



# A Collection of Jini™ Technology Helper Utilities and Services Specifications

This Collection of Jini™ Technology Helper Utilities and Services Specifications defines a set of standard helper utilities and services which extend the Jini Technology Core Platform. These helper utilities and services encapsulate desirable behaviors in the form of a set of reusable components that can be used to help simplify the process of developing Jini technology-enabled clients and services (*Jini clients and services*) for the Jini technology application environment. Employing these utilities and services to build such desirable behavior into a Jini client or service can help to avoid poor design and implementation decisions, greatly simplifying the development process.



Version 1.1Beta  
May 2000

Copyright © 2000 Sun Microsystems, Inc.  
901 San Antonio Road, Palo Alto, CA 94303 USA.  
All rights reserved.

Sun Microsystems, Inc. has intellectual property rights (“Sun IPR”) relating to implementations of the technology described in this publication (“the Technology”). In particular, and without limitation, Sun IPR may include one or more patents or patent applications in the U.S. or other countries. Your limited right to use this publication does not grant you any right or license to Sun IPR nor any right or license to implement the Technology. Sun may, in its sole discretion, make available a limited license to Sun IPR and/or to the Technology under a separate license agreement. Please visit <http://www.sun.com/software/communitysource/>.

Sun, the Sun logo, Sun Microsystems, Jini, the Jini logo, JavaSpaces, Java, JavaBeans, Solaris, NFS, PC-NFS, EmbeddedJava, PersonalJava, and Solstice are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

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.

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 ANY TECHNOLOGY, PRODUCT, OR PROGRAM DESCRIBED IN THIS SPECIFICATION AT ANY TIME.

---

# Contents

<b>Introduction to Helper Utilities and Services</b>	<b>1</b>
<b>US.1 Summary</b>	<b>1</b>
<b>US.2 Terminology</b>	<b>3</b>
US.2.1 Terms Related to Discovery and Join	3
US.2.2 Jini Clients and Services	4
US.2.3 Helper Service	4
US.2.4 Helper Utility	5
US.2.5 Managed Sets	5
US.2.6 What Exceptions Imply About Future Behavior	5
US.2.7 Unavailable Lookup Services	7
US.2.8 Discarding a Lookup Service	8
US.2.9 Activation	11
<b>US.3 Helper Utility Specifications</b>	<b>13</b>
US.3.1 The Jini Discovery Utilities Specification	13
US.3.2 The Jini Lease Utilities Specification	17
US.3.3 The Jini Join Utilities Specification	17
US.3.4 The Jini Service Discovery Utilities Specification	18
<b>US.4 Helper Service Specifications</b>	<b>21</b>
US.4.1 The Jini Lookup Discovery Service Specification	21
US.4.2 The Jini Lease Renewal Service Specification	21
US.4.3 The Jini Event Mailbox Service Specification	22
<b>US.5 Dependencies</b>	<b>23</b>
 <b>Jini™ Discovery Utilities Specification</b>	 <b>25</b>
<b>DU.0 Introduction</b>	<b>25</b>
DU.0.1 Dependencies	25
DU.0.2 Comments	26
<b>DU.1 The Discovery Management Interfaces</b>	<b>27</b>
DU.1.1 Overview	27

DU.1.2	Other Types	28
DU.1.3	The DiscoveryManagement Interface	29
DU.1.4	The DiscoveryGroupManagement Interface	32
DU.1.5	The DiscoveryLocatorManagement Interface	35
DU.1.6	Supporting Interfaces and Classes	37
DU.1.7	Serialized Forms	41
<b>DU.2</b>	<b>LookupDiscovery Utility</b>	<b>43</b>
DU.2.1	Other Types	43
DU.2.2	The Interface	44
DU.2.3	The Semantics	44
DU.2.4	Supporting Interfaces and Classes	45
DU.2.5	Serialized Forms	48
<b>DU.3</b>	<b>The LookupLocatorDiscovery Utility</b>	<b>49</b>
DU.3.1	Overview	49
DU.3.2	Other Types	50
DU.3.3	The Interface	50
DU.3.4	The Semantics	51
DU.3.5	Supporting Interfaces	52
<b>DU.4</b>	<b>The LookupDiscoveryManager Utility</b>	<b>53</b>
DU.4.1	Overview	53
DU.4.2	Other Types	53
DU.4.3	The Interface	54
DU.4.4	The Semantics	54
DU.4.5	Supporting Interfaces and Classes	57
<b>DU.5</b>	<b>Low-Level Discovery Protocol Utilities</b>	<b>59</b>
DU.5.1	The Constants Class	59
DU.5.2	The OutgoingMulticastRequest Utility	61
DU.5.3	The IncomingMulticastRequest Utility	62
DU.5.4	The OutgoingMulticastAnnouncement Utility	64
DU.5.5	The IncomingMulticastAnnouncement Utility	66
DU.5.6	The OutgoingUnicastRequest Utility	68
DU.5.7	The IncomingUnicastRequest Utility	69
DU.5.8	The OutgoingUnicastResponse Utility	71
DU.5.9	The IncomingUnicastResponse Utility	72
	<b>Jini™ Entry Utilities Specification</b>	<b>75</b>
<b>EU.1</b>	<b>Entry Utilities</b>	<b>75</b>
EU.1.1	AbstractEntry	75
EU.1.2	Serialized Form	76

EU.1.3	Comments	76
<b>Jini™ Lease Utilities Specification</b>		<b>77</b>
<b>LM.1</b>	<b>Introduction</b>	<b>77</b>
<b>LM.2</b>	<b>The LeaseRenewalManager</b>	<b>79</b>
LM.2.1	Other Types	80
LM.2.2	The Interface	80
LM.2.3	The Semantics	81
LM.2.4	Supporting Interfaces and Classes	87
<b>Jini™ Join Utilities Specification</b>		<b>91</b>
<b>JU.1</b>	<b>Introduction</b>	<b>91</b>
<b>JU.2</b>	<b>The JoinManager</b>	<b>93</b>
JU.2.1	Other Types	94
<b>JU.3</b>	<b>The Interface</b>	<b>95</b>
<b>JU.4</b>	<b>The Semantics</b>	<b>97</b>
<b>JU.5</b>	<b>Supporting Interfaces and Classes</b>	<b>103</b>
JU.5.1	The DiscoveryManagement Interface	103
JU.5.2	The ServiceIDListener Interface	104
<b>Jini™ Service Discovery Utilities Specification</b>		<b>105</b>
<b>SD.1</b>	<b>Introduction</b>	<b>105</b>
<b>SD.2</b>	<b>The ServiceDiscoveryManager</b>	<b>107</b>
SD.2.1	The Object Types	109
<b>SD.3</b>	<b>The Interface</b>	<b>111</b>
<b>SD.4</b>	<b>The Semantics</b>	<b>113</b>
SD.4.1	Exporting RemoteEventListener Objects	113
<b>SD.5</b>	<b>Supporting Interfaces and Classes</b>	<b>125</b>
SD.5.1	The DiscoveryManagement Interface	125
SD.5.2	The ServiceItemFilter Interface	126
SD.5.3	The ServiceDiscoveryEvent Class	127
SD.5.4	The ServiceDiscoveryListener Interface	129
SD.5.5	The LookupCache Interface	131
<b>Jini™ Lookup Attribute Schema Specification</b>		<b>135</b>
<b>LS.1</b>	<b>Introduction</b>	<b>135</b>
LS.1.1	Terminology	136
LS.1.2	Design Issues	136

LS.1.3	Dependencies .....	137
LS.1.4	Comments .....	137
<b>LS.2</b>	<b>Human Access to Attributes .....</b>	<b>139</b>
LS.2.1	Providing a Single View of an Attribute's Value .....	139
<b>LS.3</b>	<b>JavaBeans Components and Design Patterns .....</b>	<b>141</b>
LS.3.1	Allowing Display and Modification of Attributes .....	141
LS.3.2	Associating JavaBeans Components with Entry Classes ....	142
LS.3.3	Supporting Interfaces and Classes .....	144
<b>LS.4</b>	<b>Generic Attribute Classes .....</b>	<b>145</b>
LS.4.1	Indicating User Modifiability .....	145
LS.4.2	Basic Service Information .....	145
LS.4.3	More Specific Information .....	147
LS.4.4	Naming a Service .....	148
LS.4.5	Adding a Comment to a Service .....	148
LS.4.6	Physical Location .....	149
LS.4.7	Status Information .....	150
LS.4.8	Serialized Forms .....	151
<b>Jini™</b>	<b>Lookup Discovery Service .....</b>	<b>153</b>
<b>LD.1</b>	<b>Introduction .....</b>	<b>153</b>
LD.1.1	Other Types .....	156
<b>LD.2</b>	<b>The Interface .....</b>	<b>159</b>
<b>LD.3</b>	<b>The Semantics .....</b>	<b>161</b>
LD.3.1	Registration Semantics .....	161
LD.3.2	Event Semantics .....	164
LD.3.3	Leasing Semantics .....	166
<b>LD.4</b>	<b>Supporting Interfaces and Classes .....</b>	<b>167</b>
LD.4.1	The LookupDiscoveryRegistration Interface .....	167
LD.4.2	The RemoteDiscoveryEvent Class .....	176
LD.4.3	The LookupUnmarshalException Class .....	180
<b>Jini™</b>	<b>Lease Renewal Service Specification .....</b>	<b>185</b>
<b>LR.1</b>	<b>Introduction .....</b>	<b>185</b>
LR.1.1	Other Types .....	187
<b>LS.2</b>	<b>The Interface .....</b>	<b>189</b>
<b>LS.3</b>	<b>The Semantics .....</b>	<b>201</b>
<b>LS.4</b>	<b>Supporting Interfaces and Clients .....</b>	<b>203</b>
<b>Jini™</b>	<b>Event Mailbox Service Specification .....</b>	<b>205</b>

<b>EM.1 Introduction</b> .....	<b>205</b>
EM.1.1 Other Types .....	206
<b>EM.2 The Interface</b> .....	<b>207</b>
<b>EM.3 The Semantics</b> .....	<b>209</b>
<b>EM.4 Supporting Interfaces and Classes</b> .....	<b>211</b>





## Introduction to Helper Utilities and Services

### US.1 Summary

**W**HEN developing clients and services that will participate in the application environment for Jini™ technology, there are a number of behaviors that the developer may find desirable to incorporate in the client or service. Some of these behaviors may satisfy requirements described in the specifications of various Jini technology components; some behaviors may simply represent design practices that are desirable and should be encouraged. Examples of the sort of behavior that is required or desirable include the following:

- ◆ It is a requirement of the Jini discovery protocols that a service must continue to listen for and act on announcements from lookup services in which the service has registered interest.
- ◆ It is a requirement of the Jini discovery protocols that, until successful, a service must continue to attempt to join the specific lookup services with which it has been configured to join.
- ◆ Under many conditions, a Jini technology-enabled client (*Jini client*) or service will wish to regularly renew leases that it holds. For example, when a Jini technology-enabled service (*Jini service*) registers with a Jini lookup service, the service is requesting residency in the lookup service. Residency in a lookup service is a leased resource. Thus, when the requested residency is granted, the lookup service also imposes a lease on that residency. Typically, such a registered service will wish to extend the lease on its residency

beyond the original expiration time; resulting in a need to renew the lease on a regular basis.

- ◆ Many Jini services will need to maintain a dormant (inactive) state, becoming active only when needed.
- ◆ Many Jini clients and services will need to have a mechanism for finding and managing Jini services.
- ◆ Many Jini clients and services will find it desirable to employ a separate service that will handle events, in some useful way, on behalf of the participant.

In order to help simplify the process of developing clients and services for the application environment for Jini technology (*Jini application environment*), a set of reusable components that encapsulate behaviors such as those outlined above, are specified in *A Collection of Jini Technology Helper Utilities and Services Specifications* (the *collection* or *helper utilities and services specifications*). Employing these utilities and services to build such desirable behavior into a Jini client or service can help to avoid poor design and implementation decisions, greatly simplifying the development process.

What is presented first is terminology that may be helpful when analyzing these specifications. Following the section on terminology, brief summaries of the content of each of the current helper utilities and services specifications are provided. Finally, the other specifications on which these specifications depend are listed for reference.

---

## US.2 Terminology

**T**HIS section defines terms and discusses concepts that may be referenced throughout the helper utilities and services specifications. While the terms and concepts appearing in this section are general in nature and may apply to multiple components specified in this collection, each specification may define additional terms and concepts to further facilitate the understanding of a particular component. Each specification may also present supplemental information about some of the terms defined in this section and their relationship with the component being specified.

Because this document makes use of a number of terms defined in the *Jini Technology Glossary*, reviewing the glossary is recommended. A number of the terms defined in the glossary are also defined in this section to provide easy reference, and because those terms are used extensively in the helper utilities and services specifications. Additionally, this section augments the definitions of some of the terms from the glossary with details relevant to those specifications.

In addition to the glossary, the *Jini Technology Core Platform Specification* (referred to as the *JCP Specification* or simply, the *core specification*) present detailed definitions of a number of terms and concepts appearing both in this section and throughout the helper utilities and services specifications. When appropriate, the relevant Jini specification will be referenced.

### US.2.1 Terms Related to Discovery and Join

The *Jini™ Discovery and Join Specification* defines a *discovering entity* as one or more cooperating software objects written in the Java™ programming language (*Java software objects*), executing on the same host, that are in the process of obtaining references to Jini lookup services. That specification also defines a *joining entity* as one or more cooperating Java software objects, on the same host, that have received a reference to a lookup service and are in the process of obtaining services from, and possibly exporting services to, a federation of lookup services referred to as a *djinn*. The lookup services comprising a djinn may be organized into one or more sets known as *groups*. Multiple groups may or may not be dis-

joint. Each group of lookup services is identified by a logical name represented by a `String` object.

The *Jini™ Discovery and Join Specification* defines two protocols used in the discovery process: the *multicast discovery protocol* and the *unicast discovery protocol*.

When a discovering entity employs the multicast discovery protocol to discover lookup services that are members of one or more groups belonging to a set of groups, that discovery process is referred to as *group discovery*.

The utility class `net.jini.core.discovery.LookupLocator` is defined in the *Jini™ Discovery and Join Specification*. Any instance of that class is referred to as a *locator*. When a discovering entity employs the unicast discovery protocol to discover specific lookup services, each corresponding to an element in a set of locators, that discovery process is referred to as *locator discovery*.

## US.2.2 Jini Clients and Services

For the purposes of this document, a Jini client is defined as a discovering entity that can retrieve a service (or a remote reference to a service) registered with a discovered lookup service, and invoke the methods of the service to meet the entity's requirements. An entity that acts only as a client never registers with (requests residency in) a lookup service.

A Jini service is defined as both a discovering and a joining entity containing methods that may be of use to some other Jini client or service, and that registers with discovered lookup services to provide access to those methods. Note that a Jini service can also act as a Jini client.

The term *client-like entity* may be used, in general, when referring to Jini clients and Jini services that act as clients.

Note that when the term *entity* is used, that term may be referring to a discovering entity, a joining entity, a client-like entity, a service, or some combination of these types of entities. Whenever that general term is used, it should be clear from the context what type of entity is being discussed.

## US.2.3 Helper Service

A Jini technology-enabled *helper service* is defined in this document as an interface or set of interfaces, with an associated implementation, that encapsulates behavior that is either required or highly desirable in service entities that adhere to the Jini technology programming model (or simply, the *Jini programming model*).

A helper service is a Jini service that can be registered with any number of lookup services, and whose methods can execute on remote hosts.

In general, a helper service should be of use to more than one type of entity participating in the Jini application environment, and should provide a significant reduction in development complexity for developers of such entities.

## US.2.4 Helper Utility

This document distinguishes between a helper *utility* and a helper *service*. Helper utilities are programming components that can be used during the construction of Jini services and/or clients. Helper utilities are *not* remote and do not register with a lookup service. Helper utilities are instantiated locally by entities wishing to employ them.

## US.2.5 Managed Sets

When performing discovery duties, entities will often maintain references to discovered lookup services in a set referred to as the *managed set of lookup services*. The entity may also maintain two other notable sets: the *managed set of groups* and the *managed set of locators*.

Each element of the managed set of groups is a name of a group whose members are lookup services that the entity wishes to be discovered via group discovery. The managed set of groups is typically represented as a String array, or a Collection of String elements.

Each element of the managed set of locators corresponds to a specific lookup service that the entity wishes to be discovered via locator discovery. Typically, this set is represented as an array of `net.jini.core.discovery.LookupLocator` objects or some other Collection type whose elements are `LookupLocator` objects.

Note that when the general term *managed set* is used, it should be clear from the context whether groups, locators, or lookup services are being discussed.

## US.2.6 What Exceptions Imply About Future Behavior

When interacting with a remote object, an entity may call methods that result in exceptions. The specification of those methods should define what each possible exception implies (if anything) about the current state of the object. One important aspect of an object's state is whether or not further interactions with the object are

likely to be fruitful. Throughout the helper utilities and services specifications, the following general terms may be used to classify what a given exception implies about the probability of success of future operations on the object that threw the exception:

- ◆ Bad object exception: If a method invocation on an object throws a *bad object exception*, it can be assumed that any further operations on that object will also fail.
- ◆ Bad invocation exception: If a method invocation on an object throws a *bad invocation exception*, it can be assumed that any retries of the *same* method with the *same* arguments that are expected to return the *same* value will also fail. No new assertions can be made about the probability of success of any future invocation of that method with different arguments or if a different return value is expected, nor can any new assertions be made about the probability of success of invocations of the object's other methods.
- ◆ Indefinite exception: If a method invocation on an object throws an *indefinite exception*, no new assertions can be made about the probability of success of any future invocation of that method, regardless of the arguments used or return value expected, nor can any new assertions be made about the probability of success of any *other* operation on the same object.

Unless otherwise noted, the throwing of a bad object, bad invocation, or indefinite exception by one object does not imply anything about the state of another object, even if both objects are associated with the same remote entity.

These terms can be used both in the specification of a method to describe the meaning of exceptions that might be thrown, as well as in the specification of what a given utility or service will, may, or should do when it receives an exception in the course of interacting with a given object.

In this document, if a given specification does not specify otherwise, the following classification is used to categorize `RuntimeExceptions`, `Errors`, and `java.rmi.RemoteExceptions` as bad object, bad invocation, or indefinite exceptions:

- ◆ Bad object exceptions:
  - Any instance of a `java.rmi.NoSuchObjectException`
  - Any instance of a `java.rmi.ServerError` in which the value of the error's `detail` field is a bad object exception
  - Any instance of a `java.lang.RuntimeException`

- Any instance of a `java.rmi.ServerException` in which the value of the exception's `detail` field is a bad object exception
  - Any instance of a `java.lang.Error` *unless* it is an instance of `java.lang.LinkageError` or `java.lang.OutOfMemoryError`
- ◆ Bad invocation exceptions:
- Any instance of a `java.rmi.MarshalException` in which the value of the exception's `detail` field is an instance of `java.io.ObjectStreamException`
  - Any instance of a `java.rmi.UnmarshalException` in which the value of the exception's `detail` field is an instance of `java.io.ObjectStreamException`
  - Any instance of a `java.rmi.ServerException` in which the value of the exception's `detail` field is a bad invocation exception
- ◆ Indefinite exceptions
- Any instance of a `java.rmi.RemoteException` *except* those that can be classified as either a bad invocation or bad object exception
  - Any instance of a `java.lang.OutOfMemoryError`
  - Any instance of a `java.lang.LinkageError`

## US.2.7 Unavailable Lookup Services

While interacting (or attempting to interact) with a lookup service, an entity may encounter one of the exception types described in the previous section. When the entity does receive such an exception, what may be concluded about the state of the lookup service is dependent on the type of exception encountered.

If a bad object exception is received, then the entity can usually conclude that the lookup service is not currently running; for example, it may have been taken down for maintenance. The entity can interpret the occurrence of an indefinite exception as a communication failure that may or may not be only temporary. Finally, entities that encounter a bad invocation exception while interacting with a lookup service should view the lookup service as being in an unknown, possibly corrupt, state, and should discontinue further interaction with that lookup service until the problem is resolved.

Whenever an entity receives any of these exceptions while interacting with a lookup service, the affected lookup service is referred to as *unavailable* or *unreachable*. For most entities, the unavailability of a particular lookup service

should not prevent the entity from continuing its processing, although in other situations an entity might consider at least some of these exceptional conditions unrecoverable. In general, when an entity encounters an unreachable lookup service, the exception or error indicating that the lookup service is unavailable should be caught and handled, usually by requesting that the lookup service be *discarded* (see the next section), and the entity should continue its processing.

## US.2.8 Discarding a Lookup Service

When an already discovered lookup service is removed from the managed set of lookup services, it is said to be *discarded*. The process of discarding a lookup service is initiated either directly or indirectly by the discovering entity itself, or by the sub-process that the entity employs to perform the actual discovery duties.

Whenever a lookup service is discarded, a notification event referencing both the discarded lookup service and the *member groups* to which the lookup service belongs, will be sent to all of the entity's discovery listeners. This event is referred to as a *discarded event*. Since there are a number of situations that can result in a discard request being issued, it may be useful to note that discarded events can be classified by two characteristics:

- ◆ Whether the event was generated as a direct consequence of an explicit request made by the entity itself (*active*), or as a consequence of a determination made by some sub-process of the entity (*passive*)
- ◆ Whether the event is related to communication problems, or lost interest—on the part of the entity—in discovering the affected lookup services

When the occurrence of exceptional conditions causes an entity to conclude that a lookup service is unreachable, the entity typically will request that the lookup service be discarded. When the entity itself requests that a lookup service be discarded, the resulting discarded event may be referred to as an *active communication discarded event*. The term *active* is used because the entity takes specific action to request that the lookup service be discarded; and because the entity cannot communicate with the unreachable lookup service, the event is associated with *communication*.

Whenever the entity makes a request that results in the removal of an element from the relevant managed set of groups or locators, one or more of the lookup services associated with the removed groups or locators may be discarded—even though the lookup services are still reachable. The lookup services may be discarded in this situation because the contents of the sets of groups and locators the



entity wishes to discover may have changed in such a way that one or more of the previously discovered lookup services are no longer of interest to the entity. In this case, if any already discovered lookup service is found to belong to none of the groups in the new managed set of groups, or if its locator no longer belongs to the entity's new managed set of locators, a discarded event is generated and sent to all of the entity's discovery listeners. This type of discarded event may be referred to as a *passive no-interest discarded event* (passive because the entity itself did not explicitly request that the lookup service be discarded).

If the sub-process that the entity uses to perform group (multicast) discovery determines that one of the previously discovered lookup services has stopped sending multicast announcements, that sub-process may discard the lookup service. That is, the sub-process may remove the lookup service from the managed set, and send a discarded event to notify the entity that the lookup service is unavailable. The discarded event sent in this situation is often referred to as a *passive communication discarded event*.

If a lookup service is discarded because it was found to be unreachable (associated with a communication discarded event), that lookup service will be made eligible for re-discovery. In this case, the process of discarding a lookup service — either actively or passively — can be viewed as a mechanism for the removal of stale entries in the managed set of lookup services. The discarded lookup service is effectively “marked” for re-discovery, removing the need for operations such as lease renewal attempts on a lookup service that is currently unavailable. Upon re-discovery of the discarded lookup service, the entity typically processes the re-discovered lookup service as if it were discovered for the first time.

Any lookup service corresponding to a no-interest discarded event is no longer eligible for discovery until the entity changes its managed set of locators or its managed set of groups to include, respectively, either the discarded lookup service's locator or at least one of its member groups.

An event related to the discarded event is referred to as a *changed event*. This event notifies the entity of changes in the contents of the member groups of one or more of the lookup services in the managed set. If the entity registers interest in such an event, and if the sub-process that the entity uses to perform group discovery determines that one or more of those member group sets has indeed changed, then a changed event is sent.

### US.2.8.1 Remote Objects, Stubs, and Proxies

The *Jini™ Technology Glossary* states that a *remote object* is an object whose methods can be invoked from a Java virtual machine (JVM)<sup>1</sup>, potentially on a dif-

ferent host. Furthermore, the glossary states that such an object is described by one or more *remote interfaces*.

When invoking methods remotely through Java Remote Method Invocation (RMI), it is useful to think of the invocation as consisting of two components: a client component and a server component. When the client component initiates a remote method call, the server component carries out the execution of the remote method, and RMI facilitates the necessary communication between the two parties. Note that when discussing concepts related to RMI, the term *server* (or *remote server*) is sometimes used in place of the term *remote object*.

In order to initiate the invocation of a remote method, the client must have access to an object referred to as the *stub* of the remote object. The stub is an object local to the client that acts as the “representative” of the remote object. The stub implements the same set of remote interfaces that the remote object implements. From the point of view of the client, the stub *is* the remote object. When the client invokes a method on the local stub, communication with the remote object occurs, resulting in the execution of the corresponding method in the remote object’s JVM.

The term *proxy* is used extensively throughout the helper utilities and services specifications. With respect to remote objects in general, and entities operating within a Jini application environment in particular, a proxy is simply an intermediary object through which one entity (the client) may request the invocation of the methods provided by another entity (the remote object or the service).

Proxies can take a number of different forms. A *smart proxy* typically consists of a set of local methods and a set of one or more remote object references (stubs). Clients invoke one or more of the local methods to access the methods of the remote objects referenced in the proxy.

Another form that a proxy can take is that of the stub of a remote object. That is, all stubs are simply proxies to their corresponding remote objects. Except for the local methods `equals` and `hashCode`, this type of proxy consists of remote methods only.

Some proxies are implemented as *strictly local*. Proxies of this form consist of only local methods, each executing in the client’s JVM. Unlike smart proxies, no remote invocations result when any method of a strictly local proxy is invoked.

Typically, Jini services provide a proxy that has one of the forms described above. When a service registers with a lookup service, the service’s proxy is copied (through serialization) into the lookup service. When a client looks up the service, the service’s proxy is downloaded to the client. The client can then invoke

---

<sup>1</sup> As used in this document, the terms “Java virtual machine” or “JVM” mean a virtual machine for the Java platform.

the methods contained in the service's proxy. If the invoked method is a local method, then execution will occur in the JVM of the client. If the invoked method is a remote method (or results in a remote invocation), then execution is initiated in the client's JVM, but ultimately occurs in the JVM of the service.

Note that the term *front end proxy* (or simply, *front end*) is often used interchangeably with the term proxy. Similarly, the term *back end server* (or simply, *back end*) is often used interchangeably with the term remote object. Thus, the backend of a service is the part of the service's implementation that satisfies the contract advertised in the service's remote interface.

## US.2.9 Activation

The glossary defines the term *active object* as a remote object that is instantiated and exported in a JVM on some system. Remote objects can be implemented with the ability to change their state from inactive to active, or from active to inactive; the process of doing so is referred to as *activation* or *deactivation*, respectively. Many Jini services wishing to conserve computational resources may find this capability desirable. When the back end of any Jini service is implemented with the ability to activate and deactivate, the service is referred to as an *activatable service*. Refer to the *Java™ Remote Method Invocation Specification* for the details of activation.



---

## US.3 Helper Utility Specifications

### US.3.1 The Jini Discovery Utilities Specification

THE *Jini™ Discovery Utilities Specification* defines a set of general purpose utility interfaces collectively referred to as the *Discovery Management Interfaces*. Those interfaces define the policies to apply when implementing helper utilities that manage an entity's discovery duties. Currently, the set of discovery management interfaces consists of the following three interfaces:

- ◆ `DiscoveryManagement`
- ◆ `DiscoveryGroupManagement`
- ◆ `DiscoveryLocatorManagement`

Because the discovery management interfaces provide a uniform way to define utility classes that perform discovery-related management duties on behalf of an entity, the discovery utilities specification defines a number of helper utility classes that implement one or more of these interfaces. Those classes are:

- ◆ `LookupDiscovery`
- ◆ `LookupLocatorDiscovery`
- ◆ `LookupDiscoveryManager`

The discovery utilities specification closes with a discussion of a set of low-level utility classes that can be useful when applying the discovery management policies to build higher-level helper utilities for discovery. Those classes are:

- ◆ `Constants`
- ◆ `OutgoingMulticastRequest`
- ◆ `IncomingMulticastRequest`
- ◆ `OutgoingMulticastAnnouncement`

- ◆ IncomingMulticastAnnouncement
- ◆ OutgoingUnicastRequest
- ◆ IncomingUnicastRequest
- ◆ OutgoingUnicastResponse
- ◆ IncomingUnicastResponse

### **US.3.1.1 The DiscoveryManagement Interface**

The `DiscoveryManagement` interface defines methods related to the discovery event mechanism and discovery process termination. Through this interface an entity can register or un-register `DiscoveryListener` objects to receive discovery events, retrieve proxies to the currently discovered lookup services, discard a lookup service so that it is eligible for re-discovery, or terminate the discovery process.

### **US.3.1.2 The DiscoveryGroupManagement Interface**

The `DiscoveryGroupManagement` interface defines methods and constants related to the management of the set containing the names of the groups whose members are the lookup services that are to be discovered via group discovery. The methods of this interface define how an entity retrieves or modifies the managed set of groups to discover.

### **US.3.1.3 The DiscoveryLocatorManagement Interface**

The `DiscoveryLocatorManagement` interface defines methods related to the management of the set of `LookupLocator` objects corresponding to the specific lookup services that are to be discovered via locator discovery. The methods of this interface define how an entity retrieves or modifies the managed set of locators to discover.

### **US.3.1.4 The LookupDiscovery Helper Utility**

The `LookupDiscovery` helper utility encapsulates the functionality required of an entity that wishes to employ multicast discovery to discover a lookup service located within the entity's *multicast radius*. This utility provides an implementation that makes the process of acquiring lookup service instances, based on no

information other than group membership, which is much simpler for both services and clients.

#### **US.3.1.5 The LookupLocatorDiscovery Helper Utility**

The `LookupLocatorDiscovery` helper utility encapsulates the functionality required of an entity that wishes to employ the unicast discovery protocol to discover a lookup service. This utility provides an implementation that makes the process of finding specific instances of a lookup service much simpler for both services and clients.

#### **US.3.1.6 The LookupDiscoveryManager Helper Utility**

The `LookupDiscoveryManager` is a helper utility class that organizes and manages all discovery-related activities on behalf of a Jini client or service. Rather than providing its own facility for coordinating and maintaining all of the necessary state information related to group names, locators, and listeners, such an entity can employ this class to provide those facilities on its behalf.

#### **US.3.1.7 The Constants Class**

The `Constants` class provides easy access to defined constants that may be useful when participating in the discovery process.

#### **US.3.1.8 The OutgoingMulticastRequest Utility**

The `OutgoingMulticastRequest` class provides facilities for marshalling multicast discovery requests into a form suitable for transmission over a network to announce one's interest in discovering a lookup service.

#### **US.3.1.9 The IncomingMulticastRequest Utility**

The facilities provided by the `IncomingMulticastRequest` class encapsulate the details of the process of unmarshalling received multicast discovery requests into a form in which the individual parameters of the request may be easily accessed.

**US.3.1.10 The OutgoingMulticastAnnouncement Utility**

The `OutgoingMulticastAnnouncement` class encapsulates the details of the process of marshalling multicast discovery announcements into a form suitable for transmission over a network to announce the availability of a lookup service to interested parties.

**US.3.1.11 The IncomingMulticastAnnouncement Utility**

The `IncomingMulticastAnnouncement` class encapsulates the details of the process of unmarshalling multicast discovery announcements into a form in which the individual parameters of the announcement may be easily accessed.

**US.3.1.12 The OutgoingUnicastRequest Utility**

The `OutgoingUnicastRequest` class encapsulates the details of the process of marshalling unicast discovery requests into a form suitable for transmission over a network to attempt discovery of a specific lookup service.

**US.3.1.13 The IncomingUnicastRequest Utility**

The `IncomingUnicastRequest` class encapsulates the details of the process of unmarshalling unicast discovery requests into a form in which the individual parameters of the request may be easily accessed.

**US.3.1.14 The OutgoingUnicastResponse Utility**

The `OutgoingUnicastResponse` class encapsulates the details of the process of marshalling a unicast discovery response into a form suitable for transmission over a network to respond to a unicast discovery request.

**US.3.1.15 The IncomingUnicastResponse Utility**

The `IncomingUnicastResponse` class encapsulates the details of the process of unmarshalling a unicast discovery response into a form in which the individual parameters of the request may be easily accessed.



## US.3.2 The Jini Lease Utilities Specification

*The Jini Lease Utilities Specification* defines helper utility classes, along with supporting interfaces and supporting classes, that encapsulate functionality which provides for the coordination, systematic renewal, and overall management of a set of leases associated with some object on behalf of another object. Currently, this specification defines only one helper utility class:

- ◆ `LeaseRenewalManager`

### US.3.2.1 The `LeaseRenewalManager` Helper Utility

The `LeaseRenewalManager` is a helper utility class that organizes and manages all of the activities related to the renewal of the leases granted to a Jini client or service by another Jini service. Rather than providing its own facility for coordinating and maintaining all of the necessary state information related to lease renewal, such an entity can employ this class to provide those facilities on its behalf.

## US.3.3 The Jini Join Utilities Specification

*The Jini Join Utilities Specification* defines helper utility classes, supporting interfaces, and supporting classes, that encapsulate functionality related to discovery and registration interactions that a well-behaved Jini service will typically have with a Jini lookup service. Currently, this specification defines only one helper utility class:

- ◆ `JoinManager`

### US.3.3.1 The `JoinManager` Helper Utility

The `JoinManager` is a helper utility class that performs all of the functions related to lookup service discovery, joining, lease renewal, and attribute management—functions which the Jini programming model requires of a well-behaved Jini service. Rather than providing its own facility for providing such functions, a Jini service can employ this class to provide those facilities on its behalf.

### US.3.4 The Jini Service Discovery Utilities Specification

*The Jini Service Discovery Utilities Specification* defines helper utility classes, supporting interfaces, and supporting classes that encapsulate functionality that aids a Jini service or client in acquiring services of interest, registered with the various lookup services with which the service or client wishes to interact. Currently, the service discovery utilities specification defines only one helper utility class:

◆ `ServiceDiscoveryManager`

#### US.3.4.1 The `ServiceDiscoveryManager` Helper Utility

The `ServiceDiscoveryManager` class is a helper utility class that any entity can use to create and populate a cache of service references, and with which the entity can register for notification of the availability of services of interest. Although the `ServiceDiscoveryManager` performs lookup discovery event handling for clients and services, the primary functionality the `ServiceDiscoveryManager` provides is service discovery and management.

The `ServiceDiscoveryManager` class can be asked to “discover” services an entity is interested in using, and to cache the references to those services as each is found. The cache can be viewed as a set of services that the entity can access through a set of public, non-remote methods.

The `ServiceDiscoveryManager` class also provides a mechanism for an entity to request notification when a service of interest is discovered for the first time or has encountered a state change (such as removal from all lookup services or attribute set changes).

For convenience, the `ServiceDiscoveryManager` class also provides versions of a method named `lookup`, which employs invocation semantics similar to the semantics of the `lookup` method of the `ServiceRegistrar` interface, specified in the *Jini™ Lookup Service Specification*. Entities needing to find services on only an infrequent basis, or in which the cost of making a remote call is outweighed by the overhead of maintaining a local cache (e.g., due to limited resources), may find this method useful.

All three mechanisms described above—local queries on the cache, service discovery notification, and remote lookups—employ the same template-matching scheme as that described in the *Jini™ Lookup Service Specification*. Additionally, each mechanism allows the entity to supply an action object referred to as a *filter*. Such an object is a non-remote object that defines additional matching criteria that will be applied when searching for the entity’s services of interest. This filtering

facility is particularly useful to entities that wish to extend the capabilities of the standard template-matching scheme.



---

## US.4 Helper Service Specifications

### US.4.1 The Jini Lookup Discovery Service Specification

UNDER certain circumstances, a discovering entity may find it useful to allow a third party to perform the entity's discovery duties. For example, an activatable entity that wishes to deactivate may wish to employ a separate Jini helper service to perform discovery duties on the entity's behalf. Such an entity may wish to deactivate for various reasons, one being to conserve computational resources. While the entity is deactivated, the helper service, running on the same or a separate host, would employ the Jini discovery protocols to find lookup services in which the entity has expressed interest, and would notify the entity when a previously unavailable lookup service becomes available. Such a helper service is referred to as a *lookup discovery service*.

The `LookupDiscoveryService` interface defines the lookup discovery helper service. Through that interface, other Jini services and clients may request that discovery processing be performed on their behalf.

### US.4.2 The Jini Lease Renewal Service Specification

The Jini *lease renewal service*—defined by the `net.jini.lease.LeaseRenewalService` interface—is a Jini helper service that can be employed by both Jini clients and services to perform all lease renewal duties on their behalf. Services that wish to remain inactive until they are needed may find the lease renewal service quite useful. Such a service can request that the lease renewal service take on the responsibility of renewing the leases granted to the service, and then safely deactivate without risking the loss of access to the resources corresponding to the leases being renewed.

Entities that have continuous *access* to a network, but which cannot be continuously *connected* to that network (e.g., a cell phone), may also find this service useful. By allowing a lease renewal service (which can be continuously connected) to renew the leases on the resources acquired by the entity, the entity may

remain disconnected until needed. This lease renewal service removes the need to perform the discovery and lookup process each time the entity re-connects to the network, potentially resulting in a significant increase in efficiency.

### **US.4.3 The Jini Event Mailbox Service Specification**

The Jini *event mailbox service*—defined by the `net.jini.event.EventMailbox` interface—is a Jini helper service that can be employed by entities to store event notifications on their behalf. When an entity registers with the event mailbox service, that service will collect events intended for the registered entity until the entity initiates delivery of the events.

A service such as the event mailbox service can be particularly useful to entities that desire more control over the delivery of the events sent to them. Some entities operating in a distributed system may find it undesirable or inefficient to be contacted solely for the purpose of having an event delivered, preferring to defer the delivery to a time that is more convenient, as determined by the entity itself.

For example, an entity wishing to deactivate or detach from a network may wish to have its events stored until the entity is available to retrieve them. Additionally, some entities may wish to batch process event notifications for efficiency. In both scenarios, the entities described may find the event mailbox service useful in achieving their respective event delivery goals.

---

## US.5 Dependencies

**T**HE helper utilities and services specifications rely on one or more of the following specifications:

- ◆ *Java™ Remote Method Invocation Specification*
- ◆ *Java™ Object Serialization Specification*
- ◆ *Jini™ Technology Glossary*
- ◆ *Jini™ Distributed Event Specification*
- ◆ *Jini™ Distributed Leasing Specification*
- ◆ *Jini™ Discovery and Join Specification*
- ◆ *Jini™ Lookup Service Specification*
- ◆ *Jini™ Lookup Attribute Schema Specification*
- ◆ *Jini™ Transaction Specification*





# DU

---

## Jini™ Discovery Utilities Specification

### DU.0 Introduction

**E**ACH discovering entity in a Java™ virtual machine (JVM)<sup>1</sup> on a given host is independently responsible for obtaining references to lookup services. In this specification we first cover a set of *discovery management interfaces* that define the policies to apply when implementing helper utilities that manage an entity's discovery duties: in particular, the management of multicast (group) discovery and unicast (locator) discovery. After the discovery management interfaces are defined, a set of standard helper utility classes that implement one or more of those interfaces is presented. This specification closes with a discussion of a set of lower-level utility classes that can be useful when applying the discovery management policies to build higher-level helper utilities for discovery.

#### DU.0.1 Dependencies

This specification relies on the following other specifications:

- ◆ *Java™ Object Serialization Specification*
- ◆ *Jini™ Lookup Service Specification*
- ◆ *Jini™ Discovery and Join Specification*

---

<sup>1</sup> As used in this document, the terms “Java virtual machine” or “JVM” mean a virtual machine for the Java platform.

## **DU.0.2 Comments**

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

---

## DU.1 The Discovery Management Interfaces

### DU.1.1 Overview

**D**ISCOVERY is one behavior that is common to all entities wishing to interact with a Jini lookup service. Whether an entity is a client, a service, or a service acting as a client, the entity must first discover a lookup service, before the entity can begin interacting with that lookup service.

The interfaces collectively referred to as the *discovery management* interfaces specify sets of methods that define a mechanism that may be used to manage various aspects of the discovery duties of entities that wish to participate in an application environment for Jini technology (a *Jini application environment*). These interfaces provide a uniform way to define utility classes that perform the necessary discovery-related management duties on behalf of a client or service. Currently, there are three discovery management interfaces belonging to the package `net.jini.discovery`:

- ◆ `DiscoveryManagement`
- ◆ `DiscoveryGroupManagement`
- ◆ `DiscoveryLocatorManagement`

The `DiscoveryManagement` interface defines semantics for methods related to the discovery event mechanism and discovery process termination. Through this interface, an entity can register or un-register for discovery events, discard a lookup service, or terminate the discovery process.

The `DiscoveryGroupManagement` interface defines methods related to the management of the sets of lookup services that are to be discovered using the multicast discovery protocols (See “The Jini™ Discovery and Join Specification” in *The Jini™ Technology Core Platform Specification*). The methods of this interface define how an entity accesses or modifies the set of groups whose members are lookup services that the entity is interested in discovering through group discovery.

The `DiscoveryLocatorManagement` interface defines methods related to the management of the set of lookup services that are to be discovered using the unicast discovery protocol (as defined in the *Jini™ Discovery and Join Specification*). The methods of this interface define how an entity accesses or modifies the contents of the set of `LookupLocator` objects corresponding to the specific lookup services the entity has targeted for locator discovery.

Although each interface defines semantics for methods involved in the management of the discovery process, the individual roles each interface plays in that process are independent of each other. Because of this independence, there may be scenarios where it is desirable to implement some subset of these interfaces.

For example, a class may wish to implement the functionality defined in `DiscoveryManagement`, but may not wish to allow entities to modify the groups and locators associated with the lookup services to be discovered. Such a class may have a “hard-coded” list of the groups and locators that it internally registers with the discovery process. For this case, the class would implement only `DiscoveryManagement`.

Alternatively, another class may not wish to allow the entity to register more than one listener with the discovery event mechanism; nor may it wish to allow the entity to terminate discovery. It may simply wish to allow the entity to modify the sets of lookup services that will be discovered. A class such as this would implement both `DiscoveryGroupManagement` and `DiscoveryLocatorManagement`, but not `DiscoveryManagement`.

A specific example of a class that implements only a subset of the set of interfaces specified here is the `LookupDiscovery` utility class (defined later in this specification). That class implements both the `DiscoveryManagement` and `DiscoveryGroupManagement` interfaces, but not the `DiscoveryLocatorManagement` interface.

Throughout this discussion of the discovery management interfaces, the phrase *implementation class* refers to any concrete class that implements one or more of those interfaces. The phrase *implementation object* should be understood to mean an instance of such an implementation class. Additionally, whenever a description refers to the *discovering entity* (or simply, the *entity*), that phrase is intended to be interpreted as the object (the client or service) that has created an implementation object, and which wishes to use the public methods specified by these interfaces and provided by that object.

## DU.1.2 Other Types

The types defined in the specification of the discovery management interfaces are in the `net.jini.discovery` package. The following additional types may also be

referenced in this specification. Whenever referenced, these object types will be referenced in unqualified form:

```
net.jini.core.discovery.LookupLocator
net.jini.core.lookup.ServiceRegistrar
net.jini.discovery.DiscoveryEvent
net.jini.discovery.DiscoveryListener
net.jini.discovery.DiscoveryChangeListener
net.jini.discovery.LookupDiscovery
net.jini.discovery.LookupDiscoveryManager
java.io.IOException
java.util.EventListener
java.util.EventObject
java.util.Map
```

### DU.1.3 The DiscoveryManagement Interface

The public methods specified by the DiscoveryManagement interface are as follows:

```
package net.jini.discovery;

public interface DiscoveryManagement {
    public void addDiscoveryListener
                                   (DiscoveryListener listener);
    public void removeDiscoveryListener
                                   (DiscoveryListener listener);
    public ServiceRegistrar[] getRegistrars();
    public void discard(ServiceRegistrar proxy);
    public void terminate();
}
```

#### DU.1.3.1 The Semantics

The DiscoveryManagement interface defines methods related to the discovery event mechanism and discovery process termination. Through this interface, an entity can register or un-register DiscoveryListener objects to receive discovery events (instances of DiscoveryEvent), retrieve proxies to the currently discovered lookup services, discard a lookup service so that it is eligible for re-discovery, or terminate the discovery process.

Implementation classes of this interface may impose additional semantics on any method. For example, such a class may choose to require that rather than simply terminate discovery processing, the `terminate` method additionally should cancel all leases held by the implementation object and terminate all lease management being performed on behalf of the entity.

For information on any additional semantics imposed on a method of this interface, refer to the specification of the particular implementation class.

Note that the classes `DiscoveryEvent`, `DiscoveryListener`, and `DiscoveryChangeListener` are defined later in this specification.

- ◆ The `addDiscoveryListener` method adds a listener to the set of objects listening for discovery events. This method takes a single argument as input: an instance of `DiscoveryListener` corresponding to the listener to add to the set.

Once a listener is registered, it will be notified of all lookup services discovered to date, and will then be notified as new lookup services are discovered or existing lookup services are discarded.

If the added listener is also an instance of `DiscoveryChangeListener` (a subclass of `DiscoveryListener`), then in addition to receiving events related to discovered and discarded lookup services, that listener will also be notified of group membership changes that occur in any of the lookup services targeted for at least group discovery.

If `null` is input to this method, no action will be taken. If the listener input to this method duplicates (using the `equals` method) another element in the set of listeners, no action is taken.

- ◆ The `removeDiscoveryListener` method removes a listener from the set of objects listening for discovery events. This method takes a single argument as input: an instance of `DiscoveryListener` corresponding to the listener to remove from the set.

If the listener object input to this method does not exist in the set of listeners maintained by the implementation class, then this method will take no action.

---

**Note:** Should pending events be addressed with respect to invocations of `removeDiscoveryListener`? For example, should a requirement like the following be made: "Once an invocation of `removeDiscoveryListener` returns, it is guaranteed that none of the removed listener(s) will receive any pending notification events." This question is asked because some of the current implementations of the discovery management interfaces (e.g., `LookupDiscovery`, `LookupLocatorDiscovery`, `LookupDiscoveryManager`) queue the refer-

ences to the listeners along with the pending events (for efficiency). So, even if a listener is removed, a reference to it still exists in the event queue; and the listener may be notified when the event finally gets to the front of the queue. So should this requirement be specified—forcing existing implementations to be changed—or is it too implementation specific to include here?

---

- ◆ The `getRegistrars` method returns an array consisting of instances of the `ServiceRegistrar` interface. Each element in the returned set is a proxy to one of the currently discovered lookup services. Each time this method is invoked, a new array is returned. If no lookup services have been discovered, an empty array is returned. This method takes no arguments as input.
- ◆ The `discard` method removes a particular lookup service from the managed set of lookup services, and makes that lookup service eligible to be re-discovered. This method takes a single argument as input: an instance of the `ServiceRegistrar` interface corresponding to the proxy to the lookup service to discard.

If the proxy input to this method is `null`, or if it matches (using the `equals` method) none of the lookup services in the managed set, this method takes no action.

Currently, there exist utilities such as the `LookupDiscovery` and `LookupDiscoveryManager` helper utilities that will, on behalf of a discovering entity, automatically discard a lookup service upon determining that the lookup service has become unreachable or uninteresting. Although most entities will typically employ such a utility to help with both its discovery as well as its discard duties, it is important to note that if the entity itself determines that the lookup service is unavailable, it is the responsibility of the entity to invoke the `discard` method. This scenario usually happens when the entity attempts to interact with a lookup service, but encounters an exceptional condition (for example, a communication failure). When the entity actively discards a lookup service, the discarded lookup service becomes eligible to be re-discovered. Allowing unreachable lookup services to remain in the managed set can result in repeated and unnecessary attempts to interact with lookup services with which the entity can no longer communicate. Thus, the mechanism provided by this method is intended to provide a way to remove such “stale” lookup service references from the managed set.

Invoking the `discard` method defined by the `DiscoveryManagement` interface will result in the flushing of the lookup service from the appropriate cache, ultimately causing a discard notification—referred to as a *discarded event*—to be sent to all listeners registered with the implementation object.

When this method completes successfully, the lookup service is guaranteed to have been removed from the managed set, and the lookup service is then said to have been “discarded”. No such guarantee is made with respect to when the discarded event is sent to the registered listeners. That is, the event notifying the listeners that the lookup service has been discarded may or may not be sent asynchronously.

- ◆ The `terminate` method ends all discovery processing being performed on behalf of the entity. This method takes no input arguments.

After this method has been invoked, no new lookup services will be discovered, and the effect of any new operations performed on the current implementation object are undefined.

Any additional termination semantics must be defined by the implementation class.

## DU.1.4 The `DiscoveryGroupManagement` Interface

The public methods specified by the `DiscoveryGroupManagement` interface are as follows:

```
package net.jini.discovery;

public interface DiscoveryGroupManagement {
    public static final String[] ALL_GROUPS = null;
    public static final String[] NO_GROUPS = new String[0];

    public String[] getGroups();
    public void addGroups(String[] groups) throws IOException;
    public void setGroups(String[] groups) throws IOException;
    public void removeGroups(String[] groups);
}
```

### DU.1.4.1 The Semantics

The `DiscoveryGroupManagement` interface defines methods and constants related to the management of the set containing the names of the groups whose members are the lookup services that are to be discovered using the multicast discovery protocols; that is, lookup services that are discovered by way of group discovery. The methods of this interface define how an entity retrieves or modifies the managed set of groups to discover, where phrases such as “the groups to discover” or “dis-



covering the desired groups” refer to the discovery of the lookup services that are members of those groups.

The methods that modify the managed set of groups each take a single input parameter: a `String` array, none of whose elements may be `null`. Each of these methods throws a `NullPointerException` when at least one element of the input array is `null`.

The empty set is denoted by the empty array, and “no set” is indicated by `null`. Invoking any of these methods with an input array that contains duplicate group names is equivalent to performing the invocation with the duplicates removed from the array.

The `ALL_GROUPS` and the `NO_GROUPS` constants are defined for convenience, and represent no set and the empty set respectively.

- ◆ The `getGroups` method returns an array consisting of the names of the groups in the managed set; that is, the names of the groups the implementation object is currently configured to discover.

If the managed set of groups is empty, this method will return the empty array. If there is no managed set of groups, then `null` (`ALL_GROUPS`) is returned, indicating that any lookup service within range—even those that have no group affiliation—are to be discovered.

If the empty array is returned, that array is guaranteed to be referentially equal to the `NO_GROUPS` constant; that is, the array returned from that method and the `NO_GROUPS` constant can be tested for equality using the `==` operator.

This method takes no arguments as input and, provided the managed set of groups currently exists, will return a new array upon each invocation.

- ◆ The `addGroups` method adds a set of group names to the managed set. The array input to this method contains the group names to be added to the set.

This method throws `IOException` because an invocation of this method may result in the re-initiation of the discovery process, which can throw `IOException` when socket allocation occurs.

This method throws an `UnsupportedOperationException` if there is no managed set of groups to augment, and it throws a `NullPointerException` if `null` (`ALL_GROUPS`) is input. If the empty array (`NO_GROUPS`) is input, the managed set of groups will not change.

- ◆ The `setGroups` method replaces all of the group names in the managed set with names from a new set. The array input to this method contains the group names with which to replace the current names in the managed set.

Once a new group name has been placed in the managed set, no event will be sent to the entity's listener for the lookup services belonging to that group that have already been discovered, although attempts to discover all (as yet) undiscovered lookup services belonging to that group will continue to be made.

If `null` (`ALL_GROUPS`) is input to `setGroups`, then attempts will be made to discover all (as yet) undiscovered lookup services located within the *multicast radius* (See Section DU.2, "LookupDiscovery Utility") of the implementation object, regardless of group membership.

If the empty array (`NO_GROUPS`) is input to `setGroups`, then group discovery will be halted until the managed set of groups is changed—through a subsequent call to this method or to `addGroups`—to a set that is either a non-empty set of group names or `null` (`ALL_GROUPS`).

This method throws `IOException`. This is because an invocation of this method may result in the re-initiation of the discovery process, a process that can throw `IOException` when socket allocation occurs.

- ◆ The `removeGroups` method deletes a set of group names from the managed set of groups. The array input to this method contains the group names to be removed from the managed set.

This method throws an `UnsupportedOperationException` if there is no managed set of groups from which to remove elements. If `null` (`ALL_GROUPS`) is input to `removeGroups`, a `NullPointerException` will be thrown.

If any element of the set of groups to be removed is not contained in the managed set, `removeGroups` takes no action with respect to that element. If the empty array (`NO_GROUPS`) is input, the managed set of groups will not change.

Once a new group name is added to the managed set as a result of an invocation of either `addGroups` or `setGroups`, attempts will be made—using the multicast request protocol—to discover all (as yet) undiscovered lookup services that are members of that group. If after a specified amount of time there are no responses to the multicast requests, the implementation object will stop sending multicast requests, and will simply listen for multicast announcements containing the new groups of interest.

Any already discovered lookup service that is a member of one or more of the groups removed from the managed set as a result of an invocation of either `setGroups` or `removeGroups` will be discarded and will no longer be eligible for discovery, but only if that lookup service satisfies both of the following conditions:

- the lookup service is not a member of any group in the new managed set that resulted from the invocation of `setGroups` or `removeGroups`, and
- the lookup service is not currently eligible for discovery through other means (such as locator discovery).

## DU.1.5 The DiscoveryLocatorManagement Interface

The public methods specified by the `DiscoveryLocatorManagement` interface are as follows:

```
package net.jini.discovery;

public interface DiscoveryLocatorManagement {
    public LookupLocator[] getLocators();
    public void addLocators(LookupLocator[] locators);
    public void setLocators(LookupLocator[] locators);
    public void removeLocators(LookupLocator[] locators);
}
```

### DU.1.5.1 The Semantics

The `DiscoveryLocatorManagement` interface defines methods related to the management of the set of `LookupLocator` objects corresponding to the specific lookup services that are to be discovered using the unicast discovery protocol; that is, lookup services that are discovered by way of locator discovery. The methods of this interface define how an entity retrieves or modifies the managed set of locators to discover. Phrases such as “the locators to discover” and “discovering the desired locators” refer to the discovery of the lookup services that are associated with those locators.

The methods that modify the managed set of locators each take a single input parameter: an array of `LookupLocator` objects, none of whose elements may be `null`. Each of these methods throws a `NullPointerException` when at least one element of the input array is `null`.

Invoking any of these methods with an input array that contains duplicate locators (as determined by `LookupLocator.equals`) is equivalent to performing the invocation with the duplicates removed from the array.

- ◆ The `getLocators` method returns an array containing the set of `LookupLocator` objects in the managed set of locators; that is, the locators

of the specific lookup services that the implementation object is currently interested in discovering.

The returned set includes both the set of locators corresponding to lookup services that have already been discovered and the set of those that have not yet been discovered.

If the managed set is empty, this method returns the empty array. This method takes no arguments as input, and returns a new array upon each invocation.

- ◆ The `addLocators` method adds a set of locators to the managed set. The array input to this method contains the set of `LookupLocator` objects to add to the managed set.

If `null` is input to `addLocators`, a `NullPointerException` will be thrown. If the empty array is input, the managed set of locators will not change.

- ◆ The `setLocators` method replaces all of the locators in the managed set with `LookupLocator` objects from a new set. The array input to this method contains the set of `LookupLocator` objects with which to replace the current locators in the managed set.

If `null` is input to `setLocators`, a `NullPointerException` will be thrown.

If the empty array is input to `setLocators`, then locator discovery will be halted until the managed set of locators is changed—through a subsequent call to this method or to `addLocators`—to a set that is non-`null` and non-empty.

- ◆ The `removeLocators` method deletes a set of locators from the managed set. The array input to this method contains the set of `LookupLocator` objects to remove from the managed set.

If `null` is input to `removeLocators`, a `NullPointerException` will be thrown.

If any element of the set of locators to remove is not contained in the managed set, `removeLocators` takes no action with respect to that element. If the empty array is input, the managed set of locators will not change.

Any already discovered lookup service, corresponding to a locator that is a member of the set of locators removed from the managed set as a result of an invocation of either `setLocators` or `removeLocators`, will be discarded and will no longer be eligible for discovery; but only if it is not currently eligible for discovery through other means (such as group discovery).

## DU.1.6 Supporting Interfaces and Classes

Discovery management depends on the interfaces `DiscoveryListener` and `DiscoveryChangeListener`, and on the concrete class `DiscoveryEvent`.

### DU.1.6.1 The `DiscoveryListener` Interface

The public methods specified by the `DiscoveryListener` interface are as follows:

```
package net.jini.discovery;

public interface DiscoveryListener extends EventListener {
    public void discovered(DiscoveryEvent e);
    public void discarded(DiscoveryEvent e);
}
```

When an entity employs an object that implements one or more of the discovery management interfaces to perform and manage the entity's discovery duties, the entity often will want that object—generally referred to as a *discovery utility*—to notify the entity when a desired lookup service is either discovered or discarded. The `DiscoveryListener` interface defines a mechanism through which an entity may receive such notifications from a discovery utility. When an entity registers interest in these notifications, an implementation of this interface must be provided to the discovery utility being employed. Through this registered listener, the entity may then receive instances of the `DiscoveryEvent` class, which encapsulate the required information associated with the desired notifications.

### The Semantics

The events received by listeners implementing the `DiscoveryListener` interface can be the result of either group discovery or locator discovery. These events contain the discovered or discarded registrars, as well as the set of member groups corresponding to each registrar (see the specification of the `DiscoveryEvent` class).

- ◆ The `discovered` method is called whenever a new lookup service is discovered or a discarded lookup service is re-discovered.
- ◆ The `discarded` method is called whenever a previously discovered lookup service is discarded because the lookup service was determined to be either

unreachable or no longer interesting to the entity, and the discard process was initiated by either the entity itself (an *active* discard) or the discovery utility employed by the entity (a *passive* discard).

### DU.1.6.2 The DiscoveryChangeListener Interface

The DiscoveryChangeListener interface specifies only one public method:

```
package net.jini.discovery;

public interface DiscoveryChangeListener
    extends DiscoveryListener
{
    public void changed(DiscoveryEvent e);
}
```

In addition to being notified when a desired lookup service is discovered or discarded, some entities may also wish to be notified when a lookup service experiences changes in its group membership. The DiscoveryChangeListener interface defines an extension to the DiscoveryListener interface, providing a mechanism through which an entity may receive these additional notifications—referred to as *changed events*. As with the DiscoveryListener interface, when an entity wishes to receive changed events in addition to discovered and discarded events, an implementation of this interface must be provided to the discovery utility being employed. It is through that registered listener that the entity receives the desired notifications encapsulated in instances of the DiscoveryEvent class.

### The Semantics

When the entity receives a DiscoveryEvent object through an instance of the DiscoveryChangeListener interface, the event contains the discovered, discarded, or changed registrars, as well as the set of member groups corresponding to each registrar. In the case of a changed event, each set of groups referenced in the event contains the new groups in which the corresponding registrar is a member.

The changed method is called whenever the discovery utility encounters changes in the set of group names contained in the multicast announcements from a previously discovered lookup service.

It is important to note that instances of this interface are eligible to receive changed events for only those lookup services that the entity has requested be discovered by (at least) group discovery. That is, if the entity requests that *only* loca-

tor discovery be used to discover a specific lookup service, the listener will receive no changed events for that lookup service. This is because the semantics of this interface assume that since the entity expressed no interest in *discovering* the lookup service through its group membership, it must also have no interest in any *changes* in that lookup service's group membership. Thus, if an entity wishes to receive changed events for one or more lookup services, the entity must request that those lookup services be discovered by either group discovery alone, or by both group and locator discovery.

### DU.1.6.3 The DiscoveryEvent Class

The public methods provided by the DiscoveryEvent class are as follows:

```
package net.jini.discovery;

public class DiscoveryEvent extends EventObject {
    public DiscoveryEvent(Object source, Map groups) {...}
    public DiscoveryEvent(Object source,
                          ServiceRegistrar[] regs) {...}
    public Map getGroups() {...}
    public ServiceRegistrar[] getRegistrars() {...}
}
```

The DiscoveryEvent class provides an encapsulation of event information that discovery utilities can use to notify an entity of the occurrence of an event involving one or more ServiceRegistrar objects (lookup services) in which the entity has registered interest. Discovery utilities pass an instance of this class to the entity's discovery listener(s) when one of the following events occurs:

- ◆ Each lookup service referenced in the event has been discovered for the first time, or re-discovered after having been discarded.
- ◆ Each lookup service referenced in the event has been either actively or passively discarded.
- ◆ For each lookup service referenced in the event, the set of groups in which the lookup service is a member has changed.

The DiscoveryEvent class is a subclass of EventObject, adding the following additional items of abstract state: a set of ServiceRegistrar instances (*registrars*) referencing the affected lookup services, and a set that maps each of those registrars to their current set of member groups. Methods are defined through

which this additional state may be retrieved upon receipt of an instance of this class.

### The Semantics

The `equals` method for this class returns `true` if and only if two instances of this class refer to the same object. That is, `x` and `y` are equal instances of this class if and only if `x == y` has the value `true`.

The constructor for this class has two forms, where both forms expect two input parameters. Each form of the constructor takes, as its first input parameter, a reference to the source of the event; that is, the discovery utility object that created the event instance and sent it to the entity's listener(s) through the invocation of the `discovered`, `discarded`, or `changed` method on each listener. Depending on the constructor employed, the second parameter is one of the following:

- ◆ A `Map` instance in which each element of the map's key set is a `ServiceRegistrar` instance that references one of the lookup services to be associated with the event being constructed. Each element of the map's value set is a `String` array, containing the names of the groups in which the corresponding lookup service is a member.
- ◆ An array of `ServiceRegistrar` instances in which each element references one of the lookup services to be associated with the event being constructed.

It is important to note that when this form of the constructor is used to construct a `DiscoveryEvent`, each registrar contained in the resulting event is associated with a `null` group set. Therefore, discovery utilities should no longer use this constructor to instantiate the events they send.

With respect to the public methods of this class, the semantics are as follows:

- ◆ The `getGroups` method returns the set that maps each registrar, referenced by the event, to the registrar's current set of member groups.

The returned map's key set is made up of `ServiceRegistrar` instances corresponding to the lookup services for which the event was constructed and sent. Each element of the returned map's value set is a `String` array, containing the names of the member groups of the corresponding lookup service.

On each invocation of this method, the same `Map` object is returned; that is, a copy is not made.

- ◆ The `getRegistrars` method returns an array of `ServiceRegistrar` instances, in which each element references one of the lookup services for which the event was constructed and sent.



On each invocation of this method, the same array is returned; that is, a copy is not made.

**DU.1.7   Serialized Forms**

Class	serialVersionUID	Serialized Fields
DiscoveryEvent	5280303374696501479L	ServiceRegistrar[] regs Map groups



---

## DU.2 LookupDiscovery Utility

**I**N a Jini application environment, the multicast discovery protocols are often collectively referred to as multicast discovery or group discovery. The entities that participate in the multicast discovery protocol are a *discovering entity* (Jini client or service) and a Jini lookup service, which acts as the entity that is to be discovered. When the discovering entity starts, it uses the multicast request protocol to announce its interest in finding lookup services within range. After a specified amount of time, the entity stops sending multicast requests, and simply listens for multicast announcements from any lookup services within range that may be broadcasting their availability. Through either of these protocols, the discovering entity can obtain references to lookup services belonging to member group in which the entity is interested. For the details of the multicast discovery protocols, refer to the *Jini™ Discovery and Join Specification*.

The LookupDiscovery helper utility, belonging to the package `net.jini.discovery`, encapsulates the functionality required of an entity that wishes to employ multicast discovery to discover a lookup service located within the entity's *multicast radius* (roughly, the number of hops beyond which neither the multicast requests from the entity, nor the multicast announcements from the lookup service, will propagate). This utility provides an implementation that makes the process of acquiring lookup service instances, based on no information other than group membership, much simpler for both services and clients.

### DU.2.1 Other Types

The types defined in the specification of the LookupDiscovery utility class are in the `net.jini.discovery` package. The following additional types may also be referenced in this specification. Whenever referenced, these object types will be referenced in unqualified form:

```
net.jini.core.discovery.LookupLocator
net.jini.discovery.DiscoveryManagement
net.jini.discovery.DiscoveryGroupManagement
net.jini.discovery.DiscoveryPermission
java.io.IOException
java.io.Serializable
java.security.Permission
```

## DU.2.2 The Interface

The public methods provided by the LookupDiscovery class are as follows:

```
package net.jini.discovery;

public class LookupDiscovery
    implements DiscoveryManagement,
               DiscoveryGroupManagement
{
    public static final String[] ALL_GROUPS
        = DiscoveryGroupManagement.ALL_GROUPS;
    public static final String[] NO_GROUPS
        = DiscoveryGroupManagement.NO_GROUPS;

    public LookupDiscovery(String[] groups)
        throws IOException {...}
}
```

## DU.2.3 The Semantics

The only new public method of the LookupDiscovery helper utility class is the constructor. All other public methods implemented by this class are specified in the DiscoveryManagement and the DiscoveryGroupManagement interfaces.

Each instance of the LookupDiscovery class must behave as if it operates independently of all other instances.

The equals method for this class returns true if and only if two instances of this class refer to the same object. That is, `x` and `y` are equal instances of this class if and only if `x == y` has the value true.

For convenience, this class defines the constants `ALL_GROUPS` and `NO_GROUPS`, which represent no set and the empty set respectively. For more information on these constants, refer to the specification of the `DiscoveryGroupManagement` interface.

- ◆ The constructor of the `LookupDiscovery` class takes a single input parameter: a `String` array, none of whose elements may be `null`. If at least one element of the input array is `null`, a `NullPointerException` is thrown.

Constructing this class using an input array that contains duplicate group names is equivalent to constructing the class using an array with the duplicates removed.

If `null` (`ALL_GROUPS`) is input to the constructor, then attempts will be made to discover all lookup services located within the current multicast radius, regardless of group membership.

Although discovery events will not be sent by this class until a listener is added through an invocation of the `addListener` method, discovery processing usually starts as soon as an instance of this class is constructed. However, if the empty array (`NO_GROUPS`) is passed to the constructor, discovery will not be started until the `addGroups` or `setGroups` method is called to change the initial empty set of groups to either a non-empty set, or `null` (`ALL_GROUPS`).

Discovery processing in this class ends either when the instance is finalized prior to garbage collection, or when the `terminate` method is called.

The constructor throws `IOException` because the creation of a `LookupDiscovery` object will result in the initiation of the discovery process, a process that can throw `IOException` when socket allocation occurs.

## DU.2.4 Supporting Interfaces and Classes

The `LookupDiscovery` helper utility class depends on the interfaces `DiscoveryManagement` and `DiscoveryGroupManagement`, and on the concrete class `DiscoveryPermission`.

### DU.2.4.1 The `DiscoveryManagement` Interfaces

The `LookupDiscovery` class implements both the `DiscoveryManagement` and the `DiscoveryGroupManagement` interfaces, which together define methods related to the coordination and management of all group discovery processing. (See

Section DU.1, “The Discovery Management Interfaces” for more information on those interfaces.

#### **DU.2.4.2 Security and Multicast Discovery: The `DiscoveryPermission` Class**

When an instance of the `LookupDiscovery` class is constructed, the entity that creates the instance must be granted appropriate discovery permission. For example, if the instance of `LookupDiscovery` is currently configured to discover a non-empty, non-null set of groups, then the entity that created the instance must have permission to attempt discovery of each of the groups in that set. If the set of groups to discover is null (`ALL_GROUPS`), then the entity must have permission to attempt discovery of all possible groups. If appropriate permissions are not granted, the constructor of `LookupDiscovery`, as well as the methods `addGroups` and `setGroups`, will throw a `java.lang.SecurityException`.

Discovery permissions are controlled in security policy files using the permission class `DiscoveryPermission`. The public methods provided by the `DiscoveryPermission` class are as follows:

```
package net.jini.discovery;

public final class DiscoveryPermission extends Permission
                                   implements Serializable
{
    public DiscoveryPermission(String group) {...}
    public DiscoveryPermission(String group,
                               String actions) {...}
}
```

The `DiscoveryPermission` class is a subclass of `Permission`, adding no additional items of abstract state.

#### **The Semantics**

The `equals` method for this class returns `true` if and only if two instances of this class refer to the same object. That is, `x` and `y` are equal instances of this class if and only if `x == y` has the value `true`.

- ◆ The constructor for this class has two forms: one form expecting one input parameter, the other form expecting two input parameters. Each form of the constructor takes, as its first input parameter, a `String` representing one or more group names for which to allow discovery.

The second parameter of the second form of the constructor is a `String` value that is currently ignored because there are no actions associated with a discovery permission.

### DiscoveryPermission Examples

A number of examples that illustrate the use of this permission are presented. Note that each example represents a line in a policy file.

```
permission net.jini.discovery.DiscoveryPermission "*";
    Grant the entity permission to attempt discovery of all possible groups
permission net.jini.discovery.DiscoveryPermission "";
    Grant the entity permission to attempt discovery of only the "public" group
permission net.jini.discovery.DiscoveryPermission "foo";
    Grant the entity permission to attempt discovery of the group named "foo"
permission net.jini.discovery.DiscoveryPermission "/*.sun.com";
    Grant the entity permission to attempt discovery of all groups whose names
    end with the substring ".sun.com"
```

Each of the above declarations grants permission to attempt discovery of one name. A name does not necessarily correspond to a single group. That is, the following should be noted:

- ◆ The name "\*" grants permission to attempt discovery of *all* possible groups.
- ◆ A name beginning with "/\*. " grants permission to attempt discovery of all groups that match the *remainder* of that name; for example, the name "/\*.example.org" would match a group named "foonly.example.org" and also a group named "sf.ca.example.org".
- ◆ The empty name "" denotes the *public* group.
- ◆ All other names are treated as individual groups and must match exactly.

Finally, it is important to note that a restriction of the Java 2 platform security model requires that appropriate `DiscoveryPermission` be granted to the Jini technology infrastructure software codebase itself, in addition to any codebases that may use Jini technology infrastructure software classes.

## DU.2.5 Serialized Forms

Class	serialVersionUID	Serialized Fields
DiscoveryPermission	-3036978025008149170L	<i>none</i>



---

## DU.3 The LookupLocatorDiscovery Utility

### DU.3.1 Overview

THE *Jini™ Discovery and Join Specification* states that the “unicast discovery protocol is a simple request-response protocol”. In a Jini application environment, the entities that participate in this protocol are a discovering entity (Jini client or service) and a Jini lookup service that acts as the entity to be discovered. The discovering entity sends unicast discovery requests to the lookup service, and the lookup service reacts to those requests by sending unicast discovery responses to the interested discovering entity.

The `LookupLocatorDiscovery` helper utility (belonging to the package `net.jini.discovery`) encapsulates the functionality required of an entity that wishes to employ the unicast discovery protocol to discover a lookup service. This utility provides an implementation that makes the process of finding specific instances of a lookup service much simpler for both services and clients.

Because the `LookupLocatorDiscovery` helper utility class will participate in only the unicast discovery protocol, and because the unicast discovery protocol imposes no restriction on the physical location of a service or client relative to a lookup service, this utility can be used to discover lookup services running on hosts that are located far from, or near to, the hosts on which the service is running. This lack of a restriction on location brings with it a requirement that the discovering entity supply specific information about the desired lookup services to the `LookupLocatorDiscovery` utility; namely, the location of the device(s) hosting each lookup service. This information is supplied through an instance of the `LookupLocator` utility, defined in the *Jini™ Discovery and Join Specification*.

It may be of value to note the difference between `LookupLocatorDiscovery` and the `LookupDiscovery` helper utility for group discovery (defined earlier). Although both are non-remote utility classes that entities can use to discover at least one lookup service, the `LookupLocatorDiscovery` utility is designed to provide discovery capabilities that satisfy different needs than those satisfied by the `LookupDiscovery` utility. These two utilities differ in the following ways:

- ◆ Whereas the *LookupLocatorDiscovery* utility is used to discover lookup services by their *locators*, employing the unicast discovery protocol, the *LookupDiscovery* utility uses the multicast discovery protocols to discover lookup services by the *groups* to which the lookup services belong.
- ◆ Whereas the *LookupLocatorDiscovery* utility requires that the discovering entity supply the specific location—or address—of the desired lookup service(s) in the form of a *LookupLocator* object, the *LookupDiscovery* utility imposes no such restriction on the discovering entity.
- ◆ Whereas the *LookupLocatorDiscovery* utility can be used by a discovering entity to discover lookup services that are both “near” and “far”, the *LookupDiscovery* utility can be used to discover only those lookup services that are located within the same multicast radius as that of the discovering entity.

### DU.3.2 Other Types

The types defined in the specification of the *LookupLocatorDiscovery* utility class are in the *net.jini.discovery* package. The following additional types may also be referenced in this specification. Whenever referenced, these object types will be referenced in unqualified form:

```
net.jini.core.discovery.LookupLocator
net.jini.discovery.DiscoveryManagement
net.jini.discovery.DiscoveryLocatorManagement
```

### DU.3.3 The Interface

The public methods provided by the *LookupLocatorDiscovery* class are as follows:

```
package net.jini.discovery;

public class LookupLocatorDiscovery
    implements DiscoveryManagement
               DiscoveryLocatorManagement
{
    public LookupLocatorDiscovery
        (LookupLocator[] locators) {...}
```

```

    public LookupLocator[] getDiscoveredLocators() {...};
    public LookupLocator[] getUndiscoveredLocators() {...};
}

```

### DU.3.4 The Semantics

Including the constructor, the `LookupLocatorDiscovery` helper utility class defines three new public methods. All other public methods implemented by this class are specified in the `DiscoveryManagement` and `DiscoveryLocatorManagement` interfaces.

Each instance of the `LookupLocatorDiscovery` class must behave as if it operates independently of all other instances.

The `equals` method for this class returns `true` if and only if two instances of this class refer to the same object. That is, `x` and `y` are equal instances of this class if and only if `x == y` has the value `true`.

- ◆ The constructor of the `LookupLocatorDiscovery` class takes a single input parameter: a set of locators represented as an array of `LookupLocator` objects, none of whose elements may be `null`. Each element in the input set corresponds to a specific lookup service the discovering entity wishes to be discovered. If at least one element of the input array is `null`, a `NullPointerException` is thrown.

Invoking the constructor with an input array that contains duplicate locators (as determined by `LookupLocator.equals`) is equivalent to performing the invocation with the duplicates removed from the array.

Although discovery events will not be sent by this class until a listener is added through an invocation of the `addListener` method, discovery processing usually starts as soon as an instance of this class is constructed. However, if `null` or the empty array is passed to the constructor, discovery will not be started until the `addLocators` or `setLocators` method is called to change the managed set of locators to a set of locators that is non-`null` and non-empty.

Discovery processing in this class ends either when the instance is finalized prior to garbage collection, or when the `terminate` method is called.

- ◆ The `getDiscoveredLocators` method returns the set of `LookupLocator` objects representing the desired lookup services that are currently discovered. If the set is empty, this method will return the empty array. This method takes no arguments as input, and will return a new array upon each invocation.

- ◆ The `getUndiscoveredLocators` method returns the set of `LookupLocator` objects representing the desired lookup services that have not yet been discovered. If the set is empty, this method will return the empty array. This method takes no arguments as input, and will return a new array upon each invocation.

## **DU.3.5 Supporting Interfaces**

The `LookupLocatorDiscovery` helper utility class depends on the following interfaces: `DiscoveryManagement` and `DiscoveryLocatorManagement`.

### **DU.3.5.1 The `DiscoveryManagement` Interfaces**

The `LookupLocatorDiscovery` class implements both the `DiscoveryManagement` and the `DiscoveryLocatorManagement` interfaces, which together define methods related to the coordination and management of all locator discovery processing. See Section DU.1, "The Discovery Management Interfaces" for more information on those interfaces.

---

## DU.4 The LookupDiscoveryManager Utility

### DU.4.1 Overview

**A**LTHOUGH the goals of any well-behaved Jini client or service are application-specific, the goals of such entities with respect to their interaction with Jini lookup services generally begin with employing the Jini discovery protocols (defined in the *Jini™ Discovery and Join Specification*) to obtain a reference to at least one lookup service. Because the discovery duties performed by such entities may require the management of significant amounts of state information, those duties can become quite tedious.

The `LookupDiscoveryManager` is a helper utility class (belonging to the package `net.jini.discovery`) that organizes and manages all discovery-related activities on behalf of a Jini client or service. Rather than providing its own facility for coordinating and maintaining all of the necessary state information related to group names, `LookupLocator` objects, and `DiscoveryListener` objects, such an entity can employ this class to provide those facilities on its behalf.

### DU.4.2 Other Types

The types defined in the specification of the `LookupDiscoveryManager` utility class are in the `net.jini.discovery` package. The following additional types may also be referenced in this specification. Whenever referenced, these object types will be referenced in unqualified form:

```
net.jini.core.discovery.LookupLocator
net.jini.discovery.DiscoveryEvent
net.jini.discovery.DiscoveryListener
net.jini.discovery.DiscoveryManagement
net.jini.discovery.DiscoveryGroupManagement
net.jini.discovery.DiscoveryLocatorManagement
java.io.IOException
```

### DU.4.3 The Interface

The only new public method of the LookupDiscoveryManager helper utility class is the constructor. All other public methods implemented by this class are specified in the discovery management interfaces.

```
package net.jini.discovery;

public class LookupDiscoveryManager
    implements DiscoveryManagement,
               DiscoveryGroupManagement,
               DiscoveryLocatorManagement
{
    public LookupDiscoveryManager(String[] groups,
                                   LookupLocator[] locators,
                                   DiscoveryListener listener)
        throws IOException {...}
}
```

### DU.4.4 The Semantics

The equals method for this class returns true if and only if two instances of this class refer to the same object. That is,  $x$  and  $y$  are equal instances of this class if and only if  $x == y$  has the value true.

- ◆ The constructor for the LookupDiscoveryManager takes the following arguments as input:
  - A String array, none of whose elements may be null, in which each element is the name of a group whose members are lookup services the entity wishes to be discovered through group discovery
  - An array of LookupLocator objects, none of whose elements may be null, in which each element corresponds to a specific lookup service the entity wishes to be discovered through locator discovery
  - A reference to an instance of DiscoveryListener that will be notified when a targeted lookup service is discovered, is discarded, or—under certain conditions—has experienced a change in its group membership

The LookupDiscoveryManager will, on behalf of any entity that constructs an instance of this utility, employ the Jini discovery protocols defined in the *Jini™ Discovery and Join Specification* to attempt to find all lookup services

that satisfy the criteria set forth by the contents of the first two arguments, and it will maintain and manage any lookup services that it does discover.

If the constructor is invoked with a set of group names and a set of locators in which either or both sets contain duplicate elements (where duplicate locators are determined by `LookupLocator.equals()`), the invocation is equivalent to constructing this class with no duplicates in either set.

If `null` (`DiscoveryGroupManagement.ALL_GROUPS`) is input to the `groups` argument, then attempts will be made through group discovery to discover all lookup services located within the multicast radius of the entity, regardless of group membership.

Typically, group discovery processing is initiated as soon as an instance of this class is constructed. However, if the empty array (`DiscoveryGroupManagement.NO_GROUPS`) is passed to the `groups` argument of the constructor, no lookup service will be discovered through group discovery until the `addGroups` or `setGroups` method is called to change the managed set of groups to either a non-empty set, or `null` (`DiscoveryGroupManagement.ALL_GROUPS`).

If at least one element of the `groups` argument is `null`, a `NullPointerException` is thrown.

Typically, locator discovery processing is initiated as soon as an instance of this class is constructed. However, if an empty or `null` array is input to the `locators` argument, no attempt will be made to discover specific lookup services through locator discovery until the `addLocators` or `setLocators` method is called to change the managed set of locators to a set of locators that is non-`null` and non-empty.

If at least one element of the `locators` argument is `null`, a `NullPointerException` is thrown.

The last argument to the constructor is a reference to a listener object that will be registered to receive discovery event notifications. If a `null` reference is input to this argument, then the entity will receive no discovery events until `addDiscoveryListener` is invoked with a non-`null` instance of `DiscoveryListener`.

Once a listener is registered with the `LookupDiscoveryManager`, it will be notified of all lookup services discovered through either group or locator discovery, and will be notified whenever those lookup services are discarded.

If a listener registered with the `LookupDiscoveryManager` is also an instance of `DiscoveryChangeListener`, then in addition to receiving events related to discovered and discarded lookup services, that listener will also be

notified of group membership changes that occur in any of the lookup services targeted for at least group discovery. That is, although such listeners are eligible to receive changed events, they will receive no changed events for lookup services for which the entity has requested *only* locator discovery.

Thus, if an entity wishes to receive discovered and discarded events from the *LookupDiscoveryManager*, it is the responsibility of the entity to provide an implementation of either the *DiscoveryListener* or the *DiscoveryChangeListener* interface: an implementation that defines the actions to take upon the receipt of those types of events.

Note that if an entity wishes to receive changed events in addition to the discovered and discarded events it receives from the *LookupDiscoveryManager*, the entity must provide an implementation of *DiscoveryChangeListener* that defines the actions to take upon the receipt of any of the three possible discovery event types. That is, if the entity provides only an implementation of *DiscoveryListener*, the entity will receive no changed events for any of the discovered lookup services, regardless of the discovery mechanism employed for those lookup services.

This constructor throws *IOException*. This is because construction of a *LookupDiscoveryManager* may initiate the multicast discovery process, which can throw *IOException*.

Once a lookup service is discovered, there is no longer any need to perform discovery processing with respect to that lookup service. This means that if a lookup service becomes unreachable after it has been discovered, the *LookupDiscoveryManager* will not know when the lookup service becomes reachable again until that lookup service is discarded.

Although the *LookupDiscoveryManager* will monitor the multicast announcements for indications of unavailability, it will discard only those unreachable lookup services for which the entity requested discovery through at least group discovery. That is, if the *LookupDiscoveryManager* determines that a previously discovered lookup service has become unreachable, but the entity requested that it be discovered by locator discovery alone, then the *LookupDiscoveryManager* will not discard the lookup service.

Thus, whenever the entity itself determines that a previously discovered lookup service has become unreachable, it should not rely on the *LookupDiscoveryManager* to discard the lookup service. Instead, the entity should inform the *LookupDiscoveryManager*—through the invocation of the *discard* method—that the previously discovered lookup service is no longer available, and that attempts should be made to re-discover that lookup service. Typically, an entity determines that a lookup service is unavailable when the entity



attempts to use the lookup service but receives an exception or error (`RemoteException`, for example) as a result of the attempt.

Discovery processing in instances of the `LookupDiscoveryManager` ends either when the instance is finalized prior to garbage collection, or when the `terminate` method is invoked.

## DU.4.5 Supporting Interfaces and Classes

The `LookupDiscoveryManager` helper utility class depends on the interfaces `DiscoveryManagement`, `DiscoveryGroupManagement`, and `DiscoveryLocatorManagement`, and on the concrete class `DiscoveryPermission`.

### DU.4.5.1 The `DiscoveryManagement` Interfaces

The `LookupDiscoveryManager` class implements the `DiscoveryManagement`, the `DiscoveryGroupManagement`, and the `DiscoveryLocatorManagement` interfaces, which together define methods related to the coordination and management of all group and locator discovery processing. See Section DU.1, “The Discovery Management Interfaces” for more information on those interfaces.

### DU.4.5.2 Security and Multicast Discovery: The `DiscoveryPermission` Class

As is the case for the `LookupDiscovery` class, when an instance of the `LookupDiscoveryManager` class is constructed, the entity that creates the instance must be granted appropriate discovery permission to perform the group discovery duties that instance attempts to perform on behalf of the entity. If appropriate permissions are not granted, the constructor of `LookupDiscoveryManager`, as well as the methods `addGroups` and `setGroups`, will throw a `java.lang.SecurityException`.

Discovery permissions are controlled in security policy files using the permission class `DiscoveryPermission`. The specification of that class, as well as useful examples related to that class, are presented in the specification of the `LookupDiscovery` utility (See Section DU.1, “The Discovery Management Interfaces”).



---

## DU.5 Low-Level Discovery Protocol Utilities

**T**HE utilities presented in this section of the specification are useful when implementing higher-level utilities or other entities or components that will be involved in the Jini discovery process. These utilities encapsulate functionality that allow one to exercise more control when interacting with the Jini discovery protocols. Anyone wishing to provide their own implementation of the Jini lookup service or their own implementation of the discovery utilities presented previously in this specification, may find the utilities presented in this section useful when creating those alternate implementations.

### DU.5.1 The Constants Class

#### DU.5.1.1 Overview

The Constants class provides easy access to defined constants that may be useful when participating in the discovery process.

#### DU.5.1.2 Other Types

The types defined in the specification of the Constants class are in the `net.jini.discovery` package. The following additional types may also be referenced in this specification. Whenever referenced, these object types will be referenced in unqualified form:

```
java.net.InetAddress  
java.net.UnknownHostException
```

### DU.5.1.3 The Class Definition

The public constants defined by the Constants class are as follows:

```
package net.jini.discovery;

public class Constants {
    public static final short discoveryPort = 4160;
    public static final InetAddress getRequestAddress()
        throws UnknownHostException {...}
    public static final InetAddress getAnnouncementAddress()
        throws UnknownHostException {...}
}
```

### DU.5.1.4 The Semantics

The Constants class cannot be instantiated. This class has one public variable and two public accessor methods; each is static and final. The constant value associated with the variable, as well as the values returned by the methods, may be useful in the discovery process.

- ◆ The value of the `discoveryPort` constant serves two purposes:
  - The UDP port number over which the multicast request and announcement protocols operate
  - The TCP port number over which the unicast discovery protocol operates by default
- ◆ The `getRequestAddress` method returns an instance of `InetAddress` that contains the address of the multicast group over which the multicast request protocol takes place.
- ◆ The `getAnnouncementAddress` method returns an instance of `InetAddress` that contains the address of the multicast group over which the multicast announcement protocol takes place.

Note that either `getRequestAddress` or `getAnnouncementAddress` may throw an `UnknownHostException` if called in a circumstance under which multicast address resolution is not permitted.

## DU.5.2 The OutgoingMulticastRequest Utility

### DU.5.2.1 Overview

The `OutgoingMulticastRequest` class provides facilities for marshalling multicast discovery requests into a form suitable for transmission over a network for the purposes of announcing one's interest in discovering a lookup service. This class is useful when building components that participate in the multicast request protocol as part of a group discovery mechanism. This utility should be viewed from the perspective of an entity that wishes to transmit multicast requests in order to discover a lookup service belonging to a set of groups in which the entity is interested.

### DU.5.2.2 Other Types

The types defined in the specification of the `OutgoingMulticastRequest` utility class are in the `net.jini.discovery` package. The following additional types may also be referenced in this specification. Whenever referenced, these object types will be referenced in unqualified form:

```
net.jini.core.discovery.ServiceID  
java.io.IOException  
java.net.DatagramPacket  
java.net.InetAddress
```

### DU.5.2.3 The Interface

The public methods provided by the `OutgoingMulticastRequest` class are as follows:

```
package net.jini.discovery;  
  
public class OutgoingMulticastRequest {  
    public static DatagramPacket[] marshal(int port,  
                                           String[] groups,  
                                           ServiceID[] heard)  
        throws IOException {...}  
}
```

#### DU.5.2.4 The Semantics

The `OutgoingMulticastRequest` class cannot be instantiated. This class has only one public method, which is static.

- ◆ The `marshal` method takes as input the following arguments, none of which may be `null`:
  - The port to which respondents should connect in order to start unicast discovery
  - A `String` array, none of whose elements may be `null`, in which each element is the name of a group the requesting entity is interested in discovering
  - An array of `ServiceID` objects, none of whose elements may be `null`, in which each element corresponds to a lookup service the requesting entity has already heard from

Since implementations are not required to check for duplicated elements, the arguments represented as arrays must not contain such elements.

The `marshal` method returns an array whose elements are instances of `DatagramPacket`. The array returned will always contain at least one element, and will contain more if the request is not small enough to fit in a single packet. The array returned by this method is fully initialized; it contains a multicast request as payload and is ready to send over the network.

In the event of error, the `marshal` method may throw an `IOException` if marshalling fails. In some instances the exception thrown may be a more specific subclass of that exception.

### DU.5.3 The IncomingMulticastRequest Utility

#### DU.5.3.1 Overview

The `IncomingMulticastRequest` class provides facilities that are useful when a requesting entity's announced interest in discovering a lookup service is received. The facilities provided by this class encapsulate the details of the process of unmarshalling such received multicast discovery requests into a form in which the individual parameters of the request may be easily accessed. This class is useful when building components that participate in the multicast request protocol as part of a group discovery mechanism, where an entity that uses such a component

wishes to receive multicast requests in order to be discovered through its group membership; for example, an entity such as a lookup service.

### DU.5.3.2 Other Types

The types defined in the specification of the `IncomingMulticastRequest` utility class are in the `net.jini.discovery` package. The following additional types may also be referenced in this specification. Whenever referenced, these object types will be referenced in unqualified form:

```
net.jini.core.discovery.ServiceID
java.io.IOException
java.net.DatagramPacket
java.net.InetAddress
```

### DU.5.3.3 The Interface

The public methods provided by the `IncomingMulticastRequest` class are as follows:

```
package net.jini.discovery;

public class IncomingMulticastRequest {
    public IncomingMulticastRequest(DatagramPacket dgram)
        throws IOException {...}
    public InetAddress getAddress() {...}
    public int getPort() {...}
    public String[] getGroups() {...}
    public ServiceID[] getServiceIDs() {...}
}
```

### DU.5.3.4 The Semantics

Including the constructor, the `IncomingMulticastRequest` class defines five new public methods.

The `equals` method for this class returns `true` if and only if two instances of this class have the same address, port, groups, and service ID values.

- ◆ The constructor of the `IncomingMulticastRequest` class takes a single input parameter: an instance of `DatagramPacket`. The payload of this parameter is assumed to contain nothing but a marshalled discovery request. If the marshalled request contained in the input parameter is corrupt, an `IOException` or a `ClassNotFoundException` will be thrown. In some such instances, a more specific subclass of either exception may be thrown that will give more detailed information.
- ◆ The `getAddress` method returns an instance of `InetAddress` that represents the address of the host to contact in order to start unicast discovery.
- ◆ The `getPort` method returns an `int` value that is the port number to connect to on the remote host in order to start unicast discovery.
- ◆ The `getGroups` method returns an array consisting of the names of the groups in which the requesting entity (the originator of this request) is interested. The array returned by this method may be of zero length, none of its elements will be `null`, and elements in the returned array may or may not be duplicated. Furthermore, the set reflected in the returned array may not be complete, but other incoming packets should contain the rest of the set.
- ◆ The `getServiceIDs` method returns an array of `ServiceID` instances in which each element of the array corresponds to a lookup service from which the requesting entity has already heard. The array returned by this method may be of zero length, none of its elements will be `null`, and elements in the returned array may or may not be duplicated. Furthermore, the set returned by this method may not be complete. That is, there may be more lookup services from which the requesting entity has already heard, but the set returned by this method will not exceed the capacity of a packet.

## **DU.5.4 The OutgoingMulticastAnnouncement Utility**

### **DU.5.4.1 Overview**

The `OutgoingMulticastAnnouncement` class encapsulates the details of the process of marshalling multicast discovery announcements into a form suitable for transmission over a network for the purposes of announcing the availability of a lookup service to interested parties. This class is useful when building components that participate in the multicast announcement protocol as part of a group discovery mechanism. This utility should be viewed from the perspective of an entity that wishes to transmit multicast announcements in order to be discovered



as a lookup service belonging to a set of groups in which other discovering entities may be interested.

#### DU.5.4.2 Other Types

The types defined in the specification of the `OutgoingMulticastAnnouncement` utility class are in the `net.jini.discovery` package. The following additional types may also be referenced in this specification. Whenever referenced, these object types will be referenced in unqualified form:

```
net.jini.core.discovery.LookupLocator  
net.jini.core.discovery.ServiceID  
java.io.IOException  
java.net.DatagramPacket
```

#### DU.5.4.3 The Interface

The public methods provided by the `OutgoingMulticastAnnouncement` class are as follows:

```
package net.jini.discovery;  
  
public class OutgoingMulticastAnnouncement {  
    public static DatagramPacket[] marshal(ServiceID id,  
                                           LookupLocator loc,  
                                           String[] groups)  
        throws IOException {...}  
}
```

#### DU.5.4.4 The Semantics

The `OutgoingMulticastAnnouncement` class cannot be instantiated. This class has only one public method, which is static.

- ◆ The `marshal` method takes as input the following arguments, none of which may be null:
  - ♦ The instance of `ServiceID` that corresponds to the lookup service being advertised
  - ♦ The instance of `LookupLocator` through which the lookup service being advertised may be discovered through unicast discovery

- A non-null `String` array, none of whose elements may be `null`, in which each element is the name of a group in which the lookup service being advertised is a member

The `marshal` method returns an array whose elements are instances of `DatagramPacket`, the contents of which represents a marshalled multicast announcement. The packets created by this method, as represented by the elements of the returned array, are guaranteed to contain all of the groups in which the lookup service being advertised is a member. Note that the set of groups reflected in the returned collection of datagram packets may be distributed among those packets.

Each element of the array returned by this method is initialized such that it is ready for transmission to the appropriate multicast address and UDP port.

In the event of error, the `marshal` method may throw an `IOException` if marshalling fails. In some instances, the exception thrown may be a more specific subclass of that exception.

## **DU.5.5 The IncomingMulticastAnnouncement Utility**

### **DU.5.5.1 Overview**

The `IncomingMulticastAnnouncement` class encapsulates the details of the process of unmarshalling multicast discovery announcements into a form in which the individual parameters of the announcement may be easily accessed. This class is useful when building components that participate in the multicast announcement protocol as part of a group discovery mechanism. This utility should be viewed from the perspective of an entity that wishes to receive multicast announcements in order to discover a lookup service belonging to a set of groups in which the entity is interested.

### **DU.5.5.2 Other Types**

The types defined in the specification of the `IncomingMulticastAnnouncement` utility class are in the `net.jini.discovery` package. The following additional

types may also be referenced in this specification. Whenever referenced, these object types will be referenced in unqualified form:

```
net.jini.core.discovery.LookupLocator
net.jini.core.discovery.ServiceID
java.io.IOException
java.net.DatagramPacket
```

### DU.5.5.3 The Interface

The public methods provided by the `IncomingMulticastAnnouncement` class are as follows:

```
package net.jini.discovery;

public class IncomingMulticastAnnouncement {
    public IncomingMulticastAnnouncement(DatagramPacket p)
        throws IOException {...}
    public ServiceID getServiceID() {...}
    public LookupLocator getLocator() {...}
    public String[] getGroups() {...}
}
```

### DU.5.5.4 The Semantics

Including the constructor, the `IncomingMulticastAnnouncement` class defines four new public methods.

The `equals` method for this class returns `true` if and only if two instances of this class have the same service ID values.

- ◆ The constructor of the `IncomingMulticastAnnouncement` class takes a single input parameter: an instance of `DatagramPacket`. The constructor attempts to unmarshal the input parameter, storing the results in the various fields of this class.

If the contents of the datagram packet cannot be successfully unmarshalled, either an `IOException` or a `ClassNotFoundException` is thrown. In some such instances, a more specific subclass of either exception may be thrown that will give more detailed information.

- ◆ The `getServiceID` method returns the `ServiceID` instance corresponding to the lookup service that sent the announcement.

- ◆ The `getLocator` method returns the `LookupLocator` instance corresponding to the lookup service that sent the announcement. It is through the object returned by this method that the lookup service may be discovered via unicast discovery.
- ◆ The `getGroups` method returns an array consisting of the names of the groups in which the lookup service that sent the announcement is a member. The array returned by this method is never `null`, will contain no `null` elements, or may be empty. Additionally, elements in the returned array may or may not be duplicated.

## **DU.5.6 The OutgoingUnicastRequest Utility**

### **DU.5.6.1 Overview**

The `OutgoingUnicastRequest` class encapsulates the details of the process of marshalling unicast discovery requests into a form suitable for transmission over a network to attempt discovery of a specific lookup service. This class is useful when building components that participate in the unicast request protocol as part of either a group or a locator discovery mechanism. This utility should be viewed from the perspective of an entity that wishes to transmit unicast requests in order to discover a specific lookup service in which the entity is interested.

### **DU.5.6.2 Other Types**

The types defined in the specification of the `OutgoingUnicastRequest` utility class are in the `net.jini.discovery` package. The following additional types may also be referenced in this specification. Whenever referenced, these object types will be referenced in unqualified form:

```
java.io.IOException  
java.io.OutputStream
```

### DU.5.6.3 The Interface

The public methods provided by the `OutgoingUnicastRequest` class are as follows:

```
package net.jini.discovery;

public class OutgoingUnicastRequest {
    public static void marshal(OutputStream str)
        throws IOException {...}
}
```

### DU.5.6.4 The Semantics

The `OutgoingUnicastRequest` class cannot be instantiated. This class has only one public method, which is static.

- ◆ The `marshal` method takes only one parameter as input: an instance of `OutputStream`, which is the stream to which the unicast request is written. After the unicast request is written to the stream, the stream is flushed.

In the event of error, the `marshal` method may throw an `IOException` if writing to the stream fails. In some instances, the exception thrown may be a more specific subclass of that exception.

## DU.5.7 The IncomingUnicastRequest Utility

### DU.5.7.1 Overview

The `IncomingUnicastRequest` class encapsulates the details of the process of unmarshalling unicast discovery requests into a form in which the individual parameters of the request may be easily accessed. This class is useful when building components that participate in the unicast request protocol as part of either a group or a locator discovery mechanism. This utility should be viewed from the perspective of an entity—such as a lookup service—that wishes to receive unicast requests in order to be discovered through direct, unicast communication.

### DU.5.7.2 Other Types

The types defined in the specification of the `IncomingUnicastRequest` utility class are in the `net.jini.discovery` package. The following additional types may also be referenced in this specification. Whenever referenced, these object types will be referenced in unqualified form:

```
java.io.InputStream  
java.io.IOException
```

### DU.5.7.3 The Interface

The public methods provided by the `IncomingUnicastRequest` class are as follows:

```
package net.jini.discovery;  
  
public class IncomingUnicastRequest {  
    public IncomingUnicastRequest(InputStream str)  
        throws IOException {...}  
}
```

### DU.5.7.4 The Semantics

The only new public method defined by the `IncomingUnicastRequest` class is the constructor.

- ◆ The constructor of the `IncomingUnicastRequest` class takes a single input parameter: an instance of `InputStream`, which is the stream from which the unicast request is read.

In the event of error, an `IOException` may be thrown if reading from the stream fails. In some instances, the exception thrown may be a more specific subclass of that exception.

## DU.5.8 The OutgoingUnicastResponse Utility

### DU.5.8.1 Overview

The OutgoingUnicastResponse class encapsulates the details of the process of marshalling a unicast discovery response into a form suitable for transmission over a network to respond to a unicast discovery request. This class is useful when building components that participate in the unicast request protocol as part of either a group or a locator discovery mechanism. This utility should be viewed from the perspective of an entity—such as a lookup service—that wishes to transmit responses to unicast requests in order to be discovered through direct, unicast communication.

### DU.5.8.2 Other Types

The types defined in the specification of the OutgoingUnicastResponse utility class are in the `net.jini.discovery` package. The following additional types may also be referenced in this specification. Whenever referenced, these object types will be referenced in unqualified form:

```
net.jini.core.lookup.ServiceRegistrar  
java.io.IOException  
java.io.OutputStream
```

### DU.5.8.3 The Interface

The public methods provided by the OutgoingUnicastResponse class are as follows:

```
package net.jini.discovery;  
  
public class OutgoingUnicastResponse {  
    public static void marshal(OutputStream s,  
                               ServiceRegistrar reg  
                               String[] groups)  
        throws IOException {...}  
}
```

#### **DU.5.8.4 The Semantics**

The `OutgoingUnicastResponse` class cannot be instantiated. This class has only one public method, which is static.

- ◆ The `marshal` method takes as input the following arguments, none of which may be `null`:
  - An instance of `OutputStream`, which is the stream to which the unicast response is written.
  - An instance of `ServiceRegistrar` that references the proxy to the lookup service that will be marshalled and written to the stream.
  - A non-`null` `String` array, none of whose elements may be `null`, in which each element is the name of a group in which the lookup service referenced by the `reg` parameter is a member. Note that duplicate elements are allowed in this parameter.

The `marshal` method marshals the `reg` parameter and writes the result to the stream. It then writes each element of the `groups` parameter to the stream. After the complete unicast response is written to the stream, the stream is flushed.

This method may throw an `IOException` if a failure occurs while marshalling or writing to the stream. In some instances, the exception thrown may be a more specific subclass of that exception.

### **DU.5.9 The IncomingUnicastResponse Utility**

#### **DU.5.9.1 Overview**

The `IncomingUnicastResponse` class encapsulates the details of the process of unmarshalling a unicast discovery response into a form in which the individual parameters of the request may be easily accessed. This class is useful when building components that participate in the unicast request protocol as part of either a group or a locator discovery mechanism. This utility should be viewed from the perspective of an entity that wishes to receive unicast responses in order to discover lookup services through direct, unicast communication.



### DU.5.9.2 Other Types

The types defined in the specification of the `IncomingUnicastResponse` utility class are in the `net.jini.discovery` package. The following additional types may also be referenced in this specification. Whenever referenced, these object types will be referenced in unqualified form:

```
net.jini.core.lookup.ServiceRegistrar
java.io.InputStream
java.io.IOException
```

### DU.5.9.3 The Interface

The public methods provided by the `IncomingUnicastResponse` class are as follows:

```
package net.jini.discovery;

public class IncomingUnicastResponse {
    public IncomingUnicastResponse(InputStream s)
        throws IOException, ClassNotFoundException {...}
    public ServiceRegistrar getRegistrar() {...}
    public String[] getGroups() {...}
}
```

### DU.5.9.4 The Semantics

Including the constructor, the `IncomingUnicastResponse` class defines three new methods.

The `equals` method for this class returns `true` if and only if two instances of this class reference the same lookup service proxy (registrar).

- ◆ The constructor of the `IncomingUnicastResponse` class takes a single input parameter: an instance of `InputStream`, which is the stream from which the contents of the unicast response is read.

An `IOException` may be thrown if reading from the stream fails. A `ClassNotFoundException` may be thrown if failure occurs while unmarshalling the proxy to the lookup service contained in the unicast response. In some such instances, a more specific subclass of either exception may be thrown that will give more detailed information.

- ◆ The `getRegistrar` method returns an instance of `ServiceRegistrar` that references the proxy to the lookup service sent in the unicast response.
- ◆ The `getGroups` method returns an array consisting of the names of the groups in which the lookup service referenced in the response is a member. The array returned by this method is never `null`, will contain no `null` elements, or may be empty. Additionally, elements in the returned array may or may not be duplicated.

## Jini™ Entry Utilities Specification

### EU.1 Entry Utilities

**E**NTRIES are designed to be used in distributed algorithms for which exact-match lookup semantics are useful. An entry is a typed set of objects, each of which may be tested for exact match with a template. The details of entries and their semantics are discussed in the *Jini Entry Specification*.

When designing entries, certain tasks are commonly done in similar ways. This specification defines a utility class for such common tasks.

#### EU.1.1 AbstractEntry

The class `net.jini.entry.AbstractEntry` is a specific implementation of `Entry` that provides useful implementations of `equals`, `hashCode`, and `toString`:

```
package net.jini.entry;

public abstract class AbstractEntry implements Entry {
    public boolean equals(Object o) {...}
    public int hashCode() {...}
    public String toString() {...}
    public static boolean equals(Entry e1, Entry e2) {...}
    public static int hashCode(Entry entry) {...}
    public static String toString(Entry entry) {...}
}
```

The static method `AbstractEntry.equals` returns `true` if and only if the two entries are of the same class and for each field  $F$ , the two objects' values for  $F$  are either both `null` or the invocation of `equals` on one object's value for  $F$  with the other object's value for  $F$  as its parameter returns `true`. The static method `hashCode` returns zero XOR the `hashCode` invoked on each non-`null` field of the entry. The static method `toString` returns a string that contains each field's name and value. The non-static methods `equals`, `hashCode`, and `toString` return a result equivalent to invoking the corresponding static method with `this` as the first argument.

### EU.1.2 Serialized Form

Class	serialVersionUID	Serialized Fields
<code>AbstractEntry</code>	<code>5071868345060424804L</code>	<i>none</i>

### EU.1.3 Comments

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

# LM

---

## Jini™ Lease Utilities Specification

### LM.1 Introduction

**T**HIS specification defines helper utility classes, along with supporting interfaces and classes, that encapsulate functionality which provides for the coordination, systematic renewal, and overall management of a set of leases associated with some object on behalf of another object. Currently, this specification defines only one helper utility class:

- ◆ The `LeaseRenewalManager` Helper Utility



---

## LM.2 The LeaseRenewalManager

**T**HE LeaseRenewalManager class (belonging to the package `net.jini.lease`) encapsulates functionality that provides for the systematic renewal and overall management of a set of leases associated with one or more remote entities on behalf of a local entity.

The concept of leased resources is fundamental to the Jini technology programming model. Providing a leasing mechanism helps to prevent the accumulation of outdated and unwanted resources in time-based distributed systems, such as the Jini technology infrastructure. The leasing model for Jini connection technology (Jini technology), defined in the *Jini™ Distributed Leasing Specification*, requires renewed proof of interest to continue the existence of a leased resource. Thus, any Jini technology-enabled client (Jini client) or service that requests the use of the leased resources provided by another Jini technology-enabled service (Jini service) may be granted access to those resources for a negotiated period of time; and must continue to request renewal of the lease on each resource for as long as the client or service wishes to have access to the resource.

For example, the Jini lookup service leases two resources: residency in its database, and registration with its event notification mechanism. Thus, if a service that is registered with a Jini lookup service wishes to continue its residency beyond the length of the current lease, the service must request a lease renewal from that lookup service. This renewal process must be repeated for as long as the service wishes to maintain its residency in the lookup service. Similarly, if a client has requested that a lookup service notify it of events of interest, then prior to the expiration of the lease on the event registration, the client must request that the lookup service continue to send such events. As with residency in the lookup service, these renewal requests must be repeated for as long as the client wishes to receive event notifications.

Another example of a Jini service providing leased resources would be a service that implements the *Jini™ Transaction Specification* to manage transactions on behalf of registered participants. That specification requires that a transaction must be a leased resource. Therefore, any entity that creates such a transaction object is required to negotiate (with an entity referred to as a *transaction man-*

ager) a lease on that object, repeatedly requesting lease renewals prior to the lease's expiration, for as long as the transaction is to remain in effect.

The LeaseRenewalManager class is designed to be a simple mechanism that provides for the systematic renewal and overall management of leases granted on resources that are provided by Jini services, for which a Jini client or service has registered interest. The LeaseRenewalManager is a utility class, not a remote service. In order to use this utility, an entity must create, in its own address space, an instance of the LeaseRenewalManager to manage the entity's leases locally.

### LM.2.1 Other Types

The types defined in the specification of the LeaseRenewalManager utility class are in the `net.jini.lease` package. The following types may be referenced in this specification. Whenever referenced, these types will be referenced in unqualified form:

```
net.jini.core.lease.Lease
net.jini.core.lease.UnknownLeaseException
net.jini.core.lease.LeaseDeniedException
java.rmi.RemoteException
java.rmi.NoSuchObjectException
java.util.EventObject
java.util.EventListener
```

### LM.2.2 The Interface

THE public methods provided by the LeaseRenewalManager class are as follows.

```
package net.jini.lease;

public class LeaseRenewalManager
{
    public LeaseRenewalManager() {...}
    public LeaseRenewalManager(Lease lease,
                                long desiredExpiration,
                                LeaseListener listener) {...}
    public void renewUntil(Lease lease,
                           long desiredExpiration,
                           long renewDuration,
```



```

        LeaseListener listener) {...}
    public void renewUntil(Lease lease,
        long desiredExpiration,
        LeaseListener listener) {...}
    public void renewFor(Lease lease,
        long desiredDuration,
        long renewDuration,
        LeaseListener listener) {...}
    public void renewFor(Lease lease,
        long desiredDuration,
        LeaseListener listener) {...}
    public long getExpiration(Lease lease)
        throws UnknownLeaseException {...}
    public void setExpiration(Lease lease,
        long desiredExpiration)
        throws UnknownLeaseException {...}
    public void remove(Lease lease)
        throws UnknownLeaseException {...}
    public void cancel(Lease lease)
        throws UnknownLeaseException, RemoteException {...}
    public void clear() {...}
}

```

### LM.2.3 The Semantics

**T**HE term *client* is used in this specification to refer to the local entity which is using the LeaseRenewalManager to manage a collection of leases on its behalf.

The LeaseRenewalManager distinguishes between two time values associated with lease expiration: the *desired expiration* time for the lease, and the *actual expiration* time granted when the lease is created or last renewed. The desired expiration represents when the client would like the lease to expire. The actual expiration represents when the lease is going to expire if it is not renewed. Both time values are absolute times, not relative time durations. The desired expiration time can be retrieved using the renewal manager's `getExpiration` method, which is described below. The actual expiration time of a lease object can be retrieved by invoking the `getExpiration` method directly on the lease (see the Lease interface defined in the *Jini™ Distributed Leasing Specification*).

Each lease in the managed set also has two other associated attributes: a desired *renewal duration*, and a *remaining desired duration*. The desired renewal duration is specified by the client (directly or indirectly) when the lease is added

to the set. This duration must normally be a positive number; however, it may be `Lease.ANY` if the lease's desired expiration is `Lease.FOREVER`. The remaining desired duration is always the desired expiration less the current time.

Each time a lease is renewed, the renewal manager will ask for an extension equal to the lease's renewal duration if the renewal duration is:

- ◆ `Lease.ANY`, or
- ◆ less than the remaining desired duration,

otherwise it will ask for an extension equal to the lease's remaining desired duration.

The term *definite exception* is used to refer to exceptions that result from operations on a lease (such as a renewal attempt) that are indicative of a permanent failure of the lease. For the purposes of this document, all bad object exceptions (See *Notes on Jini™ technology-enabled clients, services, and utilities*, Section NX.2.6), bad invocation exceptions, and `LeaseExceptions` are considered to be definite exceptions.

The `LeaseRenewalManager` makes certain concurrency guarantees. When the `LeaseRenewalManager` makes a remote call (for example, when requesting the renewal of a lease), any invocations made on the methods of the `LeaseRenewalManager` will not be blocked.

Similarly, the `LeaseRenewalManager` makes a re-entrancy guarantee with respect to `LeaseListener` objects registered with the `LeaseRenewalManager`. Should the `LeaseRenewalManager` invoke a method on a registered listener (a local call), calls from that method to any method of the `LeaseRenewalManager` are guaranteed not to result in a deadlock condition.

The `equals` method for this class returns `true` if and only if two instances of this class refer to the same object. That is, `x` and `y` are equal instances of this class if and only if `x == y` has the value `true`.

The individual semantics of each method of this class are described below.

- ◆ The constructor has two forms:
  - The first form of the constructor takes no arguments. This form of the constructor instantiates a `LeaseRenewalManager` object that initially manages no leases.
  - The second form of the constructor creates a `LeaseRenewalManager` that initially manages a single lease. This form of the constructor requires that a reference to the initial lease be supplied as an argument.

This constructor also takes a `desiredExpiration` argument that represents the desired expiration time for the lease, and a reference to a

`LeaseListener` object that should receive notifications of exceptional conditions that occur when attempting to renew this lease.

Creating a `LeaseRenewalManager` using this form of the constructor is equivalent to invoking the no-argument constructor followed by an invocation of the three-argument form of the `renewUntil` method (described later).

- ◆ The `renewUntil` method adds a lease to the set of leases being managed by the `LeaseRenewalManager`. There are two versions of this method: a four-argument form that allows the client to specify the renewal duration directly, and a three-argument form that infers the renewal duration from the desired expiration argument. The four-argument form will be described first.

This method takes as arguments: a reference to the lease to manage, the desired expiration time of the lease, the renewal duration time for the lease, and a reference to the `LeaseListener` object that will receive notification of exceptional conditions when attempting to renew this lease. The `LeaseListener` argument may be `null`.

If `null` is passed as the lease parameter, a `NullPointerException` will be thrown. If the `desiredExpiration` parameter is `Lease.FOREVER`, the `renewDuration` parameter may be `Lease.ANY` or any positive value; otherwise, the `renewDuration` parameter must be a positive value. If the `renewDuration` parameter does not meet these requirements, an `IllegalArgumentException` will be thrown.

If the lease passed to this method is already in the set of managed leases, the listener object, the desired expiration, and the renewal duration associated with that lease will be replaced with the new listener, desired expiration, and renewal duration.

A lease will remain in the set of managed leases until one of the following occurs:

- The lease's desired or actual expiration time is reached.
- An explicit removal of the lease from the set is requested via a `cancel`, `clear`, or `remove` call on the renewal manager.
- The renewal manager tries to renew the lease and gets a definite exception.

This method will interpret the value of the `desiredExpiration` parameter as the *desired* absolute system time after which the lease is no longer valid. This argument provides the ability to indicate an expiration time that extends

beyond the actual expiration of the lease. If the value passed for this argument does indeed extend beyond the lease's actual expiration time, then the lease will be systematically renewed at appropriate times until one of the conditions listed above occurs. If the value is less than or equal to the actual expiration time, nothing will be done to modify the time when the lease actually expires. That is, the lease will *not* be renewed with an expiration time that is less than the actual expiration time of the lease at the time of the call.

The `desiredRenewal` parameter is interpreted as the renewal duration, in milliseconds, to associate with the lease.

If a non-`null` object reference is passed in as the `LeaseListener` parameter, this object will receive notification of exceptional conditions occurring upon a renewal attempt of the lease. In particular, exceptional conditions include the reception of a definite exception or the lease's actual expiration being reached before its desired expiration.

If a definite exception occurs during a lease renewal request, the exception will be wrapped in an instance of the `LeaseRenewalEvent` class (described later) and sent to the listener.

If an indefinite exception (See *Notes on Jini™ technology-enabled clients, services, and utilities*, Section NX.2.6) occurs during a renewal request for a particular lease, renewal requests will continue to be made for that lease until: the lease is renewed successfully, a renewal attempt results in a definite exception, or the lease's actual expiration time has been exceeded. If the lease cannot be successfully renewed before its actual expiration is reached, the exception associated with the most recent renewal attempt will be wrapped in an instance of the `LeaseRenewalEvent` class and sent to the listener.

If the lease's actual expiration is reached before the lease's desired expiration time, and either 1) the last renewal attempt succeeded or 2) there have been no renewal attempts, a `LeaseRenewalEvent` containing a `null` exception will be sent to the listener. Case 1 can occur if the extension granted by the last renewal was very short. Case 2 can occur if the client adds a lease that has already expired (or is about to) to the managed set of leases.

If `null` is passed in as the `LeaseListener` parameter, no notifications will be generated for exceptional conditions occurring as a result of renewal attempts on the lease.

Calling the three-argument form of `renewUntil` with a `desiredExpiration` of `Lease.ANY` is equivalent to making the following call:

```
renewUntil(lease, Lease.FOREVER, Lease.ANY, listener)
```

Otherwise, the three-argument form is equivalent to:

```
renewUntil(lease, desiredExpiration, Lease.FOREVER,
           listener)
```

---

**Usage Note** – Unless an application has a good reason for doing otherwise, it should use `Lease.ANY` or `Lease.FOREVER` for the renewal duration of a given lease. Using these values gives the grantor of the lease the most flexibility in the length of time for which it grants renewals. In most cases, the grantor of a lease is in a better position than the lease holder to make trade-offs between renewal frequency and the risk of holding on to resources longer than necessary. Specifying a value for the renewal duration of a lease might make sense if the holder of the lease has more information on the value of the leased resource than the grantor, or if the holder needs to ensure that there is an upper bound on how long the lease will remain valid.

---

- ◆ The `renewFor` method adds a lease to the set of leases being managed by the `LeaseRenewalManager`. Like `renewUntil` this method has both a three- and four-argument form. The four-argument form of this method takes as parameters: `lease`, a reference to the lease to manage; `desiredDuration`, a `long` representing the desired duration of lease; `renewDuration`, a `long` representing the renewal duration; and `listener`, a reference to a `LeaseListener` object that will receive notifications of exceptional conditions encountered when attempting to renew lease. Both `desiredDuration` and `renewDuration` are expressed in milliseconds.

The semantics of the four-argument form of `renewFor` are similar to those of the four-argument form of `renewUntil`, with `desiredDuration` + current time being used for the value of the `desiredExpiration` parameter of `renewUntil`. The only exception to this is that, in the context of `renewFor` the value of the `renewDuration` parameter may only be `Lease.ANY` if the value of the `desiredDuration` parameter is *exactly* `Lease.FOREVER`.

This method tests for arithmetic overflow in the desired expiration time computed from the value of `desiredDuration` parameter (`desiredDuration` + current time). Should such overflow be present, a value of `Lease.FOREVER` is used to represent the lease's desired expiration time.

The three-argument form of this method is equivalent to the following call:

```
renewFor(lease, desiredDuration, Lease.FOREVER,  
        listener)
```

Note that for both versions of `renewFor`, `Lease.ANY`, as a value for `desiredDuration`, does not have any special semantics associated with it. Calling either version of `renewFor`, with a `desiredDuration` of `Lease.ANY`, will result in the lease having a desired expiration one millisecond in the past, which will have the effect of causing the lease to be dropped from the managed set of leases immediately without any further notification.

- ◆ The `getExpiration` method returns the current desired expiration time requested for a particular lease, not the actual expiration that was granted when the lease was created or last renewed. The only argument to this method is the reference to the lease object. If the lease is not in the set of managed leases, an `UnknownLeaseException` will be thrown.
- ◆ The `setExpiration` method replaces the current desired expiration of a given lease contained in the set of managed leases with a new desired expiration time. The only arguments to this method are the reference to the lease object and the new expiration time.

An invocation of this method with a lease that is currently a member of the managed set is equivalent to an invocation of the `renewUntil` method with the lease's current listener input to the `listener` parameter. In particular, if the value of the `expiration` parameter is less than or equal to the lease's current actual expiration, this method takes no action.

An invocation of this method with a lease that is not in the set of managed leases will result in an `UnknownLeaseException`.

- ◆ The `remove` method removes a given lease from the set of managed leases. The only argument to this method is the reference to the lease object. If the lease is not in the set of managed leases, an `UnknownLeaseException` will be thrown.

Note that this method does not cancel the given lease; activities such as lease cancellation are left for the client to manage.

- ◆ The `cancel` method both removes a given lease from the set of managed leases and cancels the given lease. The only argument to this method is the reference to the lease object. If the lease is not in the set of managed leases, an `UnknownLeaseException` will be thrown.

Any exception (definite or otherwise) occurring during the cancellation of the lease will have no effect on the removal of the lease from the managed

set. That is, even if an exception occurs during the `cancel` operation, the lease will have been removed from the managed set upon return from this method.

Any exception thrown by the `cancel` method of the lease object itself may also be thrown by this method.

- ◆ The `clear` method removes all leases from the set of managed leases. It does not request the cancellation of those leases. This method takes no arguments.

## LM.2.4 Supporting Interfaces and Classes

THE `LeaseRenewalManager` utility class depends on one interface, `LeaseListener`, which references one class, `LeaseRenewalEvent`.

### LM.2.4.1 The `LeaseListener` Interface

The `LeaseListener` interface defines the mechanism through which the client receives notification that the `LeaseRenewalManager` has failed to renew one of the leases that the `LeaseRenewalManager` is managing. Such renewal failures typically occur because one of the following conditions is met:

- ◆ After successfully renewing a lease any number of times and experiencing no failures, the `LeaseRenewalManager` determines — prior to the next renewal attempt — that the actual expiration time of the lease has passed; implying that any further attempt to renew the lease would be fruitless.
- ◆ An indefinite exception occurs during each attempt to renew a lease from the point that the first such exception occurs until the point when the `LeaseRenewalManager` determines that lease's actual expiration time has passed.
- ◆ A definite exception occurs during a lease renewal attempt.

It is the responsibility of the client to pass into the `LeaseRenewalManager` a reference to an object that implements the `LeaseListener` interface, which defines the actions to take upon receipt of a notification. When one of the above conditions occurs, the `LeaseRenewalManager` will send an instance of `LeaseRenewalEvent` to that listener object.

```
package net.jini.lease;
```

```
public interface LeaseListener extends EventListener
```

```
{  
    void notify(LeaseRenewalEvent e);  
}
```

### The Semantics

The `notify` method is invoked by the `LeaseRenewalManager` when it fails to renew a lease because one of the conditions described above has occurred. This method takes one parameter: an instance of the `LeaseRenewalEvent` class, which contains information about the lease on which the failed renewal attempt was made and information on what caused the failure.

Note that prior to invoking this method, the `LeaseRenewalManager` removes the affected lease from the managed set of leases. Note also that because of the re-entrancy guarantee made by the `LeaseRenewalManager`, new leases can be safely added from within any listeners that implement the `LeaseListener` interface.

#### LM.2.4.2 The LeaseRenewalEvent Class

This class defines the event that is sent by the `LeaseRenewalManager` to the client's registered listener when the `LeaseRenewalManager` fails to renew one of the leases it is managing. As previously stated, such failures typically occur because the actual expiration time of the lease has been reached before a successful renewal request can be made, or a renewal request resulted in a definite exception. The `LeaseRenewalEvent` class encapsulates information about the lease on which the failure occurred and information on what caused the failure.

```
package net.jini.lease;  
  
public class LeaseRenewalEvent extends EventObject  
{  
    public LeaseRenewalEvent(LeaseRenewalManager source,  
                             Lease lease,  
                             long expiration,  
                             Throwable ex){...}  
    public Lease getLease(){...}  
    public long getExpiration(){...}  
    public Throwable getException(){...}  
}
```

The `LeaseRenewalEvent` class is a subclass of the class `EventObject`, adding the following additional items of abstract state: a reference to the affected



Lease object; a long value representing the desired expiration of the affected lease; and the exception (if any) that caused the event to be sent. In addition to the methods of the `EventObject` class, this class defines methods through which this additional state may be retrieved.

## The Semantics

The constructor of the `LeaseRenewalEvent` class takes the following parameters as input:

- ◆ A reference to the instance of the `LeaseRenewalManager` that generated the event
- ◆ The lease on which the renewal attempt failed, and to which the event pertains
- ◆ The desired expiration time of the affected lease
- ◆ The `Throwable` associated with the last renewal attempt (if any)

The `getLease` method returns a reference to the `Lease` object associated with the event. This method takes no arguments.

The `getExpiration` method returns a long value representing the desired expiration of the `Lease` object associated with the event. This method takes no arguments.

The `getException` method returns the exception that caused the event to be sent. This method takes no arguments. The conditions under which a `LeaseRenewalEvent` may be sent, and the related values returned by this method, are as follows:

- ◆ When any lease in the managed set has passed its actual expiration time, and either the most recent renewal attempt was successful or there have been no renewal attempts, the `LeaseRenewalManager` will cease any further attempts to renew the lease, and will send a `LeaseRenewalEvent` with no associated exception. In this case, invoking this method will return `null`.
- ◆ For any lease from the managed set for which the most recent renewal attempt was unsuccessful because of the occurrence of an indefinite exception, the `LeaseRenewalManager` will continue to attempt to renew the affected lease at the appropriate times until: the renewal succeeds, the lease's expiration time has passed, or a renewal attempt throws a definite exception. If a definite exception is thrown or the lease expires, the `LeaseRenewalManager` will cease any further attempts to renew the lease, and will send a

`LeaseRenewalEvent` containing the exception associated with the last renewal attempt.

- ◆ If, while attempting to renew a lease from the managed set, a definite exception is encountered, the `LeaseRenewalManager` will cease any further attempts to renew the lease, and will send a `LeaseRenewalEvent` containing the particular exception that occurred.

# JU

---

## Jini™ Join Utilities Specification

### **JU.1 Introduction**

**T**HIS specification defines helper utility classes, along with supporting interfaces and classes, that encapsulate functionality that can help Jini services demonstrate good behavior in their discovery and registration related interactions with Jini lookup services. In particular, the Jini join utilities perform functions related to lookup service discovery and registration (joining), as well as lease renewal and attribute management, which the Jini technology programming model requires of a well-behaved Jini technology-enabled service. Currently, this specification defines only one helper utility class:

- ◆ The JoinManager Helper Utility



---

## JU.2 The JoinManager

**T**HE goal of any well-behaved Jini™ technology-enabled service (Jini service), implemented within the bounds defined by the Jini technology programming model, is to advertise the service it provides by requesting residency within at least one Jini lookup service. Making such a request of a Jini lookup service is known as registering with, or *joining*, a lookup service. To demonstrate this good behavior, a service must comply with both the *multicast discovery protocol* and the *unicast discovery protocol* in order to discover the lookup services it is interested in joining. The service must also comply with the *join protocol* to register with the desired lookup services. The details of the discovery and join protocols are described in the *Jini™ Discovery and Join Specification*.

In order for the service to maintain its residency in the lookup services it has joined, the service must provide for the coordination, systematic renewal, and overall management of all leases on that residency. In addition to handling all discovery and join duties, as well as managing all leases on lookup residency, the service must provide for the coordination and management of any attribute sets with which it may have registered.

With respect to the duties described above, a Jini service may perform all but the attribute set management duties through the use of the `LookupDiscoveryManager` and `LeaseRenewalManager` helper utility classes (for information on these classes, refer to the *Jini™ Discovery Utilities Specification* and *Jini™ Lease Renewal Service Specification*).

Rather than writing a service to use these classes in a coordinated fashion (in addition to providing for attribute management), the service may be written to employ the `JoinManager` class from the `net.jini.lookup` package. This utility class performs all of the functions related to discovery, joining, service lease renewal, and attribute management which the Jini technology programming model requires of a well-behaved Jini service. Each of these activities is intimately involved with the maintenance of a service's residency in one or more lookup services (the service's *join state*), thus the name `JoinManager`.

The `JoinManager` class provides an implementation of the functionality described above. The use of this class in a wide variety of services can help mini-

mize the work resulting from having to repeatedly implement this required functionality in each service.

The JoinManager is a utility class, not a remote service. Jini services that wish to use this utility will create an instance of the JoinManager in the service's address space to manage the entity's join state locally.

Note that when the term *service* is used, it refers to the object that has created an instance of the JoinManager and avails itself of the public methods of that utility class.

## JU.2.1 Other Types

The types defined in the specification of the JoinManager utility class are in the `net.jini.lookup` package. The following types may be referenced in this chapter. Whenever referenced, these object types will be referenced in unqualified form:

```
net.jini.core.lease.Lease
net.jini.core.entry.Entry
net.jini.core.lookup.ServiceID
net.jini.core.lookup.ServiceRegistrar
net.jini.core.lookup.ServiceRegistration
net.jini.discovery.DiscoveryListener
net.jini.discovery.DiscoveryManagement
net.jini.lookup.entry.ServiceControlled
net.jini.lease.LeaseRenewalManager
net.jini.discovery.LookupLocatorDiscovery
net.jini.discovery.LookupDiscoveryManager
java.io.IOException
java.rmi.MarshalledObject
java.util.EventListener
```

---

## JU.3 The Interface

**T**HE public methods provided by the JoinManager class are as follows:

```
package net.jini.lookup;

public class JoinManager
{
    public JoinManager(Object obj,
                        Entry[] attrSets,
                        ServiceIDListener callback,
                        DiscoveryManagement discoveryMgr,
                        LeaseRenewalManager leaseMgr)
                        throws IOException{...}

    public JoinManager(Object obj,
                        Entry[] attrSets,
                        ServiceID serviceID,
                        DiscoveryManagement discoveryMgr,
                        LeaseRenewalManager leaseMgr)
                        throws IOException{...}

    public DiscoveryManagement getDiscoveryManager(){...}
    public LeaseRenewalManager getLeaseRenewalManager(){...}
    public ServiceRegistrar[] getJoinSet(){...}

    public Entry[] getAttributes(){...}
    public void addAttributes(Entry[] attrSets){...}
    public void addAttributes(Entry[] attrSets,
                              boolean checkSC){...}
    public void setAttributes(Entry[] attrSets){...}
    public void modifyAttributes(Entry[] attrSetTemplates,
                                Entry[] attrSets){...}
    public void modifyAttributes(Entry[] attrSetTemplates,
                                Entry[] attrSets,
```

```
        boolean checkSC){...}  
    public void terminate(){...}  
}
```



---

## JU.4 The Semantics

**T**HE `equals` method for this class returns `true` if and only if two instances of this class refer to the same object. That is, `x` and `y` are equal instances of this class if and only if `x == y` has the value `true`.

The individual semantics of each method of this class are described below.

- ◆ The constructor of the `JoinManager` class has two forms. Each form of the constructor throws `IOException` because construction of a `JoinManager` may initiate the multicast discovery process, which can throw `IOException`.

The first form of the constructor takes the following parameters as input:

- A reference to the service requesting the services of the `JoinManager`
- An array containing the service's attributes
- A reference to an object that implements the `ServiceIDListener` interface (belonging to the package `net.jini.lookup`)
- A reference to an object that implements the `DiscoveryManagement` interface
- An instance of the `LeaseRenewalManager` utility class

Passing `null` as the value of the `attrSets` parameter is equivalent to passing an empty `Entry` array.

The assignment of a service ID to the service will result in an event notification being sent to the listener object that was passed as the `ServiceIDListener` argument (`callback`). If a `null` value is passed in through this argument, then no such notification will be sent.

In order to use the `JoinManager`, the service supplies an object through which notifications that indicate a lookup service has been discovered or discarded will be received. At a minimum, this object must satisfy the contract defined in the `DiscoveryManagement` interface. That is, this object must provide the `JoinManager` with the ability to set discovery listeners and to discard previously discovered lookup services when they are found to be unavailable.

The `DiscoveryManagement` argument may be set to a value of `null`. If `null` is the value of this argument, then an instance of the `LookupDiscoveryManager` utility class will be constructed to listen for events announcing the discovery of only those lookup services that are members of the public group.

The `LeaseRenewalManager` argument may be set to a value of `null`. If `null` is the value of this argument, an instance of the `LeaseRenewalManager` class will be created, initially managing no Lease objects. This feature allows a service that employs the `JoinManager` to use either a single entity to manage all of its leases, or to use separate entities: one to manage the leases unrelated to the join process, and one to manage the leases that result from the join process and which are accessible only within the `JoinManager`.

The first form of the constructor is typically used by services that have not yet been assigned a service ID, but which have been pre-configured to join lookup services that the service identifies through the initialization of a discovery manager.

The second form of the constructor takes the same arguments as the first, except that an instance of the `ServiceID` replaces an instance of the `ServiceIDListener` interface. Note that the `ServiceID` class is defined in the *Jini™ Lookup Service Specification*, and the `ServiceIDListener` interface is described later in this document.

The second form of the constructor applies the same semantics to the `attrSets`, `discoveryMgr`, and `leaseMgr` arguments as is applied by the first form of the constructor.

The second form of the constructor should be used by services that have already been assigned a service ID (possibly by the service provider or as a result of a prior registration with some lookup service), and which may or may not have been pre-configured to join lookup services identified by group or by specific location.

- ◆ The `getDiscoveryManager` method returns the instance of `DiscoveryManagement` that was either passed into the constructor by the entity, or that was created as a result of `null` being input to that parameter. This method takes no arguments as input.

The object returned by this method encapsulates the mechanism by which either the `JoinManager` or the entity itself can set discovery listeners and discard previously discovered lookup services when they are found to be unavailable.

- ◆ The `getLeaseRenewalManager` method returns an instance of the `LeaseRenewalManager` class. This method takes no arguments as input.

The object returned by this method manages the leases requested and held by the `JoinManager`. Although it may also manage leases unrelated to the join process that are requested and held by the service itself, the leases with which the `JoinManager` is concerned are the leases that correspond to the service registration requests the `JoinManager` has made with each lookup service the service wishes to join.

- ◆ The `getJoinSet` method returns an array of `ServiceRegistrar` objects, each corresponding to a lookup service with which the service is currently registered (joined). If there are no lookup services with which the service is currently registered, this method returns the empty array.

This method takes no arguments as input, and will return a new array upon each invocation.

- ◆ The `getAttributes` method returns an array containing the set of attributes currently associated with the service. If the service is not currently associated with an attribute set, this method returns the empty array.

This method takes no arguments as input, and will return a new array upon each invocation.

- ◆ The `addAttributes` method associates a new set of attributes with the service, in addition to the service's current set of attributes. The association of this new set of attributes with the service will be propagated to each lookup service with which the service is registered. This propagation must be performed asynchronously, so there is no guarantee that the propagation of the attributes to all lookup services with which the service is registered will have completed upon return from this method.

The set of attributes consisting of the union of the new set with the old set will be associated with the service in all future join processing.

There are two forms of the `addAttributes` method. Both forms of this method take as input an argument (`attrSets`) representing the set of attributes to associate with the service. This set is represented as an array of `Entry` objects, none of whose elements may be `null`. If at least one element of this input set is `null`, a `NullPointerException` is thrown.

An invocation of either form of this method with duplicate elements in the `attrSets` parameter (where duplication means attribute equality as defined by calling the `MarshaledObject.equals` method on field values) is equiv-

alent to performing the invocation with the duplicates removed from that parameter. If `null` is passed in as the value of this parameter, a `NullPointerException` will be thrown.

The second form of this method also takes as input a flag indicating whether or not this method should determine if the attributes in the input set are instances of the `ServiceControlled` interface, which is a marker interface that is used to control which entities may modify a service's attribute set. For more information on this interface, refer to section 4.1 of the *Jini™ Lookup Attribute Schema Specification*. If the value of this flag is `true` and at least one of the attributes to be added is an instance of the `ServiceControlled` interface, a `SecurityException` will be thrown and propagated through this method.

Note that because there is no guarantee that attribute propagation will have completed upon return from this method, services that invoke this method must take care not to modify the contents of the input array. Doing so could cause the service's attribute state to be corrupted or inconsistent on a subset of the lookup services with which the service is registered as compared with the state reflected on the remaining lookup services. It is for this reason that the effects of modifying the contents of the input array, after this method is invoked, are undefined.

- ◆ The `setAttributes` method replaces the service's current set of attributes with the given new set of attributes. This method takes a single argument as input: an array of `Entry` objects, none of whose elements may be `null`, which represents the set of attributes that will replace the current set of attributes. If at least one element of this input set is `null`, a `NullPointerException` is thrown.

The replacement of the service's current set of attributes with the new set of attributes will be propagated to each lookup service with which the service is registered. This propagation must be performed asynchronously, so there is no guarantee that the propagation of the attributes to all lookup services with which the service is registered will have completed upon return from this method.

The service's new set of attributes will be associated with the service in all future join processing.

An invocation of this method with duplicate elements in the `attrSets` parameter (where duplication means attribute equality as defined by calling the `MarshaledObject.equals` method on field values) is equivalent to performing the invocation with the duplicates removed from that parameter.

If null is input to `setAttributes`, a `NullPointerException` will be thrown.

For the same reason as noted above in the description of the `addAttributes` method, the effects of modifying the contents of the input array, after `setAttributes` is invoked, are undefined.

- ◆ The `modifyAttributes` method changes the service's current set of attributes using the same semantics as the `modifyAttributes` method of the `ServiceRegistration` class (see the *Jini™ Lookup Service Specification*). This method has two forms. The first form takes two arguments, the second form takes three arguments. Both forms will take an array of templates in the first argument, and an array of attributes in the second argument. The templates are used to identify which elements to modify from the service's current set of attributes. The attribute array contains the actual modifications to be made. The additional argument in the signature of the second form of `modifyAttributes` is a flag indicating whether or not this method should determine if the attributes in the input set are instances of the `ServiceControlled` interface, which is a marker interface used to control which entities may modify a service's attribute set (see section 4.1 of the *Jini™ Lookup Attribute Schema Specification*). If the value of this flag is true and at least one of the attributes to be modified is an instance of the `ServiceControlled` interface, a `SecurityException` will be thrown and propagated through this method.

The association of the new set of attributes with the service will be propagated to each lookup service with which the service is registered. This propagation must be performed asynchronously. Because of this asynchronous behavior, there is no guarantee that the propagation of the attributes to all lookup services with which the service is registered will have completed upon return from this method.

The set of attributes that results after the modifications have been applied will be associated with the service in all future join processing.

If the length of the array containing the templates does not equal the length of the array containing the attributes, an `IllegalArgumentException` will be thrown and propagated through this method.

For the same reason as that noted above in the description of the `addAttributes` method, the effects of modifying the contents of the `attr-Sets` parameter, after `modifyAttributes` is invoked, are undefined.

- ◆ The `terminate` method performs cleanup duties related to the termination of the *lookup service* discovery event mechanism, as well as to the lease and thread management performed by the `JoinManager`.

This method will cancel all of the service's managed leases that were granted by the lookup services with which the service is registered, and will terminate all threads that have been created.

If the discovery manager employed by the `JoinManager` was created by the `JoinManager` itself, this method will terminate *all* discovery processing being performed by that manager object on behalf of the service; otherwise, the discovery manager supplied by the service is still valid.

Whether an instance of the `LeaseRenewalManager` class was supplied by the service or created by the `JoinManager` itself, any reference to that object obtained by the service prior to termination will still be valid after termination.

The `JoinManager` makes certain concurrency guarantees with respect to an invocation of the `terminate` method while other method invocations are in progress. The termination process described above will not begin until completion of all invocations of the methods defined in the public interface of the `JoinManager`. Furthermore, once the termination process has begun, no remote method invocations will be made by the `JoinManager`, and all other method invocations on the `JoinManager` will not return until the termination process has completed. Upon completion of the termination process, the semantics of all current and future method invocations on the current instance of the `JoinManager` are undefined, although the reference to the `LeaseRenewalManager` object employed by the `JoinManager` is still valid.

---

## JU.5 Supporting Interfaces and Classes

**T**HE `JoinManager` utility class depends on the following interfaces: `DiscoveryManagement` and `ServiceIDListener` discussed below.

This class also references the following concrete classes: `LookupDiscoveryManager` and `LeaseRenewalManager`, each described in a separate chapter of this document.

### JU.5.1 The `DiscoveryManagement` Interface

Although it is not necessary for the `JoinManager` itself to execute the discovery process, it does need to be notified when one of the lookup services it wishes to join is discovered or discarded. Thus, at a minimum, the `JoinManager` requires access to the discovery events sent to the listeners registered with the discovery process' event mechanism. The instance of `DiscoveryManagement` that is passed as an argument to the constructor of the `JoinManager` provides a mechanism for acquiring access to those events. For a complete description of the semantics of the methods of this interface, refer to the *Jini™ Discovery Utilities Specification*.

One noteworthy item about the semantics of the `JoinManager` is the effect that invocations of the `discard` method of `DiscoveryManagement` will have on any discovery listeners created by the `JoinManager`. The `DiscoveryManagement` interface specifies that the `discard` method will remove a particular lookup service from the managed set of lookup services that have already been discovered, allowing that lookup service to be re-discovered. Invoking this method will result in the flushing of the lookup service from the appropriate cache, ultimately causing a discard notification to be sent to all `DiscoveryListener` objects registered with the event mechanism of the discovery process, including all listeners registered by the `JoinManager`.

The receipt of an event notification indicating that a lookup service has been discarded ultimately results in the removal (but not cancellation) of the registration lease that was granted by the discarded lookup service, and which is managed by the `LeaseRenewalManager` on behalf of the `JoinManager`. After removal occurs, the lease will eventually expire.

## JU.5.2 The ServiceIDListener Interface

The ServiceIDListener interface defines the methods used by a service to register a request for notification from the JoinManager upon the assignment of a serviceID by a lookup service. It is the responsibility of the service to create and pass into the JoinManager an object that implements this interface. That implementation must provide the definition of the actions to take upon receipt of the notification. Typically, the action taken will be to persist the assigned serviceID reference.

```
package net.jini.lookup;

public interface ServiceIDListener extends EventListener
{
    public void serviceIDNotify(ServiceID serviceID);
}
```



# SD

---

## Jini™ Service Discovery Utilities Specification

### **SD.1 Introduction**

**T**HIS specification defines helper utility classes, along with supporting interfaces and classes, that encapsulate functionality that can help a Jini service or client in acquiring services of interest that are registered with the various lookup services with which the service or client wishes to interact. Currently, the service discovery utilities specification defines only one helper utility class:

- ◆ The `ServiceDiscoveryManager` Helper Utility



---

## SD.2 The ServiceDiscoveryManager

THE interactions of an entity that operates in a client-like fashion within a Jini™ application environment are generally distinguished by the fact that the entity first *discovers* one or more Jini lookup services, then *queries* one or more of the discovered lookup services for references to Jini technology-enabled services (*Jini services*) that the entity may employ in some task. This process, in which Jini services as well as Jini technology-enabled clients (*Jini clients*) may participate, is often referred to as *service discovery*. Since services and clients can perform both *lookup* discovery and *service* discovery, the primary characteristic that distinguishes a Jini service from a client is the service's ability to be registered with a lookup service. Thus, with respect to service discovery, there is no difference between a Jini service and a Jini client.

Because there is no need to make such a distinction, throughout this chapter one of two terms will be used interchangeably when referring to Jini clients or services that create an instance of the ServiceDiscoveryManager (from the package `net.jini.lookup`) and avail themselves of the public methods of that class to perform and manage their service discovery duties. Those terms are: *entity* and *client-like entity*.

Once a client-like entity discovers a set of lookup services and retrieves references to desired services from those lookup services, the entity may choose to discontinue query-related discovery processing. That is, having obtained references to all of the services it wishes to employ, the entity may view the references it holds to the lookup services as no longer necessary.

But over the execution life of any such entity, partial failures such as system crashes or network outages may intermittently affect the availability of some of those services of interest. This results in a need to re-query the lookup services to find references to new instances of the service that can replace the unavailable instance. Such scenarios make it desirable for a client-like entity to maintain its references to the lookup services it queries. If an instance of a service is found to be unavailable, the entity can query those lookup services to obtain an instance of the service which is available.

Since a query on a lookup service is a remote call, such calls are much more costly in terms of overhead and failure risk than are local calls. This cost is magni-

fied when an entity must make frequent queries for multiple services, so an entity may find it desirable to cache the services it obtains from the original queries on the lookup services. Furthermore, by populating the cache with multiple instances of the desired services, redundancy in the availability of those services can be provided. Thus, if an instance of a service is found to be unavailable when needed, the entity can execute a local query on the cache rather than one or more remote queries on the lookup services to obtain an instance which is available.

Typically, an entity will request the creation of a separate cache for each service type of interest. The cache provides a method with which the entity can retrieve an element of the cache. In general, the particular service reference that is returned should not matter to the entity. It should only matter that *a* service reference has been returned, not *which* service reference. If for some reason it does matter to an entity which service reference is returned, then the cache also provides a mechanism which will allow the entity to retrieve all elements of the cache. The entity can then iterate through each element, selecting the particular reference it desires.

Although interacting with a local cache of services in this way can be very useful to entities that need frequent access to multiple services, some client-like entities may wish to interact with the cache in a reactive manner. For example, an entity such as a service browser typically wishes to be *notified* of the arrival of new services of interest as well as any changes in the state of the current services in the cache. Polling for such changes is usually viewed as undesirable. If the cache were to also provide an event mechanism with notification semantics, the needs of both types of entity could be satisfied.

From the scenarios discussed above, one could conclude that when acting in a client-like fashion, it is desirable for an entity to maintain, as much as possible, up-to-date knowledge of the availability of the *lookup* services of interest as well as the state information associated with all other types of services in which the entity is interested. By maintaining current service state information, the entity can implement efficient mechanisms for service access and usage.

The *ServiceDiscoveryManager* class is a helper utility class that any entity can use to create and populate a cache such as that described previously, and with which the entity can register for notification of the availability of services of interest. Like the *JoinManager* utility class, this class needs to be notified when a desired lookup service is discovered. For information on the *JoinManager* utility class, refer to the *Jini™ Join Utilities Specification*.

Unlike the *JoinManager*, the *ServiceDiscoveryManager* does not register the entity as a service with discovered lookup services. Although both the *JoinManager* and the *ServiceDiscoveryManager* perform lookup discovery event handling for the entities that employ them, the *JoinManager* performs *join* processing for Jini services, while the *ServiceDiscoveryManager* performs *ser-*

*vice discovery and management* processing both for clients and for services. Thus, typical usage patterns for Jini services wishing to find and use other Jini services generally indicate the employment of both the `JoinManager` and the `ServiceDiscoveryManager` utilities, whereas Jini clients would typically use only the `ServiceDiscoveryManager`.

The `ServiceDiscoveryManager` class can be asked to “discover” services an entity is interested in using, and to cache the references to those services as each is found. The cache can be viewed as a set of service references that the entity can access locally as needed through one of the public, non-remote methods provided in the cache’s interface. A service reference added to the cache will be removed from the cache when all of the lookup services with which that service is registered have been discarded.

The `ServiceDiscoveryManager` class also provides a mechanism for an entity to request that it be notified when a service of interest is discovered for the first time or has encountered a state change such as removal from all lookup services or attribute set changes.

For convenience, this class also provides versions of a method named `lookup`, which employs invocation semantics similar to the semantics of the `lookup` method of the `ServiceRegistrar` interface defined in the *Jini™ Lookup Service Specification*. This method may be useful to entities needing to find services on an infrequent basis, or when the cost of making a remote call is outweighed by the overhead of maintaining a local cache (for example, due to limited resources).

All three mechanisms described above—local queries on the cache, service discovery notification, and remote lookups—employ the same template-matching scheme as that described in the *Jini™ Lookup Service Specification*. Additionally, each mechanism allows the entity to supply an object referred to as a *filter*. Such an object is a non-remote object that defines additional matching criteria that the `ServiceDiscoveryManager` applies when searching for the entity’s services of interest. This filtering facility is particularly useful to entities that wish to extend the capabilities of the standard template-matching scheme.

The `ServiceDiscoveryManager` is a utility class, not a remote service. Client-like entities that wish to use this utility will create an instance of the `ServiceDiscoveryManager` in the entity’s address space so as to manage the entity’s “lookup state” locally.

## SD.2.1 The Object Types

The types defined in the specification of the `ServiceDiscoveryManager` utility class are in the `net.jini.lookup` package. The following types may be refer-

enced in this chapter. Whenever referenced, these object types will be referenced in unqualified form:

```
net.jini.core.discovery.LookupLocator
net.jini.core.lease.Lease
net.jini.core.lookup.ServiceItem
net.jini.core.lookup.ServiceMatches
net.jini.core.lookup.ServiceRegistrar
net.jini.core.lookup.ServiceTemplate
net.jini.discovery.DiscoveryListener
net.jini.discovery.DiscoveryManagement
net.jini.discovery.LookupDiscoveryManager
net.jini.lease.LeaseRenewalManager
net.jini.lookup.LookupCache
net.jini.lookup.ServiceDiscoveryEvent
net.jini.lookup.ServiceDiscoveryListener
net.jini.lookup.ServiceItemFilter
java.io.IOException
java.rmi.server.UnicastRemoteObject
java.rmi.MarshalledObject
java.rmi.RemoteException
java.util.EventListener
java.util.EventObject
java.util.Set
```

---

## SD.3 The Interface

**T**HE public interface provided by the `ServiceDiscoveryManager` class defines methods that allow an entity to request that references to services matching criteria defined by the entity be found in discovered lookup services and cached for local retrieval. This interface also defines methods for retrieving the manager objects employed by this utility, and for performing termination processing.

```
package net.jini.lookup;

public class ServiceDiscoveryManager
{
    public ServiceDiscoveryManager
        (DiscoveryManagement discoveryMgr,
         LeaseRenewalManager leaseMgr)
        throws IOException{...}
    public LookupCache createLookupCache
        (ServiceTemplate tmpl,
         ServiceItemFilter filter,
         ServiceDiscoveryListener listener)
        throws RemoteException{...}
    public ServiceItem lookup(ServiceTemplate tmpl,
                             ServiceItemFilter filter){...}
    public ServiceItem lookup(ServiceTemplate tmpl,
                             ServiceItemFilter filter,
                             long waitDur)
        throws InterruptedException,
               RemoteException{...}
    public ServiceItem[] lookup(ServiceTemplate tmpl,
                                int maxMatches,
                                ServiceItemFilter filter){...}
    public ServiceItem[] lookup(ServiceTemplate tmpl,
                                int minMatches,
                                int maxMatches,
```

```
        ServiceItemFilter filter,  
        long waitDur)  
        throws InterruptedException,  
            RemoteException{...}  
    public DiscoveryManagement getDiscoveryManager(){...}  
    public LeaseRenewalManager getLeaseRenewalManager(){...}  
    public void terminate(){...}  
}
```



---

## SD.4 The Semantics

**T**HE `ServiceDiscoveryManager` makes certain concurrency guarantees with respect to the methods it defines. When a method of the `ServiceDiscoveryManager` invokes a remote method, although such an invocation may block other remote calls made in the `ServiceDiscoveryManager`, invocations of local methods will not be blocked.

The `equals` method for this class returns `true` if and only if two instances of this class refer to the same object. That is, `x` and `y` are equal instances of this class if and only if `x == y` has the value `true`.

### SD.4.1 Exporting `RemoteEventListener` Objects

A subset of the methods on the `ServiceDiscoveryManager`, when invoked, will result in a request for registration with the event mechanism of one or more lookup services. The methods that result in such a request are `createLookupCache` and the so-called *blocking* versions of the lookup method, each described in this section.

Any entity that invokes one of these methods must export to each lookup service with which a registration occurs, the stub classes of the `RemoteEventListener` object through which instances of `RemoteEvent` will be received. Furthermore, each of these methods must throw `RemoteException`. The reasons that a `RemoteException` can occur fall into one of the following categories:

- ◆ Each of these methods attempts to export a remote object, a process that can throw `RemoteException`.
- ◆ Each of these methods attempts to register with the event mechanism of at least one lookup service, a process that can throw `RemoteException`.

How this set of `ServiceDiscoveryManager` methods handles the `RemoteException` is dependent on the reason for the exception. If a

`RemoteException` (or any other non-fatal exception or error) is thrown during an attempt to register for events from a lookup service, that lookup service will be discarded and made eligible for re-discovery. On the other hand, if the `RemoteException` is thrown during an attempt to export the listener, the method will exit and the exception will be propagated.

The potential for `RemoteException` during the export process imposes the following requirement: the *same* instance of the listener must be exported to each lookup service from which events will be requested. Furthermore, the creation and export of the listener must occur prior to the event registration process. This requirement guarantees that should a `RemoteException` occur after the registration process has begun, the exception will not be propagated and event processing will continue.

To understand the significance of this requirement, consider the scenario where a different instance of the listener is exported to each lookup service. If a new lookup service is discovered after the event process has begun for the other lookup services in the managed set, a new instance of the listener must be created and exported. Should a `RemoteException` occur during the export process, the exception will be propagated and all event processing will stop—a result that many entities may view as undesirable.

In order to facilitate exporting the listener, the entity—whether it is a Jini client or a Jini service—is responsible for providing and advertising a mechanism through which each lookup service will acquire the listener’s stub classes.

For example, one implementation of the `ServiceDiscoveryManager` might provide a special JAR file containing only the listener stub classes (to optimize download time). By including this JAR file in the entity’s `java.rmi.server.codebase` property (in the appropriate format specifying transport protocol and location), the entity *advertises* the mechanism that lookup services can employ to acquire the stub classes. By executing a process to serve up the JAR file (for example, an HTTP server), the mechanism through which each lookup service acquires those stub classes is *provided*.

It is important to note that should such a mechanism not be made available to each lookup service with which event registration will be requested, a “silent failure” can occur repeatedly. If the mechanism is not available, each lookup service cannot acquire the exported listener. Because each lookup service cannot acquire the exported listener, any attempts to register for events will fail. Whenever an attempt to register for events fails, the associated lookup service will be discarded and made eligible for re-discovery. Upon re-discovery of the discarded lookup service, the cycle repeats when a new attempt to register for events is made.

The individual semantics of each method of this class are described here:

- ◆ The constructor of the `ServiceDiscoveryManager` takes two arguments: an object that implements the `DiscoveryManagement` interface and a reference to a `LeaseRenewalManager` object. The constructor throws an `IOException` because construction of a `ServiceDiscoveryManager` may initiate the multicast discovery process, a process that can throw `IOException`.

In order to use the `ServiceDiscoveryManager`, an entity supplies an object through which notifications that indicate a lookup service has been discovered or discarded will be received. At a minimum, this object must satisfy the contract defined in the `DiscoveryManagement` interface. That is, this object must provide the `ServiceDiscoveryManager` with the ability to set discovery listeners and to discard previously discovered lookup services when they are found to be unavailable.

A value of `null` may be passed as the `DiscoveryManagement` argument. If the value of the argument is `null`, then an instance of the `LookupDiscoveryManager` utility class will be constructed to listen for events announcing the discovery of only those lookup services that are members of the public group.

A value of `null` may be passed as the `LeaseRenewalManager` argument. If the value of the argument is `null`, an instance of the `LeaseRenewalManager` class will be created, initially managing no `Lease` objects.

- ◆ The `createLookupCache` method allows an entity to request that the `ServiceDiscoveryManager` create a new managed set (or cache) and populate it with services, which match criteria defined by the entity, and whose references are registered with one or more of the lookup services the entity has targeted for discovery.

This method returns an object of type `LookupCache`. Through this return value, the entity can query the cache for services of interest, manage the cache's event mechanism for service discoveries, or terminate the cache. The specification of the `LookupCache` interface is presented later in this chapter.

An entity typically uses the object returned by this method to provide *local* storage of, and access to, references to services that it is interested in using. Entities needing frequent access to numerous services will find the object returned by this method quite useful because acquisition of those service references is provided through local method invocations. Additionally, because the object returned by this method provides an event mechanism, it is also useful to entities wishing to simply monitor, in an event-driven manner, the state changes that occur in the services of interest.

The `createLookupCache` method takes three arguments: an instance of `ServiceTemplate`, an instance of `ServiceItemFilter`, and an instance of `ServiceDiscoveryListener`. Both the `ServiceItemFilter` and `ServiceDiscoveryListener` interfaces are presented later in this chapter.

Together, the `tmpl` and the `filter` arguments define the criteria with which service-matching should be performed. The `listener` argument references an object that will receive notifications when services matching the input criteria are discovered for the first time, or have encountered a state change such as removal from all lookup services or attribute set changes. If `null` is input to the `listener` argument for a particular invocation of this method, the cache resulting from that invocation will send no such notifications.

The `tmpl` argument employs template-matching semantics identical to the semantics described in the *Jini™ Lookup Service Specification* (where the `ServiceTemplate` interface is defined) to identify the service(s) to acquire from lookup services in the managed set. The object passed to the `filter` argument is then used to apply additional matching criteria to any service references found through template-matching. The additional matching criteria defined by the `filter` parameter are application-specific, and therefore, must be defined by the client-like entity itself (as described in the section titled *The ServiceItemFilter Interface*). Furthermore, once an instance of the cache is created, any filter associated with that instance will not change during the life of that particular cache.

As a convenience, a `null` reference input to the `tmpl` argument is treated as equivalent to inputting a `ServiceTemplate` constructed with all `null` arguments (all *wildcards*). That is, the cache will attempt to discover all services contained in each lookup service in the managed set. If a `null` value is passed as the filter argument, then only template-matching will be employed to find the desired services.

Entities that invoke this method must take care not to modify the contents of the object input through the `tmpl` parameter after the cache has been created. Doing so could cause the state of the cache to become corrupted or inconsistent. It is for this reason that the effects of modifying the contents of the `tmpl` parameter, after this method is invoked, are undefined.

#### **SD.4.1.1 Events and the Cache**

In order to keep its contents up-to-date, the cache must register with the event mechanism of each lookup service in the managed set. From the point of view of the cache, a service is “discovered” when it receives a remote event from one of

those lookup services notifying the cache of the existence of a service matching the input criteria. In addition, whenever one of the cache's discovered services experiences a state change in one of the lookup services in which it is registered, the cache will receive a remote event identifying that state change whenever the change satisfies the matching criteria.

For a number of reasons the cache may receive multiple events corresponding to the same Jini service. For example, a particular Jini service may be registered with more than one lookup service from the managed set. Also, multiple configurations of the service may be registered with one or more of those lookup services. If the cache requests events from each lookup service using a template configured with no restriction along the service ID search axis and little or no restriction along the attribute search axis, the cache will receive a notification each time one of the following events occurs:

- ◆ Some configuration of the service, matching the template, is registered with one of the lookup services.
- ◆ The lease of one of the service references matching the template is cancelled or expires.
- ◆ An attribute set associated with one of the service references matching the template is modified in some way.

Just as the cache requests that it be notified of state changes in matching services occurring within each lookup service, an entity may request that the cache deliver events that indicate analogous state changes in the service references stored in the cache.

There are two significant differences in the event mechanism between the lookup services and the cache, and the event mechanism between the cache and the client-like entity. First and foremost, the events sent from the lookup services to the cache are *remote* events, whereas the events sent from the cache to the entity are *local* events. Secondly, each registration or state-change event sent from the cache to the entity may actually have been a result of multiple corresponding events received by the cache from a set of lookup services. Thus, there is a “many-to-one” relationship between the events received by the cache and the events sent by the cache.

For many entities that use the cache's event mechanism to interact with the cache's discovered services, knowledge of the number of distinct service references, as well as identification of the lookup services with which those references are registered, is of no interest. Such entities typically are interested only in acquiring *a* reference—not *all* references—to the desired services. Thus, the relationship between the two event mechanisms described previously allows the

`ServiceDiscoveryManager` to hide the lookup services with which the cache interacts from the entity. For entities that are interested in the additional information, the cache provides methods separate from the event mechanism for obtaining such information.

To summarize, although the cache may receive *multiple* events signaling a state change related to a particular matching service, the cache will typically send only a *single* corresponding event to the entity. That is, for any matching service:

- ◆ The cache will send a *service discovery event* to the entity only once: after the cache acquires the *first* reference to the matching service.
- ◆ The cache will send a *service removal event* to the entity only once: after every reference to the service has had its lease expire or cancelled; that is, only after all references to the matching service have been removed from every lookup service in the cache's managed set.
- ◆ For each set of event(s) notifying the cache that a particular modification has been made to the attribute set associated with one of the service references, one *service modification event* will be sent to the entity; but *only if* the attribute set state reflected in the received event represents an actual change in the service's current attribute set state (as maintained by the cache).

With respect to the state of the attribute sets associated with the service references stored in the cache, the cache should be viewed as maintaining a single attribute set state for each collection of service references that represent the same service. That single state will always be equivalent to the state reflected in the last attribute set modification event received by the cache.

For example, suppose three different references to a service which matches the input criteria are each registered with three lookup services in the managed set. Suppose the attribute sets associated with each service reference are modified in exactly the same way. For this specific case, the cache would receive three events—one from each lookup service—signaling these modifications. Upon receipt of the first event, the cache modifies its current notion of the service's attribute set state, and then notifies the entity of the change; but only if the state reflected in the event represents a change in the current state. Because the remaining two events received by the cache represent the same state change as that represented in the first event, the cache sends no other notification.

Next, suppose a second modification, different from the first, is made on only two of the service references, and a third unique modification is made on the remaining service reference. In this case, the cache will still receive three events, but how the cache handles the events is dependent on the order of arrival of the events. For simplicity, call the three events  $e_1$ ,  $e_2$ , and  $e_3$ . Use  $s$  to represent the

cache's current notion of the service's attribute set state, and use  $s_1$  and  $s_2$  to represent the states resulting after each attribute modification has occurred. In this example,  $e_1$  and  $e_2$  will be sent to the cache after the service's attribute sets are each modified to  $s_1$  in their respective lookup services. Event  $e_3$  is sent after the service's attribute sets are modified to  $s_2$  in the remaining lookup service.

If the order of arrival is  $e_1$ ,  $e_2$ , and then  $e_3$ , the cache will change  $s$  into  $s_1$  and notify the entity after the arrival of  $e_1$ , but will do nothing upon the arrival of  $e_2$ . Upon the arrival of  $e_3$ , the cache will change  $s$  (which is now  $s_1$ ) into  $s_2$ . If the order of arrival of the events is  $e_1$ ,  $e_3$ , and then  $e_2$ , the cache will first change  $s$  into  $s_1$ , then into  $s_2$ , and then back into  $s_1$  again. Furthermore, for each state change made, the cache will send a notification to the entity.

Thus, the events generated by the cache's event mechanism and sent by the cache to the entity are more representative of the state changes that occur in the cache than in the lookup services.

An entity may register for events from the cache in one of two ways. The entity may supply an instance of `ServiceDiscoveryListener` to the listener argument of the `createLookupCache` method, or it may invoke a method on the cache to add a listener to the cache. Thus, an entity may register for events from the cache at any time during the execution life of the cache.

Similarly, the cache provides a method that an entity, which is currently registered for events from the cache, may use at any time to un-register with the cache's event mechanism.

- ◆ The `lookup` method queries each available lookup service in the managed set for service reference(s) that match criteria defined by the entity that invokes this method. Entities typically employ this method when they need infrequent access to services, and when the cost of making remote queries is outweighed by the overhead of maintaining a local cache (for example, because of resource limitations).

Although the `lookup` method has four versions, each version falls into one of two categories: those versions of this method that return a single instance of `ServiceItem`, and those versions that return a set of service references as an array of `ServiceItem` objects.

Two arguments are common to all versions of this method: an instance of `ServiceTemplate` and an instance of `ServiceItemFilter`.

The difference between the two versions of `lookup` in each of the two categories differ only in whether or not a particular version provides what is referred to as a "wait" (or blocking) feature. That is, within each category, one version of `lookup` will return immediately upon failure (or success), whereas the blocking version will declare failure only after waiting (and fail-

ing) a specified amount of time. The particular version of this method that an entity employs is typically determined by the entity's intended usage pattern.

The descriptions here refer to all versions of this method, except where explicitly noted.

The `tmpl` argument and the `filter` argument both have semantics identical to that defined for these arguments in the description of the `createLookupCache` method above. In particular,

- ◆ A null reference value for the `tmpl` argument is treated as the equivalent of a “wildcarded” `ServiceTemplate`.
- ◆ If null is the value for the `filter` argument, then only template-matching will be employed to find the desired services.
- ◆ The effects of modifying the contents of the `tmpl` parameter while the invocation is in progress are unpredictable and undefined.

If no service can be found that matches the desired criteria, then the versions of `lookup` from the first category—those that return a single instance of `ServiceItem`—will return null; whereas the versions from the second category—those that return an array of `ServiceItem` instances—will return an empty array.

The versions of this method from the first category can be used in a fashion similar to the first form of the `lookup` method defined in the `ServiceRegistrar` interface described in the *Jini™ Lookup Service Specification*. That is, an entity would typically invoke one of these versions of `lookup` when it wishes to find a *single* service reference, and the particular lookup service with which that service reference is registered is unimportant to the entity.

These versions of `lookup` differ with the corresponding version of `lookup` in `ServiceRegistrar` in the following ways:

- ◆ These versions of `lookup` query *multiple* lookup services (the order in which the lookup services are queried is defined by the implementation).
- ◆ These versions of `lookup` can apply additional matching criteria, in the form of a filter object, when deciding whether a service reference found through standard template-matching should be returned to the entity.

The two versions of this method that return an array of `ServiceItem` objects can be used in a fashion similar to the second form of `lookup` defined in the



`ServiceRegistrar` interface. That is, an entity would typically invoke these versions of `lookup` when it wishes to find *multiple* service references matching the input criteria. Each of the versions of `lookup` that return an array of `ServiceItem` objects takes as one of its arguments an `int` parameter, `maxMatches`, that represents the maximum number of matches that should be returned. The array returned by these methods will contain no more than `maxMatches` service references, although it may contain fewer than that number.

As with the versions of `lookup` that return a single instance of `ServiceItem`, multiple queries and filtering are also notable differences between the second-category versions of this method and their counterpart in `ServiceRegistrar`.

For all of the versions of `lookup`, whenever a lookup service query returns a `null` service reference, the filter will be by-passed. On the other hand, whenever a lookup service query returns a non-`null` service reference in which one or more elements of the associated attribute sets array is `null`, the filter will be applied, but with the `null` elements of the attribute sets array removed. Upon finding a successful match, the service item will be returned in the appropriate way, with the `null` elements of the attribute sets array removed.

Each version of `lookup` may be confronted with *duplicate* references during a search for a service of interest. This is because the same service may register with more than one lookup service in the managed set. As with the cache, when a set of service references is returned by `lookup`, each service reference in the return set will be unique with respect to all other service references in the set; as determined by the `equals` method provided by each reference.

If it is determined that a lookup service is unavailable (due to an exception or some other non-fatal error) while interacting with a lookup service from the managed set, all versions of `lookup` will invoke the `discard` method on the instance of `DiscoveryManagement` being employed by the `ServiceDiscoveryManager`. Doing so will result in the unavailable lookup service being discarded and made eligible for re-discovery.

Recall that the propagation of modifications to a service's attributes across a set of lookup services typically occurs asynchronously. It is for this reason that while invoking `lookup` to find a set of matching services, it is possible that the set returned may reflect multiple references having the same service ID with different attributes. Note that although this sort of inconsistent state

can also occur if the entity employs a cache, the cache will eventually reflect the correct state.

#### **SD.4.1.2 The Blocking Feature of Lookup**

As noted above, each category contains a version of `lookup` that provides a feature in which the entity can request that if the number of service references found throughout the available lookup services does not fall into a desired range, the method will wait a finite period of time until either an acceptable minimum number of service references are discovered, or the specified time period has passed.

The versions of `lookup` providing this blocking feature each takes as one of its parameters a value of type `long` that represents the number of milliseconds to wait for the service to be discovered. In addition to `RemoteException` (described previously for these methods), each of these versions of `lookup` may throw an `InterruptedException`.

One of these blocking versions of `lookup` implicitly uses a value of one for both the acceptable minimum and the allowable maximum number of service references to discover. The other blocking version requires that the entity specify the range through the `minMatches` and `maxMatches` parameters respectively.

Prior to blocking, each of these versions of `lookup` first queries each available lookup service in an attempt to retrieve a satisfactory number of matching services. Whether or not the method actually blocks is dependent on how many matching service references are found during the query process. Blocking occurs only if after querying *all* of the available lookup services, the number of matching services found is less than the acceptable minimum. If the waiting period passes before that minimum number of service references is found, the method will return the service references that have been discovered up to that point. If the waiting period passes and no services have been found, `null` or an empty array (depending on the version of `lookup`) will be returned.

If after querying all of the available lookup services the number of matching services found is greater than or equal to the specified minimum, but less than the specified maximum, the method will return the currently discovered service references without blocking. If the initial query process produces the desired maximum number of service references, the method will return the results immediately.

The blocking versions of `lookup` are quite useful to entities that cannot proceed until such a service of interest is found. If a non-positive value is input to the `waitDur` argument, then the method will not wait. It will simply query the available lookup services and employ the return semantics described above.

The values of the `minMatches` and `maxMatches` arguments must both be positive, and `maxMatches` must be greater than or equal to `minMatches`; otherwise, an `IllegalArgumentException` will be thrown.

The blocking versions of `lookup` make a concurrency guarantee with respect to the discovery of new lookup services during the wait period. That is, while waiting for matching service reference(s) to be discovered, if one or more of the desired—but previously unavailable—lookup services is discovered and added to the managed set, those new lookup services will also be queried for the service(s) of interest.

Additionally, both blocking versions of `lookup` throw an `InterruptedException`. When an entity invokes either version with valid parameters, the entity may decide during the wait period that it no longer wishes to wait the entire period for the method to return. Thus, while the method is blocking on the discovery of matching service(s), it may be interrupted by invoking the `interrupt` method from the `Thread` class. The intent of this mechanism is to allow the entity to interrupt a blocking `lookup` in the same way it would a sleeping thread.

- ◆ The `getDiscoveryManager` method will return an object that implements the `DiscoveryManagement` interface. The object returned by this method provides the `ServiceDiscoveryManager` with the ability to set discovery listeners and to discard previously discovered lookup services when they are found to be unavailable. This method takes no arguments.
- ◆ The `getLeaseRenewalManager` method will return an instance of the `LeaseRenewalManager` class. The object returned by this method manages the leases requested and held by the `ServiceDiscoveryManager`. In general, these leases correspond to the registrations made by the `ServiceDiscoveryManager` with the event mechanism of each lookup service in the managed set. This method takes no arguments.
- ◆ The `terminate` method performs cleanup duties related to the termination of the event mechanism for *lookup service* discovery, the event mechanism for *service* discovery, and the cache management duties of the `ServiceDiscoveryManager`.

For each instance of `LookupCache` created and managed by the `ServiceDiscoveryManager`, the `terminate` method will do the following:

- Either remove all listener objects registered for receipt of `DiscoveryEvent` objects or, if the discovery manager employed by the `ServiceDiscoveryManager` was created by the

`ServiceDiscoveryManager` itself, terminate *all* discovery processing being performed by that manager object on behalf of the entity.

- Cancel all event leases granted by each lookup service in the managed set of lookup services.
- Un-export all remote listener objects registered with each lookup service in the managed set.
- Terminate all threads involved in the process of retrieving and storing references to discovered services of interest.

Note that if the entity obtains references to the discovery manager (as long as it wasn't created by the `ServiceDiscoveryManager` itself) and the lease renewal manager prior to termination, those references will still be valid after termination.

The `ServiceDiscoveryManager` makes certain concurrency guarantees with respect to an invocation of the `terminate` method while other method invocations are in progress. The termination process described above will not begin until completion of all invocations of the methods defined in the public interface of the `ServiceDiscoveryManager`; that is, until completion of invocations of `createLookupCache`, `lookup`, `getDiscoveryManager`, and `getLeaseRenewalManager`.

Additionally, once the termination process has begun, no remote method invocations will be made by the `ServiceDiscoveryManager`, and all other method invocations on the `ServiceDiscoveryManager` will not return until the termination process has completed. Upon completion of the termination process, the semantics of all current and future method invocations on the current instance of the `ServiceDiscoveryManager` are undefined.

---

## SD.5 Supporting Interfaces and Classes

THE `ServiceDiscoveryManager` utility class depends on the following interfaces defined in the *Jini™ Lookup Service Specification*: `ServiceTemplate`, `ServiceItem`, and `ServiceMatches`. This class also depends on a number of interfaces, each defined in this section; those interfaces are: `DiscoveryManagement`, `ServiceItemFilter`, `ServiceDiscoveryListener`, and `LookupCache`.

The `ServiceDiscoveryManager` class references the following concrete classes: `LookupDiscoveryManager` and `LeaseRenewalManager`, each described in a separate chapter of this document, and `ServiceDiscoveryEvent` defined in this chapter.

### SD.5.1 The `DiscoveryManagement` Interface

Although it is not necessary for the `ServiceDiscoveryManager` itself to execute the discovery process, it does need to be notified when one of the lookup services it wishes to query is discovered or discarded. Thus, at a minimum, the `ServiceDiscoveryManager` requires access to the instances of `DiscoveryEvent` sent to the listeners registered with the event mechanism of the discovery process. The instance of `DiscoveryManagement` passed to the constructor of the `ServiceDiscoveryManager` provides a mechanism for acquiring access to those events. For a complete description of the semantics of the methods of this interface, refer to the *Jini™ Discovery Utilities Specification*.

One noteworthy item about the semantics of the `ServiceDiscoveryManager` is the effect that invocations of the `discard` method of `DiscoveryManagement` have on any cache objects created by the `ServiceDiscoveryManager`. The `DiscoveryManagement` interface specifies that the `discard` method will remove a particular lookup service from the managed set of lookup services that have already been discovered, allowing that lookup service to be re-discovered. Invoking this method will result in the flushing of the lookup service from the appropriate cache. This effect ultimately causes a `discard` notification to be sent to all `DiscoveryListener` objects registered with the event mechanism of the discov-

ery process (including all listeners registered by the `ServiceDiscoveryManager`).

The receipt of an event notification indicating that a lookup service from the managed set has been discarded must ultimately result in the cancellation and removal of all event leases that were granted by the discarded lookup service, and that are managed by the `LeaseRenewalManager` on behalf of the `ServiceDiscoveryManager`.

Furthermore, every service reference stored in the cache that is registered with the discarded lookup service but is not registered with any of the remaining lookup services in the managed set, will be “discarded” as well. That is, all previously discovered service references that are registered with only unavailable lookup services, will be removed from the cache and made eligible for service re-discovery.

## SD.5.2 The `ServiceItemFilter` Interface

The `ServiceItemFilter` interface defines the methods used by an object such as the `ServiceDiscoveryManager` or the `LookupCache` to apply additional matching criteria when searching for services in which an entity has registered interest. It is the responsibility of the entity requesting the application of additional criteria to construct an implementation of this interface that defines the additional criteria, and to pass the resulting object (referred to as a *filter*) into the object that will apply it.

The filtering mechanism provided by implementations of this interface is particularly useful to entities that wish to extend the capabilities of the standard template matching scheme. For example, because template matching does not allow one to search for services based on a range of attribute values, this additional matching mechanism can be exploited by the entity to ask the managing object to find all registered printer services that have a resolution attribute between say, 300 dpi and 1200 dpi.

```
package net.jini.lookup;

public interface ServiceItemFilter
{
    public boolean check(ServiceItem item);
}
```

### SD.5.2.1 The Semantics

The check method defines the implementation of the additional matching criteria to apply to a `ServiceItem` object found through standard template matching. This method takes one argument: the `ServiceItem` object to test against the additional criteria. This method returns `true` if the input object satisfies the additional criteria, and `false` otherwise.

Neither a `null` reference nor a `ServiceItem` object containing `null` fields will be passed into this method by the `ServiceDiscoveryManager`.

If the parameter input to this method is a `ServiceItem` object that has non-`null` fields, but is associated with attribute sets containing `null` entries, this method must process that parameter in a reasonable manner.

Should an exception occur during an invocation of this method, the semantics of how that exception is handled are undefined.

This method must not modify the contents of the input `ServiceItem` object because it could result in unpredictable and undesirable effects on future processing by the `ServiceDiscoveryManager`. That is why the effects of any such modification to the contents of that input parameter are undefined.

## SD.5.3 The `ServiceDiscoveryEvent` Class

The `ServiceDiscoveryEvent` class encapsulates the service discovery information made available by the event mechanism of the `LookupCache`. All listeners that an entity has registered with the cache's event mechanism will receive an event of type `ServiceDiscoveryEvent` upon the discovery, removal, or modification of one of the cache's services, as described previously in the section titled *Events and the Cache*.

This class is a sub-class of the class `EventObject`. In addition to the methods of the `EventObject` class, this class provides two additional accessor methods which can be used to retrieve the additional state associated with the event: `getPreEventServiceItem` and `getPostEventServiceItem`.

The `getSource` method of the `EventObject` class returns the instance of `LookupCache` from which the given event originated.

```
package net.jini.lookup;

public class ServiceDiscoveryEvent extends EventObject
{
    public ServiceDiscoveryEvent(Object source,
                                ServiceItem preEventItem,
```

```
ServiceItem
postEventItem){...}

    public ServiceItem getPreEventServiceItem(){...}
    public ServiceItem getPostEventServiceItem(){...}
}
```

### SD.5.3.1 The Semantics

The constructor of `ServiceDiscoveryEvent` takes three arguments:

- ◆ An instance of `Object` corresponding to the instance of `LookupCache` from which the given event originated
- ◆ A `ServiceItem` reference representing the state of the service (associated with the given event) *prior to* the occurrence of the event
- ◆ A `ServiceItem` reference representing the state of the service *after* the occurrence of the event

If `null` is passed as the `source` parameter for the constructor, a `NullPointerException` will be thrown.

Depending on the nature of the discovery event, a `null` reference may be passed as one or the other of the remaining parameters, but never both. If `null` is passed as both the `preEventItem` and the `postEventItem` parameters, a `NullPointerException` will be thrown.

Note that the constructor will not modify the contents of either `ServiceItem` argument. Doing so can result in unpredictable and undesirable effects on future processing by the `ServiceDiscoveryManager`. That is why the effects of any such modification to the contents of either input parameter are undefined.

The `getPreEventServiceItem` method returns an instance of `ServiceItem` containing the service reference corresponding to the given event. The service state reflected in the returned service item is the state of the service *prior to* the occurrence of the event.

If the event is a discovery event (as opposed to a removal or modification event), then this method will return `null` because the discovered service had no state in the cache prior to its discovery.

The `getPostEventServiceItem` method returns an instance of `ServiceItem` containing the service reference corresponding to the given event. The service state reflected in the returned service item is the state of the service *after* the occurrence of the event.



If the event is a removal event, then this method will return `null` because the discovered service has no state in the cache after it is removed from the cache.

Because making a copy can be a very expensive process, neither accessor method returns a copy of the service reference associated with the given event. Rather, each method returns the appropriate service reference from the cache itself. Due to this cost, listeners (see `ServiceDiscoveryListener`, next) that receive a `ServiceDiscoveryEvent` must not modify the contents of the object returned by these methods; doing so could cause the state of the cache to become corrupted or inconsistent because the objects returned by these methods are also members of the cache. This potential for corruption or inconsistency is why the effects of modifying the object returned by either accessor method are undefined.

## SD.5.4 The `ServiceDiscoveryListener` Interface

The `ServiceDiscoveryListener` interface defines the methods used by objects such as a `LookupCache` to notify an entity that events of interest related to the elements of the cache have occurred. It is the responsibility of the entity wishing to be notified of the occurrence of such events to construct an object that implements the `ServiceDiscoveryListener` interface and then register that object with the cache's event mechanism. Any implementation of this interface must define the actions to take upon receipt of an event notification. The action taken is dependent on both the application and the particular event that has occurred.

```
package net.jini.lookup;

public interface ServiceDiscoveryListener
{
    public void serviceAdded(ServiceDiscoveryEvent event);
    public void serviceRemoved(ServiceDiscoveryEvent event);
    public void serviceChanged(ServiceDiscoveryEvent event);
}
```

### SD.5.4.1 The Semantics

As described previously in the section titled *Events and the Cache*: when the cache receives from one of the managed lookup services, an event signaling the *registration* of a service of interest for the *first time* (or for the first time since the service has been discarded), the cache invokes the `serviceAdded` method on all instances of `ServiceDiscoveryListener` that are registered with the cache; doing so notifies the entity that a service of interest has been discovered. The ser-

viceAdded method takes one argument: an instance of `ServiceDiscoveryEvent` containing references to the service item corresponding to the event, including representations of the service's state both before and after the event.

When the cache receives, from a managed lookup service, an event signaling the *removal* of a service of interest from the *last* such lookup service with which it was registered, the cache invokes the `serviceRemoved` method on all instances of `ServiceDiscoveryListener` that are registered with the cache; doing so notifies the entity that a service of interest has been discarded. The `serviceRemoved` method takes one argument: a `ServiceDiscoveryEvent` object containing references to the service item corresponding to the event, including representations of the service's state both before and after the event.

When the cache receives, from a managed lookup service, an event signaling the unique *modification* of the attributes of a service of interest (across the attribute sets of all references to the service), the cache invokes the `serviceChanged` method on all instances of `ServiceDiscoveryListener` that are registered with the cache; doing so notifies the entity that the state of a service of interest has changed. The `serviceChanged` method takes one argument: a `ServiceDiscoveryEvent` object containing references to the service item corresponding to the event, including representations of the service's state both before and after the event.

Should an exception occur during an invocation of any of the methods defined by this interface, the semantics of how that exception is handled are undefined.

Each method defined by this interface must not modify the contents of the `ServiceDiscoveryEvent` parameter; doing so can result in unpredictable and undesirable effects on future processing by the `ServiceDiscoveryManager`. It is for this reason that if one of these methods modifies the contents of the parameter, the effects are undefined.

This interface makes the following concurrency guarantee. For any given listener object that implements this interface, no two methods (either the same two methods or different methods) defined by the interface can be invoked at the same time by the same cache. For example, the `serviceRemoved` method must not be invoked while the invocation of another listener's `serviceAdded` method is in progress.

Finally, it should be noted that the intent of the methods of this interface is to allow the recipient of the `ServiceDiscoveryEvent` to be informed that a service has been added to, removed from, or modified in the cache. Calls to these methods are synchronous to allow the entity that makes the call (for example, a thread that interacts with the various lookup services of interest) to determine whether or not the call succeeded. However, it is not part of the semantics of the call that the notification return can be delayed while the recipient of the call reacts to the occurrence of the event. Thus, it is highly recommended that implementations of this

interface avoid time consuming operations, and return from the method as quickly as possible. For example, one strategy might be to simply note the occurrence of the `ServiceDiscoveryEvent`, and perform any time consuming event handling asynchronously.

## SD.5.5 The LookupCache Interface

The `LookupCache` interface defines the methods provided by the object created and returned by the `ServiceDiscoveryManager` when an entity invokes the `createLookupCache` method. It is within that object that discovered service references, which match criteria defined by the entity, are stored. Through this interface the entity may retrieve one or more of the stored service references, register and un-register with the cache's event mechanism, and terminate all of the cache's processing.

```
package net.jini.lookup;

public interface LookupCache
{
    public ServiceItem    lookup(ServiceItemFilter filter);

    public ServiceItem[]  lookup(ServiceItemFilter filter,
                                int maxMatches);

    public void addListener
                        (ServiceDiscoveryListener listener);
    public void removeListener
                        (ServiceDiscoveryListener listener);

    public void discard(Object serviceReference);

    public void terminate();
}
```

### SD.5.5.1 The Semantics

Depending on which version is invoked, the `lookup` method of the `LookupCache` interface returns one or more elements — each matching the input criteria — that were stored in the associated cache. The object returned is either a single instance of `ServiceItem`, or a set of service references in the form of an array of

`ServiceItem` objects. Each service item returned by either form of this method must have been previously discovered to be both registered with one or more of the lookup services in the managed set, and to match criteria defined by the entity.

One argument is common to both forms of lookup: an instance of `ServiceItemFilter`. The semantics of the filter argument are identical to those of the `filter` argument specified for a number of the methods defined in the interface of the `ServiceDiscoveryManager` utility class. This argument is intended to allow an entity to separate its filtering into two steps: an initial filter applied during the discovery phase, and then a finer resolution filter applied upon retrieval from the cache. As with the methods of the `ServiceDiscoveryManager`, if `null` is the value of this argument, then no additional filtering will be performed.

The second form of the `lookup` method of the `LookupCache` interface takes an additional argument: a parameter of type `int` that represents the maximum number of matches that should be returned. The array returned by this form of lookup will contain no more than the requested number of service references, although it may contain less than that number. The value input to this argument must be positive, otherwise an `IllegalArgumentException` will be thrown.

If the cache is empty, or if no service can be found that matches the input criteria, then the first form of lookup will return `null`; whereas the second form of lookup will return an empty array. The algorithm used to select the return element(s) from the set of matching service references is implementation dependent.

Neither form of the `lookup` method of the `LookupCache` interface returns a copy of the matching service reference(s) that were selected; rather, each form returns the actual service reference(s) from the cache itself. Because the actual service reference(s) are returned, entities that invoke either form of this method must not modify the contents of the returned reference(s). Modifying the returned service reference(s) could cause the state of the cache to become corrupted or inconsistent. This potential for corruption or inconsistency is why the effects of modifying the service reference(s) returned by either form of lookup is undefined.

Typically, an entity will request the creation of a separate cache for each service type of interest. When the entity simply needs a reference to a service of a particular type, the entity should invoke the first form of lookup to retrieve one element from the cache; in this case, which particular service reference that is returned will not, in general, matter to the entity. If for some reason it does matter to an entity which service reference is returned, then the entity can invoke the second form of lookup requesting that `Integer.MAX_VALUE` service references be returned; doing so will return all elements of the cache that match the input criteria. The entity can then iterate through each element, selecting the desired reference.

The `addListener` method will register a `ServiceDiscoveryListener` object with the event mechanism of a `LookupCache`. This listener object will receive a `ServiceDiscoveryEvent` upon the discovery, removal, or modification of one of the cache's services, as described previously in the section titled *Events and the Cache*. This method takes one argument: a reference to the `ServiceDiscoveryListener` object to register.

Once a listener is registered, it will be notified of all service references discovered to date, and will be notified as new services are discovered and existing services are modified or discarded. If the parameter value is `null`, no action will be taken.

The `LookupCache` makes a re-entrancy guarantee with respect to any `ServiceDiscoveryListener` objects registered with it. Should the `LookupCache` invoke a method on a registered listener (a local call), any call from that method to a local method of the `LookupCache` is guaranteed not to result in a deadlock condition.

The `removeListener` method will remove a `ServiceDiscoveryListener` object from the set of listeners currently registered with a `LookupCache`. Once all listeners are removed from the cache's set of listeners, the cache will send no more `ServiceDiscoveryEvent` notifications. This method takes one argument: a reference to the `ServiceDiscoveryListener` object to remove.

If the parameter value to `removeListener` is `null`, or if the listener passed to this method does not exist in the set of listeners maintained by the implementation class, then this method will take no action.

If an entity determines that a service reference retrieved from the cache is no longer available, the entity should request the removal of that reference from the cache. The mechanism for discarding an unavailable service from the cache is provided by the `discard` method of the `LookupCache` interface. The `discard` method takes one argument: an instance of `Object` whose reference is the service reference to remove from the cache. If the proxy input to this method is `null`, or if it matches (using the `equals` method) none of the service references in the cache, this method takes no action.

The `discard` method not only deletes the service reference from the cache, but also causes a notification to be sent to all registered listeners indicating that the service has been discarded (see the description of the `serviceRemoved` method in the section that specifies the `ServiceDiscoveryListener` interface). The service is guaranteed to have been removed from the cache when this method completes successfully; the service is then said to have been "discarded". No such guarantee is made with respect to when the discard event is sent to the client's registered listeners. That is, the event notifying the client that the service has been discarded may or may not be sent asynchronously.

Note that after the service has been discarded, there is no guarantee that the service will again become eligible for discovery. If the service's residency in at least one lookup service from the managed set ends (usually because the service's lease with that lookup service has expired), the service will be re-discovered when it becomes available and re-registers with that lookup service.

The `terminate` method performs cleanup duties related to the termination of the processing being performed by a particular instance of `LookupCache`. For that instance, this method cancels all event leases granted by the lookup services that supplied the contents of the cache, and un-exports all remote listener objects registered with those lookup services. The `terminate` method is typically called when the entity is no longer interested in the contents of the `LookupCache`.

---

# Jini™ Lookup Attribute Schema Specification

## LS.1 Introduction

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 that it provides for registering and querying based on attributes is centered on 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 by 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.

### LS.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

### LS.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 toward 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 Section LS.2)

We will cover human accessibility of attributes soon.

### **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.

## **LS.1.3 Dependencies**

This document relies on the following other specifications:

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

## **LS.1.4 Comments**

Please direct comments to [jini-comments@java.sun.com](mailto:jini-comments@java.sun.com).



---

## LS.2 Human Access to Attributes

### LS.2.1 Providing a Single View of an Attribute's Value

**C**ONSIDER 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.
- ◆ Provide either a non-default implementation of the `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.



---

## LS.3 JavaBeans Components and Design Patterns

### LS.3.1 Allowing Display and Modification of Attributes

**W**E use JavaBeans components to provide a layer of abstraction on top of the individual classes that implement the `net.jini.core.entry.Entry` interface. 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.

#### LS.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.

### LS.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.

### LS.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 it is passed. If a problem occurs creating the JavaBeans component, the method throws either `java.io.IOException` or `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. 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, an `IllegalArgumentException` is thrown. If the given attribute class cannot be found, a `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.



---

## LS.4 Generic Attribute Classes

**W**E will now describe 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
```

### LS.4.1 Indicating User Modifiability

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.

### LS.4.2 Basic Service Information

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

```
public class ServiceInfo extends AbstractEntry
    implements ServiceControlled
{
    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 register itself with only 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 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 reseller. This field exists so that in cases in which resellers relabel 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.

### LS.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 extends AbstractEntry
    implements ServiceControlled
{
    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 class `ServiceType` 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.

### LS.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”).

### LS.4.5 Adding a Comment to a Service

In cases in which 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.

## LS.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 might 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.

## LS.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 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).

## LS.4.8 Serialized Forms

Class	serialVersionUID	Serialized Fields
Address	2896136903322046578L	<i>all public fields</i>
AddressBean	4491500432084550577L	Address asoc
Comment	7138608904371928208L	<i>all public fields</i>
CommentBean	5272583409036504625L	Comment asoc
Location	-3275276677967431315L	<i>all public fields</i>
LocationBean	-4182591284470292829L	Location asoc
Name	2743215148071307201L	<i>all public fields</i>
NameBean	-6026791845102735793L	Name asoc

Class	serialVersionUID	Serialized Fields
ServiceInfo	-1116664185758541509L	<i>all public fields</i>
ServiceInfoBean	8352546663361067804L	ServiceInfo asoc
ServiceType	-6443809721367395836L	<i>all public fields</i>
Status	-5193075846115040838L	<i>all public fields</i>
StatusBean	-1975539395914887503L	Status asoc
StatusType	-8268735508512712203L	int type



# LD

---

## Jini™ Lookup Discovery Service

### LD.1 Introduction

**P**ART of the *Jini™ Discovery and Join Specification* is devoted to defining the discovery requirements for well-behaved Jini clients and services, called *discovering entities*, which are required to participate in the multicast discovery protocols. Discovering entities are required to send multicast discovery requests to lookup services with which the entities wish to interact. In addition, they must continuously listen for and act on announcements from the desired lookup services. Interactions with a discovered lookup service may involve registration with that lookup service, or may simply involve querying the lookup service for services of interest (or both). In order to find *specific* lookup services, discovering entities also need to be able to participate in the unicast discovery protocol.

Under certain circumstances, a discovering entity may find it useful to allow a third party to perform the entity's discovery duties. For example, an activatable entity that wishes to deactivate may wish to employ a special Jini service—referred to as a *lookup discovery service*—to perform discovery duties on its behalf. Such an entity may wish to deactivate for various reasons, one being to conserve computational resources. While the entity is inactive, the lookup discovery service, running on the same or a separate host, would employ the discovery protocols to find lookup services in which the entity has expressed interest, and would notify the entity when a previously unavailable lookup service has become available.

The facilities of the lookup discovery service are of particular value in a scenario where a new lookup service is added to a long-lived djinn containing multiple inactive services. Without the use of a lookup discovery service, the time

frame over which the new lookup service is fully populated can be unpredictable and unbounded.

Because an inactive service has no way of discovering a new lookup service, each inactive service wishing to discover and join a new lookup service must first activate. Since activation of an inactive service occurs when some client attempts to use the service, the amount of time that passes between the arrival of the new lookup service and the activation of the service can vary greatly over the range of services in the djinn. Thus, the time frame over which the lookup service becomes fully populated is unpredictable since it could take arbitrarily long before all of the services activate and then discover and join the new lookup service.

The time frame over which the lookup service becomes fully populated is not only unpredictable, it is also unbounded because there is no guarantee that the lookup service will send multicast announcements between the time the service activates and the time it deactivates. If the timing is right, it is possible that one or more of the services in the djinn may never discover and join the new lookup service. Thus, without the use of the lookup discovery service, the new lookup service may never fully populate.

As another example of a discovering entity that may find it useful to allow a third party to perform the entity's discovery duties, consider an entity that exists in an environment with one of the following characteristics:

- ◆ The environment does not support multicast.
- ◆ The environment contains no lookup services within the entity's *multicast radius* (roughly, the number of hops beyond which neither the multicast requests from the entity, nor the multicast announcements from the lookup service, will propagate).
- ◆ The environment does contain lookup service(s) within the entity's multicast radius, but at least one service needed by the entity is not registered with any lookup service within that radius.

If such an entity was provided with references to lookup services—located outside of the entity's multicast radius—which contain services needed by the entity, the entity could contact each lookup service and retrieve the desired service references. One way to provide the entity with access to those lookup services might be to configure the entity to find and use a lookup discovery service, operating beyond the entity's range, that can employ multicast discovery to find nearby lookup services belonging to groups in which the entity has expressed interest. After acquiring references to the targeted lookup services, the lookup discovery service would pass those references to the entity, providing the entity with access to the services registered with each lookup service. In this way, the entity partici-

pates in the multicast discovery protocols through a proxy relationship with the lookup discovery service, gaining access not only to lookup services outside of its own range, but also to all of the services registered with those lookup services.

Note that the scenario just described does not come without restrictions. In order for the lookup discovery service to be able to “link” an entity with lookup services in the way just described, the lookup discovery service must be registered with a lookup service having a location that is either known to the entity, or is within the multicast radius of the entity. Furthermore, the lookup discovery service must be running on a host that is located within the multicast radius of the lookup services with which the entity wishes to be linked. That is, the entity must be able to find the lookup discovery service, and the lookup discovery service must be able to find the other desired lookup services.

To address these scenarios, the lookup discovery service participates in both the multicast discovery protocols and the unicast discovery protocol on behalf of a registered discovering entity or *client*. This service will listen for and process multicast announcement packets from Jini lookup services, and will, until successful, repeatedly attempt to discover specific lookup services that the client is interested in finding.

Upon discovery of a previously undiscovered lookup service of interest, the lookup discovery service notifies all entities that have requested the discovery of that lookup service that such an event has occurred. The event mechanism employed by the lookup discovery service satisfies the requirements defined in the *Jini™ Distributed Event Specification*. Note that the entity that receives such an event notification does not have to be the client of the lookup discovery service; it may be a third-party event-handling service such as an event mailbox service. Once a client is notified of the discovery of a lookup service, it is left to the client to define the semantics of how it interacts with that lookup service. For example, the client may wish to join the lookup service, simply query it for other useful services, or both.

The lookup discovery service must be implemented as a well-behaved Jini service, and must comply with all of the policies embodied in the Jini technology programming model. Thus, the resources granted by this service are leased, and implementations of this service must adhere to the distributed leasing model for Jini technology as defined in the *Jini™ Distributed Leasing Specification*. That is, the lookup discovery service will only grant its services for a limited period of time without an active expression of continuing interest on the part of the client.

### LD.1.0.1 Goals & Requirements

The requirements of the interfaces and classes specified in this document are as follows:

- ◆ To define a service that not only employs the Jini discovery protocols to discover, by way of either group association or `LookupLocator` association, lookup services in which clients have registered interest, but also which notifies its clients of the discovery of those lookup services
- ◆ To provide this service in such a way that it can be used by entities that deactivate
- ◆ To comply with the policies embodied in the Jini technology programming model

The goals of this document are as follows:

- ◆ To describe the lookup discovery service
- ◆ To provide guidance in the use and deployment of services that implement the `LookupDiscoveryService` interface and its related classes and interfaces

### LD.1.1 Other Types

The types defined in the specification of the `LookupDiscoveryService` interface are in the `net.jini.discovery` package. The following object types may be referenced in this chapter. Whenever referenced, these object types will be referenced in unqualified form:

```
net.jini.core.discovery.LookupLocator  
net.jini.core.event.EventRegistration  
net.jini.core.event.RemoteEventListener  
net.jini.core.lease.Lease  
net.jini.core.lookup.ServiceID  
net.jini.core.lookup.ServiceRegistrar  
net.jini.discovery.DiscoveryEvent  
net.jini.discovery.DiscoveryGroupManagement  
net.jini.discovery.DiscoveryListener  
java.io.IOException  
java.rmi.MarshalledObject
```

```
java.rmi.NoSuchObjectException  
java.rmi.RemoteException  
java.util.Map
```



---

## LD.2 The Interface

**T**HE `LookupDiscoveryService` interface defines the service—referred to as the *lookup discovery service*—previously introduced in this specification. Through this interface, other Jini services and clients may request that discovery processing be performed on their behalf. This interface belongs to the `net.jini.discovery` package, and any service implementing this interface must comply with the definition of a Jini service. This interface is not a remote interface; each implementation of this service exports frontend proxy objects that implement this interface local to the client, using an implementation-specific protocol to communicate with the actual remote server (the backend). All of the proxy methods must obey normal Java Remote Method Invocation (RMI) remote interface semantics except where explicitly noted. Two proxy objects are equal (using the `equals` method) if they are proxies for the same lookup discovery service.

The one method defined in this interface throws a `RemoteException`, and only requires the default serialization semantics so that this interface can be implemented directly using Java RMI.

```
package net.jini.discovery;

public interface LookupDiscoveryService {
    public LookupDiscoveryRegistration register
        (String[] groups,
         LookupLocator[] locators,
         RemoteEventListener listener,
         MarshallableObject handback,
         long leaseDuration)
        throws RemoteException;
}
```

When requesting a registration with the lookup discovery service, the client indicates the lookup services it is interested in discovering by submitting two sets of objects. Each set may contain zero or more elements. One set consists of the names of the groups whose members are lookup services the client wishes to be

discovered. The other set consists of LookupLocator objects, each corresponding to a specific lookup service the client wishes to be discovered.

For each successful registration, the lookup discovery service will manage both the set of group names and the set of locators submitted. These sets will be referred to as the *managed set of groups*, and the *managed set of locators*, respectively. The managed set of groups associated with a particular registration contains the names of the groups whose members consist of lookup services that the client wishes to be discovered through *multicast discovery*. Similarly, the managed set of locators contains instances of LookupLocator, each corresponding to a specific lookup service that the client wishes to be discovered through *unicast discovery*. The references to the lookup services that have been discovered will be maintained in a set referred to as the *managed set of lookup services* (or managed set of *registrars*).

Note that when the general term *managed set* is used, it should be clear from the context whether groups, locators, or registrars are being discussed. Furthermore, when the term *group discovery* or *locator discovery* is used, it should be taken to mean, respectively, the employment of either the multicast discovery protocols or the unicast discovery protocol to discover lookup services that correspond to members of the appropriate managed set.



---

## LD.3 The Semantics

**I**N order to employ the lookup discovery service to perform discovery on its behalf, a client must first register with the lookup discovery service by invoking the `register` method defined in the `LookupDiscoveryService` interface. The `register` method is the only method specified by this interface.

### LD.3.1 Registration Semantics

An invocation of the `register` method produces an object—referred to as a registration object (or simply, a *registration*)—that is mutable. That is, the registration object contains methods through which it may be changed. Because the returned object is mutable, each invocation of the `register` method produces a new registration object. Thus, the `register` method is not idempotent.

The `register` method may throw a `RemoteException`. Typically, this exception occurs when there is a communication failure between the client and the lookup discovery service. When this exception does occur, the registration may or may not have been successful.

Each registration with the lookup discovery service is persistent across restarts (or crashes) of the lookup discovery service, until the lease on the registration expires or is cancelled.

The `register` method takes the following as arguments:

- ◆ A `String` array, none of whose elements may be `null`, consisting of zero or more elements in which each element is the name of a group whose members are lookup services the client requesting the registration wishes to be discovered via group discovery
- ◆ An array of `LookupLocator` objects, none of whose elements may be `null`, consisting of zero or more elements in which each element corresponds to a specific lookup service the client requesting the registration wishes to be discovered via locator discovery

- ◆ A non-null `RemoteEventListener` object which specifies the entity that will receive events notifying the registration when a lookup service of interest is discovered or discarded
- ◆ Either `null` or an instance of `MarshaledObject` specifying an object that will be included in the notification event that the lookup discovery service sends to the registered listener
- ◆ A `long` value representing the amount of time (in milliseconds) for which the resources of the lookup discovery service are being requested

The `register` method returns an object that implements the `LookupDiscoveryRegistration` interface. It is through this returned object that the client interacts with the lookup discovery service. This interaction includes activities such as group and locator management, state retrieval, and discarding discovered but unavailable lookup services so they are eligible for re-discovery. The semantics of the methods of the `LookupDiscoveryRegistration` interface are defined in the section of this specification titled *The LookupDiscoveryRegistration Interface*.

The `groups` argument takes a `String` array, none of whose elements may be `null`. Although it is acceptable to input `null` (equivalent to `DiscoveryGroupManagement.ALL_GROUPS`) for the `groups` argument itself, if the argument contains one or more `null` elements, a `NullPointerException` is thrown. If the value is `null`, the lookup discovery service will attempt to discover all lookup services located within the multicast radius of the host on which the lookup discovery service is running. If an empty array (equivalent to `DiscoveryGroupManagement.NO_GROUPS`) is passed in, then no group discovery will be performed for the associated registration until the client, through the registration's `setGroups` or `addGroups` method, changes the contents of the managed set of groups to either a non-empty set of group names, or `null` (`DiscoveryGroupManagement.ALL_GROUPS`).

The `locators` argument takes an array of `LookupLocator` objects, none of whose elements may be `null`. If either the empty array or `null` is passed in as the `locators` argument, then no locator discovery will be performed for the associated registration until the client, through the registration's `addLocators` or `setLocators` method, changes the managed set of locators to a non-empty set of locators. Although it is acceptable to input `null` for the `locators` argument itself, if the argument contains one or more `null` elements, a `NullPointerException` is thrown.

If the `register` method is invoked with a set of group names and a set of locators in which either or both sets contain duplicate elements (where duplicate

locators are determined by `LookupLocator.equals()`, the invocation is equivalent to constructing this class with no duplicates in either set.

Upon discovery of a lookup service, through either group discovery or locator discovery, the lookup discovery service will send an event, referred to as a *discovered event*, to the listener associated with the registration produced by the call to `register`.

After initial discovery of a lookup service, the lookup discovery service will continue to monitor the group membership state reflected in the multicast announcements from that lookup service. Depending on the lookup service's current group membership, the lookup discovery service may send either a discovered event or an event referred to as a *discarded event*. The conditions under which either a discovered event or a discarded event will be sent are as follows:

- ◆ If the multicast announcements from an already discovered lookup service indicate that the lookup service is a member of a new group, a discovered event will be sent to the listener of each registration that has yet to receive a discovered event for that lookup service, but which has previously registered interest in the new group.
- ◆ If the multicast announcements from an already discovered lookup service indicate that the lookup service has changed its group membership in such a way that the lookup service is no longer of interest to one or more of the registrations that previously registered interest in the groups of that lookup service, a discarded event will be sent to the listener of each such registration. This type of discarded event is sometimes referred to as a *passive no-interest discarded event* ("passive" because the lookup discovery service, rather than the client, initiated the discard process).
- ◆ If the multicast announcements from an already discovered lookup service are no longer being received, a discarded event will be sent to the listener of each registration that previously registered interest in one or more of that lookup service's member groups. This type of discarded event is sometimes referred to as a *passive communication discarded event*.

It is important to note that when the lookup discovery service (passively) discards a lookup service, due to group membership changes (lost interest) or unavailability (communication failure), the discarded event will be sent to only the listeners of those registrations that have previously requested the affected lookup service be discovered through at least group discovery. That is, the listener of any registration that is interested in the affected lookup service through *only* locator discovery will not be sent either type of passive discarded event. This is because the semantics of the lookup discovery service assume that since the cli-

ent, through the registration request, expressed no interest in discovering the lookup service through its group membership, the client must also have no interest in any group-related changes in that lookup service's state.

A more detailed discussion of the event semantics of the lookup discovery service is presented in the section of this specification titled *Event Semantics*.

A valid parameter must be passed as the `listener` argument of the `register` method. If a `null` value is input to this argument, then a `NullPointerException` will be thrown and the registration fails.

Note that if an indefinite exception occurs while attempting to send a discovered or discarded event to a registration's listener, the lookup discovery service will continue to attempt to send the event until either the event is successfully delivered, or the client's lease on that registration expires. If an `UnknownEventException`, a bad object exception, or a bad invocation exception occurs while attempting to send a discovered or discarded event to a registration's listener, the lookup discovery service assumes the client is in an unknown, possibly corrupt state, and will cancel the lease on the registration and clear the registration from its managed set.

The state information maintained by the lookup discovery service includes the set of group names, locators, and listeners submitted by each client through each invocation of the `register` method, with duplicates eliminated. This state information contains no knowledge of the clients that register with the lookup discovery service. Thus, there is no requirement that a client identify itself during the registration process.

### LD.3.2 Event Semantics

For each registration created by the lookup discovery service, an event identifier will be generated that uniquely maps the registration to the listener as well as to the registration's managed set of groups and managed set of locators. This event identifier is returned as a part of the returned registration object, and is unique across all other active registrations with the lookup discovery service.

Whenever the lookup discovery service finds a lookup service matching the discovery criteria of one or more of its registrations, it sends an instance of `RemoteDiscoveryEvent` (a sub-class of `RemoteEvent`) to the listener corresponding to each such registration. The event sent to each listener will contain the appropriate event identifier.

Once an event signaling the discovery (by group or locator) of a desired lookup service has been sent, no other discovered events for that lookup service will be sent to a registration's listener until the lookup service is discarded (either *actively*, by the client through the registration, or *passively* by the lookup discov-

ery service) and then re-discovered. Note that more information about what it means for a lookup service to be discarded is presented in the sections of this specification titled, *Registration Semantics* and *Discarding Lookup Services*.

If, between the time a lookup service is discarded and the time it is re-discovered, a new registration is requested having parameters indicating interest in that lookup service, upon re-discovery of the lookup service an event will also be sent to that new registration's listener.

The sequence numbers for a given event identifier are *strictly increasing* (as defined in the *Jini™ Distributed Event Specification*), which means that when any two such successive events have sequence numbers differing by only a value of 1, then no events have been missed. On the other hand, when viewing the set of received events in order, if the difference between the sequence numbers of two successive events is greater than 1, then one or more events may or may not have been missed. For example, a difference greater than 1 could occur if the lookup discovery service crashes, even if no events are lost because of the crash. When two such successive events have sequence numbers whose difference is greater than 1, there is said to be a *gap* between the events.

When a gap occurs between events, the local state (on the client) related to the discovered lookup services may or may not fall "out of sync" with the corresponding remote state maintained by the lookup discovery service. For example, if the gap corresponds to a missed event representing the (initial) discovery of a targeted lookup service, the remote state will reflect this discovery, whereas the client's local state will not. To allow clients to identify and correct such a situation, each registration object provides a method which returns a set consisting of the proxies to the lookup services that have been discovered for that registration. With this information, the client can update its local state.

When requesting a registration with the lookup discovery service, a client may also supply (as a parameter to the `register` method) a reference to an object, wrapped in a `MarshaledObject`, referred to as a *handback*. When the lookup discovery service sends an event to a registration's listener, the event will also contain a reference to this handback object. The lookup discovery service will not change the handback object. That is, the handback object contained in the event sent by the lookup discovery service will be identical to the handback object registered by the client with the event mechanism.

The semantics of the object input to the handback argument are left to each client to define, although `null` may be input to this argument. The role of the handback object in the remote event mechanism is detailed in the *Jini™ Distributed Event Specification*.

### LD.3.3 Leasing Semantics

When a client registers with the lookup discovery service, it is effectively requesting a lease on the resources provided by that service. The initial duration of the lease granted to a client by the lookup discovery service will be less than or equal to the requested duration reflected in the value input to the `LeaseDuration` argument. That value must be positive, `Lease.FOREVER`, or `Lease.ANY`. If any other value is input to this argument, an `IllegalArgumentException` will be thrown. The client may obtain a reference to the `Lease` object granted by the lookup discovery service through the associated registration returned by the service (refer to the section of this specification titled, *The LookupDiscoveryRegistration Interface*).

---

## LD.4 Supporting Interfaces and Classes

**T**HE lookup discovery service depends on the `LookupDiscoveryRegistration` interface, as well as on the concrete classes `RemoteDiscoveryEvent` and `LookupUnmarshalException`.

### LD.4.1 The `LookupDiscoveryRegistration` Interface

When a client requests a registration with the lookup discovery service, an object that implements the `LookupDiscoveryRegistration` interface is returned. It is through this interface that the client manages the state of its registration with the lookup discovery service.

```
package net.jini.discovery;

public interface LookupDiscoveryRegistration {
    public EventRegistration getEventRegistration();
    public Lease getLease();
    public ServiceRegistrar[] getRegistrars()
                                throws LookupUnmarshalException,
                                    RemoteException;
    public String[] getGroups() throws RemoteException;
    public LookupLocator[] getLocators()
                                throws RemoteException;
    public void addGroups(String[] groups)
                                throws RemoteException;
    public void setGroups(String[] groups)
                                throws RemoteException;
    public void removeGroups(String[] groups)
                                throws RemoteException;
    public void addLocators(LookupLocator[] locators)
                                throws RemoteException;
    public void setLocators(LookupLocator[] locators)
```

```
        throws RemoteException;  
    public void removeLocators(LookupLocator[] locators)  
        throws RemoteException;  
    public void discard(ServiceRegistrar registrar)  
        throws RemoteException;  
}
```

As with the `LookupDiscoveryService` interface, the `LookupDiscoveryRegistration` interface is not a remote interface. Each implementation of the lookup discovery service exports proxy objects that implement this interface local to the client, using an implementation-specific protocol to communicate with the actual remote server. All of the proxy methods must obey normal Java RMI remote interface semantics except where explicitly noted. Two proxy objects are equal (using the `equals` method) if they are proxies for the same registration created by the same lookup discovery service.

The discovery facility of the lookup discovery service, together with its event mechanism, make up the set of resources clients register to use. Because the resources of the lookup discovery service are leased, access is granted for only a limited period of time, unless there is an active expression of continuing interest on the part of the client.

When a client uses the registration process to request that a lookup discovery service perform discovery of a set of desired lookup services, the client is also registered with the service's event mechanism. Because of this implicit registration with the event mechanism, the lookup discovery service "bundles" both resources under a single lease. When that lease expires, both discovery processing and event notifications will cease with respect to the registration that resulted from the client's request.

To facilitate lease management and event handling, the `LookupDiscoveryRegistration` interface defines methods which allow the client to retrieve its event registration information. Additional methods defined by this interface allow the client to retrieve references to the registration's currently discovered lookup services, as well as to modify the managed sets of groups and locators.

If the client's registration with the lookup discovery service has expired or been cancelled, then any invocation of a remote method defined in this interface will result in a `NoSuchObjectException`. That is, any method that communicates with the backend server of the lookup discovery service will throw a `NoSuchObjectException` if the registration on which the method is invoked no longer exists. It should be noted that if a client receives a `NoSuchObjectException` as a result of an invocation of such a method, although



the client can assume that the registration no longer exists, the client cannot assume that the lookup discovery service itself no longer exists.

Each remote method of this interface may throw a `RemoteException`. Typically, this exception occurs when there is a communication failure between the client and the lookup discovery service. Whenever this exception occurs as a result of the invocation of one of these methods, the method may or may not have completed its processing successfully.

### **LD.4.1.1 The Semantics**

The methods defined by this interface are organized into a set of accessor methods, a set of group mutator methods, a set of locator mutator methods, and the discard method. Through the accessor methods, various elements of a registration's state can be retrieved. The mutator methods provide a mechanism for changing the set of groups and locators to be discovered for the registration. Through the discard method, a particular lookup service may be made eligible for re-discovery.

#### **The Accessor Methods**

The `getEventRegistration` method returns an `EventRegistration` object that encapsulates the information needed by the client to identify a notification sent by the lookup discovery service to the registration's listener. This method is not remote and takes no arguments.

The `getLease` method returns the `Lease` object that controls a client's registration with the lookup discovery service. It is through the `Lease` object returned by this method that the client requests the renewal or cancellation of the registration with the lookup discovery service. This method is not remote and takes no arguments.

Note that the object returned by the `getEventRegistration` method also provides a `getLease` method. That method and the `getLease` method defined by the `LookupDiscoveryRegistration` interface both return the same `Lease` object. The `getLease` method defined here is provided as a convenience to avoid the indirection associated with the `getLease` method on the `EventRegistration` object, as well as to avoid the overhead of making two method calls.

The `getRegistrars` method returns a set of instances of the `ServiceRegistrar` interface. Each element in the set is a proxy to one of the lookup services that have already been discovered for the registration. Additionally, each element in the set will be unique with respect to all other elements in the set, as determined by the `equals` method provided by each element. The contents

of the set make up the current remote state of the set of lookup services discovered for the registration. This method returns a new array on each invocation.

This method can be used to maintain synchronization between the set of discovered lookup services making up a registration's local state on the client and the registration's corresponding remote state maintained by the lookup discovery service. The local state can become un-synchronized with the remote state when a gap occurs in the events received by the registration's listener.

According to the event semantics of the lookup discovery service, if there is no gap between two sequence numbers, no events have been missed and the states remain synchronized with each other; if there is a gap, events may or may not have been missed. Thus, upon finding gaps in the sequence of events, the client can invoke this method and use the returned information to synchronize the local state with the remote state.

In order to construct its return set, the `getRegistrars` method retrieves from the lookup discovery service the set of lookup service proxies making up the registration's current remote state. When the lookup discovery service sends the requested set of proxies, the set is sent as a set of marshalled instances of the `ServiceRegistrar` interface. The lookup discovery service individually marshals each proxy in the set that it sends because if it were not to do so, *any* deserialization failure on the set would result in an `IOException`, and failure would be declared for the whole deserialization process, not just an individual element. This would mean that all elements of the set sent by the lookup discovery service—even those that were successfully deserialized—would be unavailable to the client. Individually marshalling each element in the set minimizes the “all or nothing” aspect of the deserialization process, allowing the client to recover those proxies that can be successfully unmarshalled and to proceed with processing that might not be possible otherwise.

When constructing the return set, this method attempts to unmarshal each element of the set of marshalled proxy objects sent by the lookup discovery service. When failure occurs while attempting to unmarshal any of those elements, this method throws an exception of type `LookupUnmarshalException` (described later). It is through the contents of that exception that the client can recover any available proxies and perform error handling related to the unavailable proxies. The contents of the `LookupUnmarshalException` provide the client with the following useful information:

- ◆ The knowledge that a problem has occurred while unmarshalling at least one of the elements making up the remote state of the registration's discovered lookup services

- ◆ The set of proxy objects that were successfully unmarshalled by the `getRegistrars` method
- ◆ The set of marshalled proxy objects that could not be unmarshalled by the `getRegistrars` method
- ◆ The set of exceptions corresponding to each failed attempt at unmarshalling

The type of exception that occurs when attempting to unmarshal an element of the set sent by the lookup discovery service is typically an `IOException` or a `ClassNotFoundException` (usually the more common of the two). A `ClassNotFoundException` occurs whenever a remote object on which the marshalled proxy depends cannot be retrieved and loaded—usually because the codebase of one of the object's classes or interfaces is currently 'down'. To address this situation, the client may wish to proceed with its processing using the successfully unmarshalled proxies, and attempt to unmarshal the unavailable proxies (or re-invoke this method) at some later time.

If the `getRegistrars` method returns successfully without throwing a `LookupUnmarshalException`, the client is guaranteed that all marshalled proxies belonging to the set sent by the lookup discovery service have each been successfully unmarshalled; the client then has a snapshot—relative to the point in time when this method is invoked—of the remote state of the lookup services discovered for the associated registration.

The `getGroups` method returns an array consisting of the group names from the registration's managed set; that is, the names of the groups the lookup discovery service is currently configured to discover for the associated registration. If the managed set of groups is empty, this method returns the empty array. If there is no managed set of groups associated with the registration (that is, the lookup discovery service is configured to discover `DiscoveryGroupManagement.ALL_GROUPS` for the registration), then `null` is returned.

The `getLocators` method returns an array consisting of the `LookupLocator` objects from the registration's managed set; that is, the locators of the specific lookup services the lookup discovery service is currently configured to discover for the associated registration. If the managed set of locators is empty, this method returns the empty array.

## The Group Mutator Methods

With respect to a particular registration, the groups to be discovered may be modified using the methods described in this section. In each case, a set of groups is represented as a `String` array, none of whose elements may be `null`. If any set of groups input to one of these methods contains one or more `null` elements, a

`NullPointerException` is thrown. The empty set is denoted by the empty array (`DiscoveryGroupManagement.NO_GROUPS`), and “no set” is indicated by `null` (`DiscoveryGroupManagement.ALL_GROUPS`). No set indicates that all lookup services within the multicast radius should be discovered, regardless of group membership. Invoking any of these methods with an input set of groups that contains duplicate names is equivalent to performing the invocation with the duplicate group names removed from the input set.

The `addGroups` method adds a set of group names to the registration’s managed set. This method takes one argument: a `String` array consisting of the set of group names with which to augment the registration’s managed set.

If the registration has no current managed set of groups to augment, this method throws an `UnsupportedOperationException`. If the parameter value is `null`, this method throws a `NullPointerException`. If the parameter value is the empty array, then the registration’s managed set of groups will not change.

The `setGroups` method replaces all of the group names in the registration’s managed set with names from a new set. This method takes one argument: a `String` array consisting of the set of group names with which to replace the current names in the registration’s managed set.

If `null` (`DiscoveryGroupManagement.ALL_GROUPS`) is passed to `setGroups`, the lookup discovery service will attempt to discover any undiscovered lookup services located within range of the lookup discovery service, regardless of group membership.

If the empty set (`DiscoveryGroupManagement.NO_GROUPS`) is passed to `setGroups`, then group discovery will be halted until the registration’s managed set of groups is changed—through a subsequent call to this method or to `addGroups`—to a set that is either a non-empty set of group names or `null` (`DiscoveryGroupManagement.ALL_GROUPS`).

The `removeGroups` method deletes a set of group names from the registration’s managed set. This method takes one argument: a `String` array containing the set of group names to remove from the registration’s managed set.

If the registration has no current managed set of groups from which to remove elements, this method throws an `UnsupportedOperationException`. If `null` is input, this method throws a `NullPointerException`. For any element in the input set that equals no element in the managed set, `removeGroups` takes no action with respect to that element. If the empty array is input, then the registration’s managed set of groups will not change.

Once a new group name has been placed in the registration’s managed set as a result of an invocation of either `addGroups` or `setGroups`, if there are lookup services belonging to that group that have already been discovered, no event will be sent to the registration’s listener for those particular lookup services; although

attempts to discover any undiscovered lookup services belonging to that group will continue to be made on behalf of the registration.

Any already discovered lookup service that is a member of one or more of the groups removed from the registration's managed set as a result of an invocation of either `setGroups` or `removeGroups` will be discarded and will no longer be eligible for discovery (for that registration); but only if that lookup service satisfies both of the following conditions:

- ◆ the lookup service is not a member of any group in the registration's new managed set resulting from the invocation of `setGroups` or `removeGroups`, and
- ◆ with respect to the registration, the lookup service is not currently eligible for discovery through other means (such as locator discovery); that is, the lookup service does not correspond to any element in the registration's managed set of locators.

## The Locator Mutator Methods

With respect to a particular registration, the set of locators to discover may be modified using the methods described in this section. In each case, a set of locators is represented as an array of `LookupLocator` objects, none of whose elements may be `null`. If any set of locators input to one of these methods contains one of more `null` elements, a `NullPointerException` is thrown. Invoking any of these methods with a set of locators that contains duplicate locators (as determined by `LookupLocator.equals`) is equivalent to performing the invocation with the duplicates removed from the input set.

The `addLocators` method adds a set of `LookupLocator` objects to the registration's managed set. This method takes one argument: an array consisting of the set of locators with which to augment the registration's managed set.

If `null` is passed to `addLocators`, a `NullPointerException` will be thrown. If the parameter value is the empty array, the registration's managed set of locators will not change.

The `setLocators` method replaces all of the locators in the registration's managed set with `LookupLocator` objects from a new set. This method takes one argument: an array consisting of the set of locators with which to replace the current locators in the registration's managed set.

If `null` is passed to `setLocators`, a `NullPointerException` will be thrown.

If the empty set is passed to `setLocators`, then locator discovery will be halted until the registration's managed set of locators is changed—through a sub-

sequent call to this method or to `addLocators`—to a set that is non-`null` and non-empty.

The `removeLocators` method deletes a set of `LookupLocator` objects from the registration's managed set. This method takes one argument: an array containing the set of `LookupLocator` objects to remove from the registration's managed set.

If `null` is passed to `removeLocators`, a `NullPointerException` will be thrown. If any element of the set of locators to remove is not contained in the registration's managed set, `removeLocators` takes no action with respect to that element. If the parameter value is the empty array, the managed set of locators will not change.

Whenever a new locator is placed in the managed set as a result of an invocation of one of the locator mutator methods and that new locator equals none of the previously discovered locators (across all registrations), the lookup discovery service will attempt unicast discovery of the lookup service associated with the new locator.

If locator discovery is attempted for a registration, such discovery attempts will be repeated until one of the following events occurs:

- ◆ The lookup service is discovered
- ◆ The client's lease on the registration expires
- ◆ The client explicitly removes the locator from the registration's managed set

Upon discovery of the lookup service corresponding to the new locator, or upon finding a match between the new locator and a previously discovered lookup service, a discovered event will be sent to the registration's listener.

Any already discovered lookup service corresponding to a locator that is removed from the registration's managed set as a result of an invocation of either `setLocators` or `removeLocators` will be discarded and will no longer be eligible for discovery; but only if it is not currently eligible for discovery through other means (such as group discovery). That is, only if the lookup service is not also a member of one or more of the groups in the registration's managed set of groups.

### Discarding Lookup Services

When the lookup discovery service removes an already discovered lookup service from a registration's managed set of lookup services, the lookup service is said to be *discarded*.

There are a number of situations where the lookup discovery service will discard a lookup service:

- ◆ In response to a discard request resulting from an invocation of a registration's discard method
- ◆ In response to a declaration—via an invocation of one of the mutator methods on a registration—that there is no longer any interest in one or more of the registration's already discovered lookup services
- ◆ In response to the determination that the multicast announcements from an already discovered lookup service indicate that the lookup service has changed its group membership in such a way that the lookup service is no longer of interest to one or more of the registrations that previously registered interest in the groups of that lookup service
- ◆ In response to the determination that the multicast announcements from an already discovered lookup service are no longer being received

For each of these cases, whenever the lookup discovery service discards a lookup service, it will send an event to the registration's listener to notify it that the lookup service has been discarded.

The discard method provides a mechanism for registered clients to inform the lookup discovery service of the existence of an unavailable—or *unreachable*—lookup service, and to request that the lookup discovery service discard that lookup service, and make it eligible for re-discovery.

The discard method takes a single argument: the proxy to the lookup service to discard. This method takes no action if the parameter to this method equals none of the proxies reflected in the managed set (using proxy equality as defined in the *Jini™ Lookup Service Specification*). If null is passed to discard, a `NullPointerException` will be thrown.

Note that once a lookup service is discovered, there is no longer any need to perform discovery processing with respect to that lookup service. This means that if a lookup service becomes unreachable after it has been discovered, until that lookup service is discarded, the lookup discovery service will not know when the lookup service becomes reachable again.

Although the lookup discovery service will monitor the multicast announcements for indications of unavailability, it will discard only those unreachable lookup services for which there is at least one registration that requested discovery through at least group discovery. That is, if the lookup discovery service determines that a previously discovered lookup service has become unreachable, but there is no registration wishing to discover that lookup service by group discovery, then the lookup discovery service will not discard the lookup service.

Thus, whenever a client determines that a previously discovered lookup service has become unreachable, it should not rely on the lookup discovery service to discard the lookup service. Instead, the client should inform the lookup discovery service—through the invocation of the registration's discard method—that the

previously discovered lookup service is no longer available, and that attempts should be made to re-discover that lookup service for the registration. Typically, a client determines that a lookup service is unavailable when the client attempts to use the lookup service but receives an indefinite exception, a bad object exception, or a bad invocation exception as a result of the attempt.

Note that the lookup discovery service may be acting on behalf of numerous clients having access to the same lookup service. If that lookup service becomes unavailable, many of those clients may invoke `discard` between the time the lookup service becomes unavailable and the time it is re-discovered. Upon the first invocation of `discard`, the lookup discovery service will re-initiate discovery of the relevant lookup service for the registration of the client that made the invocation. For all other invocations made prior to re-discovery, the registrations through which the invocation is made are sent a discarded event, and added to the list of registrations that will be notified when re-discovery of the lookup service does occur. That is, upon re-discovery of the lookup service, only those registrations through which the `discard` method was invoked will be notified.

Upon successful completion of the `discard` method, the proxy requested to be discarded is guaranteed to have been removed from the managed set of the registration through which the invocation was made. No such guarantee is made with respect to when the discarded event is sent to each such registration's listener. That is, the event notifying the listeners that the lookup service has been discarded may or may not be sent asynchronously.

## LD.4.2 The RemoteDiscoveryEvent Class

When the lookup discovery service discovers or discards a lookup service matching the criteria established through one of its registrations, the lookup discovery service sends an instance of the `RemoteDiscoveryEvent` class to the `RemoteEventListener` implemented by the client and registered with the lookup discovery service.

```
package net.jini.discovery;

public class RemoteDiscoveryEvent extends RemoteEvent {
    public RemoteDiscoveryEvent(Object source,
                                long eventID,
                                long seqNum,
                                MarshallableObject handback,
                                boolean discarded,
                                Map groups)
```



```

throws IOException {...}

    public boolean isDiscarded() {...}
    public ServiceRegistrar[] getRegistrars()
        throws LookupUnmarshalException {...}
    public Map getGroups() {...}
}

```

The `RemoteDiscoveryEvent` class provides an encapsulation of event information that the lookup discovery service uses to notify a registration of the occurrence of an event involving one or more `ServiceRegistrar` objects (lookup services) in which the registration has registered interest. The lookup discovery service passes an instance of this class to the registration's discovery listener when one of the following events occurs:

- ◆ Each lookup service referenced in the event has been discovered for the first time, or re-discovered after having been discarded.
- ◆ Each lookup service referenced in the event has been either actively or passively discarded.

`RemoteDiscoveryEvent` is a subclass of `RemoteEvent`, adding the following additional items of abstract state:

- ◆ A `boolean` indicating whether the lookup services referenced by the event have been discovered or discarded
- ◆ A set of marshalled instances of the `ServiceRegistrar` interface, having the characteristic that when each element is unmarshalled, the result is a proxy to one of the discovered or discarded lookup services referenced by the event
- ◆ A `Map` instance in which the elements of the map's key set are the instances of `ServiceID` that correspond to each lookup service reference returned in the event, and the map's value set contains the corresponding member groups of each lookup service reference

Methods are defined through which this additional state may be retrieved upon receipt of an instance of this class.

Clients need to know not only when a targeted lookup service has been discovered, but also when it has been discarded. The lookup discovery service uses an instance of `RemoteDiscoveryEvent` to notify a registration when either of these events occurs, as indicated by the value of the `boolean` state variable. When

the value of that variable is `true`, the event is referred to as a *discarded event*; when `false`, it is referred to as a *discovered event*.

#### LD.4.2.1 The Semantics

The constructor of the `RemoteDiscoveryEvent` class takes the following parameters as input:

- ◆ A reference to the lookup discovery service that generated the event
- ◆ The event identifier that maps a particular registration to both its listener and its targeted groups and locators
- ◆ The sequence number of the event being constructed
- ◆ The client-defined handback (which may be `null`)
- ◆ A flag indicating whether the event being constructed is a discovered event or a discarded event
- ◆ A Map whose key set contains the proxies to newly discovered or discarded lookup service(s) the event is to reference, and whose value set contains the corresponding member groups of each lookup service

If the `groups` parameter is empty, the constructor will throw an `IllegalArgumentException`. If `null` is input to the `groups` parameter, the constructor will throw a `NullPointerException`. If none of the proxies referenced in the `groups` parameter can be successfully serialized, the constructor will throw an `IOException`.

The `isDiscarded` method returns a `boolean` that indicates whether the event is a discovered event or a discarded event. If the event is a discovered event, then this method returns `false`. If the event is a discarded event, `true` is returned.

The `getRegistrars` method returns an array consisting of instances of the `ServiceRegistrar` interface. Each element in the returned set is a proxy to one of the newly discovered or discarded lookup services that caused an instance of `RemoteDiscoveryEvent` to be sent. Additionally, each element in the returned set will be unique with respect to all other elements in the set, as determined by the `equals` method provided by each element. This method does not make a remote call. With respect to multiple invocations of this method, each invocation will return a new array.

When the lookup discovery service sends an instance of `RemoteDiscoveryEvent` to the listener of a client's registration, the set of lookup service proxies contained in the event consists of marshalled instances of the `ServiceRegistrar` interface. The lookup discovery service individually marshals each proxy associ-

ated with the event because if it were not to do so, *any* deserialization failure on the set would result in an `IOException`, and failure would be declared for the whole deserialization process, not just an individual element. This would mean that all elements of the set sent in the event—even those that can be successfully deserialized—would be unavailable to the client through this method. Just as with the `getRegistrars` method defined by the `LookupDiscoveryRegistration` interface, individually marshalling each element in the set minimizes the “all or nothing” aspect of the deserialization process, allowing the client to recover those proxies that can be successfully unmarshalled and to proceed with processing that might not be possible otherwise.

When constructing the return set, this method attempts to unmarshal each element of the set of marshalled proxy objects contained in the event. When failure occurs while attempting to unmarshal any of the elements of that set, this method throws an exception of type `LookupUnmarshalException`. It is through the contents of this exception that the client can recover any available proxies and perform error handling with respect to the unavailable proxies.

If the `getRegistrars` method returns successfully without throwing a `LookupUnmarshalException`, the client is guaranteed that all marshalled proxies sent in the event have each been successfully unmarshalled during that particular invocation. Furthermore, after the first such successful invocation, no more unmarshalling attempts will be made (because such attempts are no longer necessary), and all future invocations of this method are guaranteed to return an array with contents identical to the contents of the array returned by the first successful invocation.

Note that an array, rather than a single proxy, is returned by the `getRegistrars` method so that implementations of the lookup discovery service can choose to “batch” the information sent to a registration. With respect to discoveries, batching the information may be particularly useful when a client first registers with the lookup discovery service.

Upon initial registration, multiple lookup services are typically found over a short period of time, providing the lookup discovery service with the opportunity to send all of the initially discovered lookup services in only one event. Afterwards, as so-called “late joiner” lookup services are found sporadically, the lookup discovery service may send events referencing only one lookup service.

Note that the event sequence numbers, as defined earlier in this specification in the section titled *Event Semantics*, are strictly increasing—even when the information is batched.

The `getGroups` method returns a `Map` in which the elements of the map’s key set are the instances of `ServiceID` that correspond to each lookup service for which the event was constructed and sent. Each element of the returned map’s value set is a `String` array containing the names of the member groups of the

associated lookup service whose `ServiceID` equals to the corresponding key. This method does not make a remote call. On each invocation of this method, the same `Map` object is returned; that is, a copy is not made.

The `Map` returned by the `getGroups` method is keyed by the `ServiceID` of each lookup service in the event, rather than by the proxy of each lookup service to avoid the deserialization issues addressed by the `getRegistrars` method. Thus, client's wishing to retrieve the set of member groups corresponding to any element of the array returned by the `getRegistrars` method, must use the `ServiceID` of the desired element from that array as the key to the `get` method of the `Map` returned by this method and then cast to `String[]`.

### Serialized Forms

Class	serialVersionUID	Serialized Fields
<code>RemoteDiscoveryEvent</code>	-9171289945014585248L	<code>boolean</code> discarded <code>ArrayList</code> marshalledRegs <code>ServiceRegistrar[]</code> regs <code>Map</code> groups

### LD.4.3 The `LookupUnmarshalException` Class

Recall that when unmarshalling an instance of `MarshaledObject`, one of the following checked exceptions is possible:

- ◆ An `IOException`, which can occur while deserializing the object from its internal representation
- ◆ A `ClassNotFoundException`, which can occur if, while deserializing the object from its internal representation, either the class file of the object cannot be found, or the class file of an interface or class referenced by the object being deserialized cannot be found. Typically, a `ClassNotFoundException` occurs when the codebase from which to retrieve the needed class file is not currently available

The `LookupUnmarshalException` class provides a mechanism that clients of the lookup discovery service may use for efficient handling of the exceptions that may occur when unmarshalling elements of a set of marshalled instances of the `ServiceRegistrar` interface. When elements in such a set are unmarshalled, the `LookupUnmarshalException` class may be used to collect and report pertinent

information; information generated when failure occurs during the unmarshalling process.

```
package net.jini.discovery;

public class LookupUnmarshalException extends Exception {
    public LookupUnmarshalException
        (ServiceRegistrar[] unmarshalledRegs,
         MarshalledObject[] stillMarshalledRegs,
         Throwable[] exceptions) {...}
    public ServiceRegistrar[] getUnmarshalledRegs() {...}
    public MarshalledObject[] getStillMarshalledRegs() {...}
    public Throwable[] getExceptions() {...}
    public String getMessage() {...}
}
```

The `LookupUnmarshalException` class is a subclass of `Exception`, adding the following additional items of abstract state:

- ◆ A set of `ServiceRegistrar` instances in which each element is the result of a successful unmarshalling attempt
- ◆ A set of marshalled instances of `ServiceRegistrar` in which each element is the result of an unsuccessful unmarshalling attempt
- ◆ A set of exceptions (`IOException`, `ClassNotFoundException`, or some unchecked exception) in which each element corresponds to one of the unsuccessful unmarshalling attempts
- ◆ A `String` describing the nature of the exception

Note that because a `LookupUnmarshalException` should be thrown only as a result of a failure while unmarshalling a set of marshalled instances of `ServiceRegistrar`, no mechanism is provided to modify the initial value of the descriptive `String`.

Thus, when exceptional conditions occur while unmarshalling a set of marshalled instances of `ServiceRegistrar`, the `LookupUnmarshalException` class can be used not only to indicate that an exceptional condition has occurred, but also to provide information that can be used to perform error handling activities such as:

- ◆ Determining if it is feasible to continue with processing
- ◆ Reporting errors

- ◆ Attempting recovery
- ◆ Performing debug activities

### LD.4.3.1 The Semantics

The constructor of the `LookupUnmarshalException` class takes the following parameters as input:

- ◆ An array containing the set of instances of `ServiceRegistrar` that were successfully unmarshalled
- ◆ An array containing the set of marshalled `ServiceRegistrar` instances that could not be unmarshalled
- ◆ An array containing the set of exceptions that occurred during the unmarshalling process.

Each element in the `exceptions` parameter should be an instance of `IOException`, `ClassNotFoundException`, or some unchecked exception. Furthermore, there is a one-to-one correspondence between each element in the `exceptions` parameter and each element in the `stillMarshalledRegs` parameter. That is, the element of the `exceptions` parameter corresponding to index *i* should be an instance of the exception that occurred while attempting to unmarshal the element at index *i* of the `stillMarshalledRegs` parameter.

If the number of elements in the `exceptions` parameter does not equal the number of elements in the `stillMarshalledRegs` parameter, the constructor will throw an `IllegalArgumentException`.

The `getUnmarshalledRegs` method is an accessor method that returns an array consisting of instances of `ServiceRegistrar`, where each element of the array corresponds to a successfully unmarshalled object. Note that the same array is returned on each invocation of this method; that is, a copy is not made.

The `getStillMarshalledRegs` method is an accessor method that returns an array consisting of instances of `MarshaledObject`, where each element of the array is a marshalled instance of the `ServiceRegistrar` interface, and corresponds to an object that could not be successfully unmarshalled. Note that the same array is returned on each invocation of this method; that is, a copy is not made.

The `getExceptions` method is an accessor method that returns an array consisting of instances of `Throwable`, where each element of the array corresponds to one of the exceptions that occurred during the unmarshalling process. Each element in the return set is an instance of `IOException`, `ClassNotFoundException`,

or some unchecked exception. Additionally, there should be a one-to-one correspondence between each element in the array returned by this method and the array returned by the `getStillMarshaledRegs` method. Note that the same array is returned on each invocation of this method; that is, a copy is not made.

The `getMessage` method is an accessor method that returns a `String` describing the nature of the exception. Note that the same `String` is returned on each invocation of this method; that is, a copy is not made.

Serialized Forms

Class	serialVersionUID	Serialized Fields
LookupUnmarshalException	2956893184719950537L	ServiceRegistrar[] unmarshalledRegs MarshaledObject[] stillMarshaledRegs Throwable[] exceptions String descStr





# LR

---

# Jini™ Lease Renewal Service Specification

## LR.1 Introduction

**L**EASING is a key concept in the Jini™ architecture; in general, Jini technology-enabled services (*Jini services*) only grant access to a resource for as long as the clients of those Jini services actively express interest in the resource being maintained. This pattern is in contrast to many other systems where access to a resource is granted until the client explicitly releases the resource. Using a leasing model generally makes a distributed system more robust by allowing stale information and services to be cleaned up, but it also places additional requirements on clients and services.

A client of a leased service may run into difficulties if that service deactivates. Unless the client ensures that some other process renews the client's leases while it is inactive, or that the client is activated before its leases begin to expire, the client will lose access to the resources it has acquired. This loss can be particularly dramatic in the case of lookup service registrations. A service's registration with a lookup service is lease—if the service deactivates (maybe in order to conserve computational resources on its host) and it does not take appropriate steps, its registrations with lookup services will expire, and before long it will be inaccessible. If that service only becomes active when clients invoke its methods, it may never become active again, because at this point new clients may not be able to find it.

The need to renew leases creates a constant load on clients, servers, and the network. Although batching lease renewals can help (see the *Jini™ Distributed Leasing Specification*), a given client is unlikely to have very many leases granted by any one service at any given time, thus reducing the opportunities for meaningful batching.

This additional load may be an especially great burden on clients that always have the ability to access the network, but cannot be continuously connected. A cell phone always has the ability to connect; however, being connected all the time will drain its batteries and accumulate airtime charges. One or two leases may not pose a problem, but a large number of leases could force the phone to be on the network all the time.

A lease renewal service can help mitigate these problems. Services that wish to become inactive can pass the responsibility for renewing the leases they have been granted to a renewal service. Those services can then deactivate without risk of losing access to resources it has acquired. Clients that have continuous access to the network but cannot be continuously connected, such as the cell phone described previously, can also register with a renewal service that can be continuously connected. The renewal service will renew the client's leases, allowing the client to remain disconnected most of the time. Lastly, if multiple clients pass their leases to a given renewal service, more opportunities for batching renewals will be created.

Like other Jini services, the lease renewal service will only grant its services for a limited period of time without an active expression of continuing interest. In order to break the recursive cycle that would otherwise result, the renewal service provides an optional event that is triggered before the leases that it grants expire. This event gives activatable processes that have deactivated the opportunity to wake up and renew their lease with the renewal service. Although it may seem odd for the lease renewal service to lease its resources, it is very important that it does so. If it did not, then the lease renewal service could be used to subvert the leasing model.

A renewal service is likely to grant longer leases than other Jini services. In some cases, the lease may be so long that the client will not need to worry about renewing the lease at all. In other cases, the lease may be long enough that a client that deactivates will rarely need to reactivate for the sole purpose of renewing its lease with the renewal service. In any case, the leases that the renewal service grants are likely to be sufficiently long such that the actual renewal calls do not place a significant additional load on the client, the renewal service, or the network.

### **LR.1.0.1 Goals & Requirements**

The requirements of the set of classes and interfaces in this specification are:

- ◆ To provide a service for renewing leases

- ◆ To provide this service in such a way that it can be used by activatable processes that deactivate
- ◆ To provide this service in a way that does not overly weaken the leasing model

The goals of this specification are:

- ◆ To describe the lease renewal service
- ◆ To provide guidance in the use and deployment of the lease renewal service

### LR.1.1 Other Types

The types defined in the specification of the `LeaseRenewalService` interface are in the `net.jini.lease` package. The following object types may be referenced in this chapter. Whenever referenced, these object types will be referenced in unqualified form:

```
java.io.IOException  
java.rmi.MarshalledObject  
java.rmi.RemoteException  
java.rmi.NoSuchObjectException  
net.jini.core.lease.Lease  
net.jini.core.lease.UnknownLeaseException  
net.jini.core.event.RemoteEvent  
net.jini.core.event.RemoteEventListener  
net.jini.core.event.EventRegistration
```



---

## LS.2 The Interface

**L**easeRenewalService (in the `net.jini.lease` package) defines the interface to the renewal service. The interface is not a remote interface; each implementation of the renewal service exports proxy objects that implement the LeaseRenewalService interface local to the client, and use an implementation-specific protocol to communicate with the actual remote server. All of the proxy methods obey normal RMI remote interface semantics. Two proxy objects are equal (using the `equals` method) if they are proxies for the same renewal service. All of the methods of LeaseRenewalService throw `RemoteException` and only require the default serialization semantics. Therefore, LeaseRenewalService can be implemented directly using RMI.

```
package net.jini.lease;

public interface LeaseRenewalService {
    public LeaseRenewalSet createLeaseRenewalSet(
        long leaseDuration)
        throws RemoteException;
}
```

Clients of the renewal service organize the leases they wish to have renewed into *lease renewal sets* (or *sets*, for short). A method is provided by the LeaseRenewalService interface to create these sets. These sets are then populated by methods on the sets themselves. Two leases in the same set need not be granted by the same process or have the same expiration time; in addition, they can be added or removed from the set independently.

Every method invocation on a renewal service (whether the invocation is directly on the service, or indirectly on a set the service has created) is atomic with respect other invocations.

The term *client lease* is used to refer to a lease that has been placed into a renewal set. Client leases are distinct from the leases that the renewal service grants on renewal sets it has created.

In general, there will be times when an implementation needs to pass one client lease as an argument to a method call on a second client lease. There is a secu-

ality risk in doing so, because such actions can let the second client lease “capture” the first. Implementations may want to verify that their clients can be trusted not to place leases in the set that would take such actions. Another alternative is to only pass one Lease object to another if they trust each other. Depending on the environment, conservative tests for such trust could include: ensuring the codebases of both leases are constructed from the same set of URLs, or that all of the URLs come from a common set of hosts or host/port pairs.

Each client lease has two expiration related times associated with it: the *desired expiration* time for the lease, and the *actual expiration* time granted when the lease is created or last renewed. The desired expiration represents when the client would like the lease to expire. The actual expiration represents when the lease is going to expire if it is not renewed. Both time values are absolute times, not relative time durations. When a client lease’s desired expiration arrives, the lease will be removed from the set without further client intervention.

Each client lease also has two other associated attributes: a desired *renewal duration* and a *remaining desired duration*. The desired renewal duration is specified by the client (directly or indirectly) when the lease is added to the set. This duration must normally be a positive number, however, it may be Lease.ANY if the lease’s desired expiration is Lease.FOREVER. The remaining desired duration is always the desired expiration less the current time.

Each time a client lease is renewed, the renewal service will ask for an extension equal to the lease’s renewal duration if the renewal duration is:

- ◆ Lease.ANY, or
- ◆ less than the remaining desired duration,

otherwise, it will ask for an extension equal to the lease’s remaining desired duration.

If a lease’s actual expiration is later than the lease’s desired expiration, the renewal service will not renew the lease; the lease will remain in the set until its desired expiration is reached, the set is destroyed, or it is removed by the client.

Each set is leased from the renewal service. If the lease on a set expires or is cancelled, the renewal service will destroy the set and take no further action with regard to the client leases in the set. Each lease renewal set has associated with it an expiration warning event that occurs at a client-specified time before the lease on the set expires. Clients can register for warning events using methods provided by the set. A registration for warning events does not have its own lease, but instead is covered by the same lease under which the set was granted.

The term *definite exception* is used to refer to an exception that could be thrown by an operation on a client lease (such as a remote method call) that would

be indicative of a permanent failure of the client lease. For purposes of this document, all bad object exceptions, bad invocation exceptions, and `LeaseExceptions` are considered to be definite exceptions.

Each lease renewal set has associated with it a renewal failure event that will occur in either of two cases: if any client lease in the set reaches its actual expiration before its desired expiration is reached, or if the renewal service attempts to renew a client lease and gets a definite exception. Clients can register for failure events using methods provided by the set. A registration for failure event does not have its own lease, but instead is covered by the same lease under which the set was granted.

Once placed in a set, a client lease will stay there until one or more of the following occurs:

- ◆ The lease on the set itself expires or is cancelled, causing destruction of the set
- ◆ The client lease is removed by the client
- ◆ The client lease's actual or desired expiration is reached
- ◆ A renewal attempt on the client lease results in a definite exception

Each client lease in a set will be renewed as long as it is in the set. If a renewal call throws an indefinite exception, the renewal service should retry the lease renewal until the lease would otherwise be removed from the set. The renewal service will never cancel a client lease. The preferred method of cancelling a client lease is for the client to first remove the lease from the set and then call `cancel` on it. It is also permissible for the client to cancel the lease without first removing the lease from the set, although this is likely to result in additional network traffic.

The client creates a set by calling the `createLeaseRenewalSet` method of a `LeaseRenewalService`. The `leaseDuration` argument specifies how long (in milliseconds) the client wants the set's initial lease duration to be. The duration initially granted for the set's lease will be equal to or shorter than this request; it will not be longer. The value of the `leaseDuration` argument must be positive, `Lease.FOREVER`, or `Lease.ANY`; otherwise an `IllegalArgumentException` must be thrown. Two calls to the `createLeaseRenewalSet` method should never return objects that are equal. The set's lease is obtained through a method provided by the set.

`LeaseRenewalSet` defines the interface to the sets created by the lease renewal service. This interface is not a remote interface. Each implementation of the renewal service exports proxy objects that implement the `LeaseRenewalSet` interface local to the client and use an implementation-specific protocol to communicate with the actual remote server. All of the proxy methods obey normal

RMI remote interface semantics except where explicitly noted. The proxy objects for two sets are equal (using the `equals` method) if they are proxies for the same set created by the same renewal service. Any method that communicates with the remote server should throw a `NoSuchObjectException` if the set no longer exists. If a client receives a `NoSuchObjectException` from one of the operations on a lease renewal set, the client can infer that the set has been destroyed; however, it should *not* infer that the renewal service has been destroyed.

```
package net.jini.lease;

public interface LeaseRenewalSet {
    final public static long RENEWAL_FAILURE_EVENT_ID = 0;
    final public static long EXPIRATION_WARNING_EVENT_ID = 1;

    public void renewFor(Lease leaseToRenew,
                        long desiredDuration,
                        long renewDuration)
        throws RemoteException;

    public void renewFor(Lease leaseToRenew,
                        long desiredDuration)
        throws RemoteException;

    public EventRegistration setExpirationWarningListener(
        RemoteEventListener listener,
        long minWarning,
        MarshalledObject handback)
        throws RemoteException;

    public void clearExpirationWarningListener()
        throws RemoteException;

    public EventRegistration setRenewalFailureListener(
        RemoteEventListener listener,
        MarshalledObject handback)
        throws RemoteException;

    public void clearRenewalFailureListener()
        throws RemoteException;

    public Lease remove(Lease leaseToRemove)
```



```
        throws RemoteException;

    public Lease[] getLeases()
        throws LeaseUnmarshalException, RemoteException;

    public Lease getRenewalSetLease();
}
```

Leases can be added to the set through the `renewFor` methods. There are two forms of this method, a three-argument form and a two-argument form. The three-argument form will be described first. The `leaseToRenew` argument specifies the lease to be renewed. An `IllegalArgumentException` must be thrown if the lease was granted by the renewal service itself. An `IllegalArgumentException` must also be thrown if the lease is currently a member of another set allocated by the same renewal service. If `leaseToRenew` is `null`, a `NullPointerException` must be thrown.

The `desiredDuration` parameter is the number of milliseconds the client would like the lease to remain in the set. It is used to calculate the lease's initial desired expiration by adding `desiredDuration` to the current time (as viewed by the service). If this causes an overflow, a desired expiration of `Long.MAX_VALUE` will be used. Unlike a lease duration, the desired duration is unilaterally specified by the client, not negotiated between the client and the service. Note, a negative value for `desiredDuration` (including `Lease.ANY`) will result in a desired expiration that is in the past, causing the client lease to be dropped from the set; this action will neither result in an exception or an event.

If the actual expiration time of the client lease being added to the set is less than the current time (as viewed by the renewal service) and the current time is less than the client lease's desired expiration time, the method will return normally. However, the client lease will be dropped from the set and a renewal failure event will be generated.

A `desiredDuration` of `Long.MAX_VALUE` does not imply that the lease will remain in the set forever. The lease will be ejected from the set if: the set is destroyed, the lease itself expires, the lease is removed from the set, or the renewal service makes a renewal attempt on the lease that results in a definite exception.

The `renewDuration` is the initial renewal duration to associate with the client lease (in milliseconds). If `desiredDuration` is exactly `Long.MAX_VALUE`, the `renewDuration` may be any positive number or `Lease.ANY`; otherwise it must be a positive number. If these requirements are not met, the renewal service must throw an `IllegalArgumentException`.

Calling `renewFor` with a lease that is equivalent to a client lease already in the set will associate the existing client lease in the set with the new desired duration

and renew duration. The client lease is not replaced because it is more likely that the renewal service, rather than the client, has an up-to-date lease expiration. The service is more likely to have an up-to-date expiration because the client should not be renewing a lease that it has passed to a lease renewal service unless the lease is removed first. These semantics also allow `renewFor` to be used in an idempotent fashion.

---

✎ These semantics for `renewFor` (and for `remove`, `next`) require that all leases have a reasonable definition for `equals`. This still needs to be formalized in the *Jini™ Distributed Leasing Specification*.

---

The two-argument form of `renewFor` is equivalent to making the following call:

```
renewFor(leaseToRenew, desiredDuration, Lease.FOREVER)
```

Client leases get returned to clients in a number of ways (via `remove` and `getLeases` calls, as components of events, etc.). In general, they should have their serial format set to `Lease.DURATION` before being transferred to the client's virtual machine. In some exceptional circumstances, this may not be possible (for example, the client lease was recovered from persistent storage and could not be deserialized in the server's virtual machine). In these cases, it is acceptable to transfer the lease to the client using the `Lease.ABSOLUTE` serial format.

Whenever a client lease gets returned to a client, its actual expiration should reflect either:

- ◆ The result of the last successful renewal call that the renewal service made
- or
- ◆ The expiration time the lease originally had when it was added to the set, if the renewal service has not yet successfully renewed the lease

Client leases are removed from the set using the `remove` method. Removal from the set will not cause the lease to be cancelled. The method will return the lease that is being removed. If the lease is not in the set, `null` will be returned.

The `getLeases` method returns all the client leases in the set at the time of the call, as an array of type `Lease`. If one or more of the `Leases` in the array cannot be deserialized, a `LeaseUnmarshalException` is thrown.

```
package net.jini.lease;
```

```
public class LeaseUnmarshalException extends Exception {
    public LeaseUnmarshalException(
        Lease[]          unmarshalledLeases,
        MarshallableObject[] stillMarshallledLeases,
```

```

        Throwable[]      exceptions) {...}
    public Lease[] getUnmarshalledLeases() {...}
    public MarshallableObject[] getStillMarshallLeases() {...}
    public Throwable[] getExceptions() {...}
}

```

The leases that could be successfully deserialized will be returned by the `getUnmarshalledLeases` method of the exception. If no leases could be deserialized, a zero-length array will be returned. The leases that could not be deserialized will be returned in the form of `MarshallableObject`s by the `getStillMarshallLeases` method of the exception. For each element of the array returned by the `getStillMarshallLeases` method, there will be a corresponding element of the array returned by the `getExceptions` method that will hold a `Throwable` that describes why the given lease could not be deserialized.

The throwing of a `LeaseUnmarshalException` represents a, possibly transient, failure in the ability to unmarshal one or more client leases in the set, it does not necessarily imply anything about the state of the renewal service or the set themselves.

The `getRenewalSetLease` method returns the lease associated with the set itself. This method does not make a remote call.

### LR.2.0.1 Events

The lease renewal service does not support multiple simultaneous event listener registrations for the same kind of event. Although it would be useful in some limited circumstances, to do so would require event registrations to be leased separately from the set they are associated with. For the average client of the lease renewal service, this ability would increase the number of leases that it would have to manage. Since the renewal service is based on the premise that some clients have difficulty managing their own leases, increasing the number of leases that a client would need to manage could significantly complicate the implementation of those clients. Because there can be at most one listener for each kind of event, a given set provides a `set/clear` interface instead of the more common `addListener/removeListener` or `addListener/lease.cancel` interfaces.

The source field of each event generated by a lease renewal service is the renewal set that the event is associated with. In the case of an expiration warning event, this is the set which is about to expire. In the case of a renewal failure event, this is the set the client lease was in when the event occurred. Note, the value of the source field will in general be a copy of the set in question, it will be `.equals()` to any other copies of the set the client has in its possession, but in

general it will not be the same object (that is, comparing two sets using `==` will usually return `false`).

Because a given set will only have one expiration warning event registration at a given time, all expiration warning events will have the same event ID: `LeaseRenewalSet.EXPIRATION_WARNING_EVENT_ID`. Similarly, all renewal failure events have the same event ID: `LeaseRenewalSet.RENEWAL_FAILURE_EVENT_ID`.

If a `RemoteEventListener` registered for a renewal failure or expiration warning event throws an `UnknownEventException`, this action will only clear the specific event registration. It will not cancel the lease on the renewal set, or affect any other event registration on the set. If the listener throws a bad object exception, the renewal service may clear that specific event registration; it must not clear any registration associated with other listeners, nor may it cancel the lease on the associated renewal set.

If an event listener is replaced, and one or more event delivery attempts on the original listener failed or were indeterminate, implementations may chose to send some or all of these events to the new listener.

The `setExpirationWarningListener` method of `LeaseRenewalSet` allows the client to register for notification of the approaching expiration of the set's lease. The `listener` argument specifies what listener should be notified when the lease is about to expire. The `minWarning` argument specifies a minimum number of milliseconds before lease expiration that the first event delivery attempt should be made by the service. The service may also make subsequent delivery attempts if the first and any subsequent attempts have been indeterminate. The `minWarning` argument must be zero or a positive number; if it is not, an `IllegalArgumentException` must be thrown. If the current expiration of the set's lease is less than `minWarning` milliseconds away, the event will occur immediately (though it will take time to propagate to the handler).

The `handback` argument to `setExpirationWarningListener` specifies an object that will be part of the expiration warning event notification. This mechanism is detailed in the *Jini™ Distributed Event Specification*.

The `setExpirationWarningListener` method returns the event registration for this event. The `Lease` object associated with the registration will be equivalent (in the sense of `.equals()`) to the `Lease` on the renewal set. Because the event registration shares a lease with the set, clients that want to just remove their expiration warning registration without destroying the set should use the `clearExpirationWarningListener` method described below, instead of canceling the registration's lease. The registration's event ID will be `LeaseRenewalSet.EXPIRATION_WARNING_EVENT_ID`. The source of the registration will be the set. The method must throw a `NullPointerException` if the `listener` argument is `null`. If an event handler has already been specified for

this event the current registration is replaced with the new one. Because both registrations are for the same kind of event, the events sent to the new registration must be in the same sequence as the events sent to the old registration.

The `clearExpirationWarningListener` method of `LeaseRenewalSet` removes the event registration currently associated with the approaching expiration of the set's lease. It is acceptable to call this method even if there is no active registration.

The `setRenewalFailureListener` method of `LeaseRenewalSet` allows the client to register for the event associated with the failure to renew a client lease in the set. These events are generated when a client lease expires while it is still in the set, or when the service attempts to renew a client lease and gets a definite exception. The `listener` argument specifies the listener to be notified if a client lease could not be renewed.

The `handback` argument to `setRenewalFailureListener` specifies an object that will be part of the renewal failure event notification. This mechanism is detailed in the *Jini™ Distributed Event Specification*.

The `setRenewalFailureListener` method returns the event registration for this event. The `Lease` object associated with the registration will be equivalent (in the sense of `.equals()`) to the `Lease` on the renewal set. Because the event registration shares a lease with the set, clients that want to just remove their expiration warning registration without destroying the set should use the `clearRenewalFailureListener` method described below, instead of cancelling the registration's lease. The registration's event ID will be `LeaseRenewalSet.RENEWAL_FAILURE_EVENT_ID`. The source of the registration will be the set. The method must throw `NullPointerException` if the `listener` argument is `null`. If an event handler has already been specified for this event the current registration is replaced with the new one. The returned event registration must have the same event ID as the replaced registration. Because both registrations are for the same kind of event, the events sent to the new registration must be in the same sequence as the events sent to the old registration.

The `clearRenewalFailureListener` method of `LeaseRenewalSet` removes the event registration currently associated with the failure to renew client leases. It is acceptable to call this method even if there is no active registration.

```
package net.jini.lease;

public class ExpirationWarningEvent extends RemoteEvent {
    public ExpirationWarningEvent(LeaseRenewalSet source,
                                   long seqNum,
                                   MarshallableObject handback)
    {...}
}
```

```

        public Lease getRenewalSetLease() {...}
    }

```

ExpirationWarningEvent objects are passed to the event handlers specified in calls to the LeaseRenewalSet method, setExpirationWarningListener. The ExpirationWarningEvent is a subclass of RemoteEvent, and adds no additional state. Because the source of a ExpirationWarningEvent is the set that is about to expire, the lease that needs to be renewed can be obtained by: calling getSource, casting the result to a LeaseRenewalSet, and then calling the set's getRenewalSetLease method. Additionally, a convenience method is provided, getRenewalSetLease, that uses the above technique to retrieve the lease on the set. The Lease object returned will be equivalent (in the sense of .equals()) to other Lease objects associated with the set, but may not be the same object. One notable consequence of having two different objects is that the getExpiration method of the Lease object returned by the event's getRenewalSetLease method may return a different time than the getExpiration methods of other Lease objects granted on the same set.

The expiration time associated with the Lease object returned by the getRenewalSetLease method will reflect the expiration the lease had when the event occurred. Renewal calls may have changed the expiration time of the underlying lease between the time when the event was generated and when it was delivered.

Other aspects of the event's state are described in the *Jini™ Distributed Event Specification*. Sequence numbers for a given event ID are increasing. If there is no gap between two sequence numbers, no events have been missed; if there is a gap, events might (but might not) have been missed.

```

package net.jini.lease;

public class RenewalFailureEvent extends RemoteEvent {
    public RenewalFailureEvent(LeaseRenewalSet source,
                               long seqNum,
                               MarshallableObject handback)
    {...}
    abstract public Lease getLease()
        throws IOException, ClassNotFoundException;
    abstract public Throwable getThrowable()
        throws IOException, ClassNotFoundException;
}

```

RenewalFailureEvent objects are passed to the event handlers specified in calls to the LeaseRenewalSet method, setRenewalFailureListener. The

`RenewalFailureEvent` is a subclass of `RemoteEvent`, adding two additional items of abstract state—the client lease that could not be renewed before expiration and the `Throwable` object that was thrown by the last renewal attempt (if any). The client lease is returned by the `getLease` method, and the `Throwable` object by the `getThrowable` method. If the `Throwable` object is `null`, it can be assumed that the last renewal attempt succeeded or that the renewal service was unable to call `renew` before the client lease expired.

Both the `getLease` and `getThrowable` methods are declared to throw `IOException` and `ClassNotFoundException`. This declaration allows implementations to delay unmarshalling this state until it is actually needed. Once either method of a given `RenewalFailureEvent` object returns normally, future calls on that method must return the same object and may not throw an exception.

If the renewal service was able to renew the lease before the event occurred, the expiration time of the `Lease` object returned by the event's `getLease` method will reflect the result of the last successful renewal call. Note, this time may be distorted by clock skew between hosts if it is currently set to use the `Lease.ABSOLUTE` serial format. If the `Lease` object is using the `Lease.DURATION` serial format, and the event only unmarshals the lease when `getLease` is called, the expiration time may be distorted if a long time has passed between the time the event was generated by the renewal service and when the client called `getLease`. When a renewal failure event is generated for a given lease, that lease is removed from the set.

The event's other state is described in the *Jini™ Distributed Event Specification*. Sequence numbers for a given event ID are increasing. If there is no gap between two sequence numbers, no events have been missed; if there is a gap, events might (but might not) have been missed.

```
package net.jini.lease;

public class BasicRenewalFailureEvent
    extends RenewalFailureEvent
{
    public BasicRenewalFailureEvent(
        LeaseRenewalSet source,
        long seqNum,
        MarshalledObject handback,
        MarshalledObject marshalledLease,
        Throwable throwable) {...}

    public Lease getLease()
        throws IOException, ClassNotFoundException {...}
}
```

```
    public Throwable getThrowable()
        throws IOException, ClassNotFoundException {...}
}
```

`BasicRenewalFailureEvent` provides a basic implementation of `RenewalFailureEvent`, which will not deserialize the lease or the throwable portions of its abstract state until `getLease` or `getThrowable` are called, respectively. Implementations of the renewal service may, but are not required to, use this class.



---

## LS.3 The Semantics



---

## LS.4 Supporting Interfaces and Clients



# EM

---

# Jini™ Event Mailbox Service Specification

## EM.1 Introduction

**T**HE *Jini™ Distributed Event Specification* states the ability to interpose third-party objects, or "agents", into an event notification chain as one of its design goals. This specification also describes a notification mailbox object, which stores and forwards event notifications on behalf of other objects, as a useful, third-party agent example. These mailbox objects can be particularly helpful for objects that need more control over how and when they receive event notifications.

For example, it would be impossible to send event notifications to a transient entity that has detached itself from a system of Jini™ technology-enabled services and/or devices (*Jini system*). An entity, in such a situation, could employ the services of an event mailbox to store event notifications on its behalf before leaving the system. Upon rejoining the Jini system, the entity could then contact the event mailbox in order to retrieve any collected events that it would have otherwise missed. Similarly, an entity that wishes to deactivate could use an event mailbox to collect event notifications on its behalf, while dormant.

Like other Jini technology-enabled services (*Jini services*), the event mailbox service will only grant its services for a limited period of time without an active expression of continuing interest. Therefore, event mailbox clients still need to renew their leases if they intend to maintain the mailbox's services beyond the initially granted lease period. Any resources (for example, remote objects or storage space) associated with a particular client can be freed once the client's lease has expired or been cancelled. In the previous usage scenarios, it might also benefit a transient or deactivatable entity to employ the services of a lease renewal service

(see the *Jini™ Lease Renewal Service Specification*) to help mitigate the issue of lease maintenance.

The remainder of this specification defines the requirements, interfaces, and protocols of the event mailbox service.

### **EM.1.0.1 Goals and Requirements**

The requirements of the set of interfaces specified in this document are:

- ◆ To define a service capable of storing event notifications on behalf of its clients and capable of delivering stored event notifications to those clients upon request
- ◆ To provide this service in such a way that it can be used by entities that are temporarily unable or unwilling to receive event notifications
- ◆ To provide a service that complies with the policies embodied in the Jini technology programming model

The goals of this specification are:

- ◆ To describe the event mailbox service
- ◆ To provide guidance in the use and deployment of the event mailbox service

### **EM.1.1 Other Types**

The types defined in the specification of the event mailbox service are in the `net.jini.event` package. This specification assumes knowledge of the *Jini™ Distributed Event Specification* and the *Jini™ Distributed Leasing Specification*. The following object types may be referenced in this chapter. Whenever referenced, these object types will be referenced in unqualified form:

```
java.rmi.NoSuchObjectException  
java.rmi.RemoteException  
net.jini.core.event.RemoteEvent  
net.jini.core.event.RemoteEventListener  
net.jini.core.lease.Lease
```

---

## EM.2 The Interface

**T**HE EventMailbox interface defines the interface to the event mailbox service. Through this interface, other Jini services and clients may request that event notification management be performed on their behalf. This interface belongs to the `net.jini.event` package, and any service implementing this interface must comply with the definition of a Jini service. This interface is not a remote interface; each implementation of this service exports proxy objects that implement this interface local to the client, using an implementation-specific protocol to communicate with the actual remote server. All of the proxy methods obey normal Java™ Remote Method Invocation (RMI) interface semantics and can therefore be implemented directly using RMI (except where explicitly noted). Two proxy objects are equal (using the `equals` method) if they are proxies for the same event mailbox service.

```
package net.jini.event;

public interface EventMailbox
{
    MailboxRegistration register(long leaseDuration)
        throws RemoteException;
}
```

Event mailbox clients wishing to utilize the mailbox service must first register themselves with the service by invoking its `register` method. Clients then use the methods of the returned `MailboxRegistration` object (or simply, a *registration*) in order to:

- ◆ Manage the lease for this particular registration
- ◆ Obtain a `RemoteEventListener` reference that can be registered with *event generators* (that is, objects that support event notification for changes in their abstract state). This listener will store any received notifications for this particular registration.

- ◆ Enable or disable the delivery of any stored notifications for this particular registration



---

## EM.3 The Semantics

**I**N order to employ the event mailbox service, a client must first register with the event mailbox service by invoking the `EventMailbox` interface's only method, `register`. Each invocation of the `register` method produces a new registration.

The `register` method may throw a `RemoteException`. Typically, this exception occurs when there is a communication failure between the client and the event mailbox service. If this exception does occur, the registration may or may not have been successful.

Each registration with the event mailbox service is persistent across restarts or crashes of the event mailbox service, until the lease on the registration expires or is cancelled.

The `register` method takes a single parameter of type `long` that represents the requested initial lease duration for the registration, in milliseconds. This duration value must be positive (except for the special value of `Lease.ANY`). Otherwise, an `IllegalArgumentException` is thrown.

Every method invocation on an event mailbox service (whether the invocation is directly on the service, or indirectly on a `MailboxRegistration` that the service has created) is atomic with respect to other invocations.



---

## EM.4 Supporting Interfaces and Classes

**T**HE `register` method returns an object that implements `MailboxRegistration`. It is through this interface that the client controls its registration and notification management with the event mailbox service.

```
package net.jini.event;

public interface MailboxRegistration
{
    Lease getLease();
    RemoteEventListener getListener();
    void enableDelivery(RemoteEventListener target)
        throws RemoteException;
    void disableDelivery() throws RemoteException;
}
```

The `MailboxRegistration` interface is not a remote interface. Each implementation of the event mailbox service exports proxy objects that implement this interface local to the client. These proxies use an implementation-specific protocol to communicate with the remote server. All of the remote proxy methods obey normal RMI interface semantics and can therefore be implemented using RMI. Two proxy objects are equal (using the `equals` method) if they are proxies for the same registration, created by the same event mailbox service.

Each remote method of this interface may throw a `RemoteException`. Typically, this exception occurs when there is a communication failure between the client and the event mailbox service. Whenever this exception occurs as a result of the invocation of one of these methods, the method may or may not have successfully completed.

Any invocation of a remote method defined in this interface will result in a `NoSuchObjectException` if the client's registration with the event mailbox service has expired or has been cancelled. Note that upon receipt of a `NoSuchObjectException`, the client can assume that the registration no longer exists; the client cannot assume that the event mailbox service itself no longer exists.

### EM.4.0.1 The Semantics

The `getLease` method returns the `Lease` object associated with the registration. The client can renew or cancel the registration with the mailbox service through the `Lease` object returned by this method (see the *Jini<sup>TM</sup> Distributed Leasing Specification*). This method is not remote and takes no arguments.

The `getListener` method returns an object that implements the `RemoteEventListener` interface. This object, referred to as a *mailbox listener*, can then be submitted as the `RemoteEventListener` argument to an event generator's registration method(s) (see the *Jini<sup>TM</sup> Distributed Event Specification*). Subsequent calls to this method will return equivalent objects (in the `equals` sense). Note that mailbox listeners generated by different registrations will not be equal. This method is not remote and takes no arguments.

The valid period of use for a mailbox listener is tied to the associated registration's lease. A `NoSuchObjectException` will be thrown if an attempt is made to invoke the `notify` method on a mailbox listener whose associated lease has terminated.

Mailbox listener references, just like their associated registrations, are persistent across server restarts or crashes, until their associated registration's lease terminates.

The `enableDelivery` method allows a client to initiate delivery of stored notifications to the client specified listener, referred to as the *target listener*. This method takes a single argument of type `RemoteEventListener`. Subsequent calls to this method simply replace the registration's existing target listener, if any, with the specified target listener. Passing `null` as the listener argument has the same effect as disabling delivery (see below).

Resubmitting a mailbox listener back to the same mailbox service that generated it will result in an `IllegalArgumentException` being thrown. This is necessary in order to prevent a recursive event notification chain. Therefore, the event mailbox service must keep track of any listener objects that it generates and reject the resubmission of those objects.

Once enabled, event delivery continues until there are no more events to be delivered, or until event delivery is explicitly disabled. Events received while delivery is enabled will also be scheduled for delivery.

Event delivery guarantees are implementation specific and are not specified in this document. Implementations are encouraged to retry delivery attempts in the event of failures, though. That is, if an event delivery attempt produces a remote exception, then reasonable efforts should be made to successfully redeliver the event until the associated registration's lease terminates.

Similarly, implementations are encouraged to support this method's intended semantics regarding listener replacement. That is, a mailbox client can reasonably

assume that listener replacement has occurred upon successful return from this method and can therefore safely unexport the previous listener object. This also implies that any in-progress delivery attempts to the previous listener are either successfully cancelled before returning from this method (blocking), or subsequently retried using the replacement listener after returning from this method (non-blocking). Note that the non-blocking case can potentially allow the previous listener to be notified after successfully returning from this method.

The `disableDelivery` method allows the client to cease event delivery to the existing target listener, if any. It is acceptable to call this method even if no target listener is currently enabled. This method takes no arguments.

Again, event delivery guarantees are implementation specific and are not specified in this document. Implementations are encouraged to support the method's intended semantics regarding delivery suspension. That is, a mailbox client can reasonably assume that event delivery has been suspended upon successful return from this method and can therefore safely unexport the previously enabled listener object, if desired. This also implies that any in-progress delivery attempts to the previously enabled listener are either successfully cancelled before returning from this method (blocking), or subsequently retried using the next enabled listener after returning from this method (non-blocking). Note that the non-blocking case can potentially allow the previously enabled listener to be notified after successfully returning from this method.

The event mailbox service does not normally concern itself with the attributes of the `RemoteEvents` that it receives. The one circumstance where it must concern itself, though, is when a target listener throws an `UnknownEventException` during an event delivery attempt. The event mailbox service must maintain a list, on a per-registration basis, of the particular combinations of event identifier and source reference (obtained from the offending `RemoteEvent` object) that produced the exception. The event mailbox must then propagate an `UnknownEventException` back to any event generator that attempts to deliver a `RemoteEvent` with an identifier-source combination held in a registration's unknown exception list. The service will also skip the future delivery of any stored events that have an identifier-source combination held in this list.

Note that the act of comparing event source objects for equality poses a security risk because source objects are potentially given references to other source objects that are currently using the mailbox. Again, although implementation details are not specified in this document, if security is a concern, then care should be taken to prevent independent event sources from obtaining information about each other. For example, it might be reasonable to only compare objects that have the same codebase on the assumption that classes from the same codebase are trusted. If this assumption cannot be made, then mailbox clients should ensure that they register with a mailbox service on an exclusive basis.

A registration's unknown exception list is cleared upon re-enabling delivery with any target listener. This list is persistent across service restarts or crashes, until the associated registration's lease terminates.

The event mailbox does not support multiple, concurrent notification targets per registration. As a result, the interface only supports a set/clear model rather than the more common add/remove model.

Because no two environments may benefit from a single design trade-off, operational parameters — controls for how the event mailbox deals with issues such as delivery guarantees, storage quotas, and low space behavior — may be exposed through an administration interface, which can vary across different event mailbox implementations.