans` class cannot be instantiated. Its sole method, `createBean`, creates and initializes a new JavaBeans component, and links it to the entry object passed in as its parameter. If a problem occurs in instantiating the JavaBeans component, this method throws either a `java.io.IOException` or a `java.lang.ClassNotFoundException`.

The `createBean` method uses the same mechanism for instantiating a JavaBeans component as the `java.beans.Beans.instantiate` method. It will initially try to instantiate the JavaBeans component using the same class loader as the entry it is passed, and if that fails, it will fall back to using the default class loader.

The `getBeanClass` method returns the class of the JavaBeans component associated with the given attribute class. If the class passed in does not implement the `net.jini.core.entry.Entry` interface, a `java.lang.IllegalArgumentException` is thrown. If no class can be found for the given attribute class, a `java.lang.ClassNotFoundException` is thrown.

The `EntryBean` interface must be implemented by all JavaBeans components that are intended to be linked to entry objects. The `makeLink` method establishes a link between a JavaBeans component object and an entry object,

and the `followLink` method returns the entry object linked to by a particular JavaBeans component. Note that objects that implement the `EntryBean` interface should not be assumed to perform any internal synchronization in their implementations of the `makeLink` or `followLink` methods, or in the `setFoo` or `getFoo` patterns.

This chapter describes some attribute classes that are generic to many or all services, and the JavaBeans components that are associated with each. Unless otherwise stated, all classes defined here live in the `net.jini.lookup.entry` package. The definitions assume the following classes to have been imported:

- ◆ `java.io.Serializable`
- ◆ `net.jini.entry.AbstractEntry`

4.1 *Indicating User Modifiability*

In order to indicate that certain entry classes should only be modified by the service that registered itself with instances of these entry classes, we annotate them with the `ServiceControlled` interface.

```
public interface ServiceControlled {  
}
```

Authors of administrative tools that modify fields of attribute objects at the lookup service should not permit users to either modify any fields or add any new instances of objects that implement this interface.

4.2 Basic Service Information

The `ServiceInfo` attribute class provides some basic information about a service.

```
public class ServiceInfo implements ServiceControlled
    extends AbstractEntry {
    public ServiceInfo();
    public ServiceInfo(String name, String manufacturer,
        String vendor, String version,
        String model, String serialNumber);

    public String name;
    public String manufacturer;
    public String vendor;
    public String version;
    public String model;
    public String serialNumber;
}

public class ServiceInfoBean implements EntryBean, Serializable {
    public String getName();
    public void setName(String s);
    public String getManufacturer();
    public void setManufacturer(String s);
    public String getVendor();
    public void setVendor(String s);
    public String getVersion();
    public void setVersion(String s);
    public String getModel();
    public void setModel(String s);
    public String getSerialNumber();
    public void setSerialNumber(String s);
}
```

Each service should only register itself with one instance of this class.

The fields of the `ServiceInfo` class have the following meanings:

- ◆ The `name` field contains a specific product name, such as “Ultra 30” (for a particular Sun Microsystems™ workstation) or “JavaSafe™” (for a specific configuration management service). This string should not include the name of the manufacturer or vendor.

- ◆ The `manufacturer` field provides the name of the company that “built” this service. This might be a hardware manufacturer or a software authoring company.
- ◆ The `vendor` field contains the name of the company that sells the software or hardware that provides this service. This may be the same name as is in the `manufacturer` field, or it could be the name of a VAR. This field exists so that in cases where VARs rebadge products built by other companies, users will be able to search based on either name.
- ◆ The `version` field provides information about the version of this service. It is a free-form field, though we expect that service implementors will follow normal version-naming conventions in using it.
- ◆ The `model` field contains the specific model name or number of the product, if any.
- ◆ The `serialNumber` field provides the serial number of this instance of the service, if any.

4.3 *More Specific Information*

The `ServiceType` class allows an author of a service to deliver information that is specific to a particular instance of a service, rather than to services in general.

```
public class ServiceType implements ServiceControlled
    extends AbstractEntry {
    public ServiceType();

    public java.awt.Image getIcon(int iconKind);
    public String getDisplayName();
    public String getShortDescription();
}
```

Each service may register itself with multiple instances of this class, usually with one instance for each type of service interface it implements.

This class has no public fields and, as a result, has no associated JavaBeans component.

The `getIcon` method returns an icon of the appropriate kind for the service; it works in the same way as the `getIcon` method in the `java.beans.BeanInfo` interface, with the value of `iconKind` being taken from the possibilities defined in that interface. The `getDisplayName` and `getShortDescription` methods return a localized human-readable name and description for the service, in the same manner as their counterparts in the `java.beans.FeatureDescriptor` class. Each of these methods returns null if no information of the appropriate kind is defined.

In case the distinction between the information this class provides and that provided by a JavaBeans component's metainformation is unclear, the `ServiceType` class is meant to be used in the lookup service as one of the entry classes with which a service registers itself, and so it can be customized on a per-service basis. By contrast, the `FeatureDescriptor` and `BeanInfo` objects for all `EntryBean` classes provide only generic information about those classes, and none about specific instances of those classes.

4.4 Naming a Service

People like to associate names with particular services, and may do so using the `Name` class.

```
public class Name extends AbstractEntry {
    public Name();
    public Name(String name);

    public String name;
}

public class NameBean implements EntryBean, Serializable {
    public String getName();
    public void setName(String s);
}
```

Services may register themselves with multiple instances of this class, and either services or administrators may add, modify or remove instances of this class from the attribute set under which a service is registered.

The `name` field provides a short name for a particular instance of a service (for example, "Bob's toaster").

4.5 *Adding a Comment To a Service*

In cases where some kind of comment is appropriate for a service (for example, “this toaster tends to burn bagels”), the `Comment` class provides an appropriate facility.

```
public class Comment extends AbstractEntry {
    public Comment();
    public Comment(String comment);

    public String comment;
}

public class CommentBean implements EntryBean, Serializable {
    public String getComment();
    public void setComment(String s);
}
```

A service may have more than one comment associated with it, and comments may be added, removed or edited by either a service itself, administrators, or users.

4.6 *Physical Location*

The `Location` and `Address` classes provide information about the physical location of a particular service.

Since many services have no physical location, some have one, and a few may have more than one, it may make sense for a service to register itself with zero or more instances of either of these classes, depending on its nature.

The `Location` class is intended to provide information about the physical location of a service in a single building or on a small, unified campus. The `Address` class provides more information, and may be appropriate for use with the `Location` class in a larger, more geographically distributed, organization.

```
public class Location extends AbstractEntry {
    public Location();
    public Location(String floor, String room, String building);

    public String floor;
    public String room;
    public String building;
}

public class LocationBean implements EntryBean, Serializable {
    public String getFloor();
    public void setFloor(String s);
    public String getRoom();
    public void setRoom(String s);
    public String getBuilding();
    public void setBuilding(String s);
}
```

```
public class Address extends AbstractEntry {
    public Address();
    public Address(String street, String organization,
        String organizationalUnit, String locality,
        String stateOrProvince, String postalCode,
        String country);

    public String street;
    public String organization;
    public String organizationalUnit;
    public String locality;
    public String stateOrProvince;
    public String postalCode;
    public String country;
}

public class AddressBean implements EntryBean, Serializable {
    public String getStreet();
    public void setStreet(String s);
    public String getOrganization();
    public void setOrganization(String s);
    public String getOrganizationalUnit();
    public void setOrganizationalUnit(String s);
    public String getLocality();
    public void setLocality(String s);
    public String getStateOrProvince();
    public void setStateOrProvince(String s);
    public String getPostalCode();
    public void setPostalCode(String s);
    public String getCountry();
    public void setCountry(String s);
}
```

We believe the fields of these classes to be self-explanatory, with the possible exception of the `locality` field of the `Address` class, which would typically hold the name of a city.

4.7 Status Information

Some attributes of a service may constitute long-lived status, such as an indication that a printer is out of paper. We provide a class, `Status`, that implementors can use as a base for providing status-related entry classes.

```
public abstract class Status extends AbstractEntry {
    protected Status();
    protected Status(StatusType severity);

    public StatusType severity;
}

public class StatusType implements Serializable {
    private final int type;
    private StatusType(int t) { type = t; }
    public static final StatusType ERROR = new StatusType(1);
    public static final StatusType WARNING = new StatusType(2);
    public static final StatusType NOTICE = new StatusType(3);
    public static final StatusType NORMAL = new StatusType(4);
}

public abstract class StatusBean implements EntryBean,
    Serializable {
    public StatusType getSeverity();
    public void setSeverity(StatusType i);
}
```

We define a separate `StatusType` class in order to make it possible to write a property editor that will work with the `StatusBean` class (we do not currently provide a property editor implementation).

4.8 Serialized Forms

For each attribute class (`Address`, `Comment`, `Location`, `Name`, `ServiceInfo`, `ServiceType`, `Status`), the serializable fields are the declared public fields. Each corresponding JavaBeans component class has a single serializable field named `assoc`, with the attribute class as the declared type. For example, `AddressBean` has a serializable field named `assoc` of type `Address`, `CommentBean` has a serializable field named `assoc` of type `Comment`, and so on.

StatusType has a single serializable field:

◆ int type

The serialVersionUID for each class is as follows:

- ◆ Address: 2896136903322046578
- ◆ AddressBean: 4491500432084550577
- ◆ Comment: 7138608904371928208
- ◆ CommentBean: 5272583409036504625
- ◆ Location: -3275276677967431315
- ◆ LocationBean: -4182591284470292829
- ◆ Name: 2743215148071307201
- ◆ NameBean: -6026791845102735793
- ◆ ServiceInfo: -1116664185758541509
- ◆ ServiceInfoBean: 8352546663361067804
- ◆ ServiceType: -6443809721367395836
- ◆ Status: -5193075846115040838
- ◆ StatusBean: -1975539395914887503
- ◆ StatusType: -8268735508512712203

