If the managed set of groups is empty, this method will return an 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 an 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 an 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 (*Section DU.3, "LookupDiscovery Utility"*)* of the implementation object, regardless of group membership.

If an 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 an 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 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.2.5 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.2.5.1 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 an 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 an 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 an 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 an 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.2.6 Supporting Interfaces and Classes

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

DU.2.6.1 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).

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

DU.2.6.2 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 groups in which a previously discovered lookup service is a member.

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* locator 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.2.6.3 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 mapping from 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. Note that neither form of the constructor makes a copy of the second parameter. That is, the reference input to the second parameter is shared with the invoking entity.

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`, although the resulting event contains a `non-null` registrars array, the registrars-to-groups map is `null`. Therefore, discovery utilities should no longer use this constructor to instantiate the events they send.

The `getGroups` method returns the mapping from each registrar referenced by the event to the registrar's current set of member groups. If the event was instantiated using the constructor whose second parameter is an array of `ServiceRegistrar` instances, this method will return `null`.

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.2.7 Serialized Forms

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

DU.3 LookupDiscovery Utility

In 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 *The Jini Technology Core Platform Specification,* "Discovery and Join".

The `LookupDiscovery` helper utility in 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.3.1 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.3.2 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.3.3 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 an 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).

The constructor can throw an IOException because the creation of a LookupDiscovery object causes the initiation of the discovery process, a process that can throw IOException when socket allocation occurs.

DU.3.4 Supporting Interfaces and Classes

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

DU.3.4.1 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.2, “The Discovery Management Interfaces” for more information on those interfaces.

DU.3.4.2 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 have the same group name.

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 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.3.5 Serialized Forms

Class	`serialVersionUID`	Serialized Fields
`DiscoveryPermission`	–3036978025008149170L	*none*

DU.4 The LookupLocatorDiscovery Utility

DU.4.1 Overview

THE *The Jini Technology Core Platform Specification,* "Discovery and Join", 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 Technology Core Platform Specification,* "Discovery and Join".

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.4.2 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.4.3 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.4.4 The Semantics

Including the constructor, the `LookupLocatorDiscovery` helper utility class defines three new public methods. All other public methods are inherited from 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. Although it is acceptable to input `null`, if a non-`null` array containing at least one `null` element is input, a `NullPointerException` will be 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 an 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.

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

DU.4.5 Supporting Interfaces

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

DU.4.5.1 The `DiscoveryManagement` Interfaces

The `LookupLocatorDiscovery` class implements the `DiscoveryManagement` and `DiscoveryLocatorManagement` interfaces, which together define methods related to the coordination and management of all locator discovery processing. See Section DU.2, “The Discovery Management Interfaces” for more information on those interfaces.

DU.5 The LookupDiscoveryManager Utility

DU.5.1 Overview

ALTHOUGH 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 Technology Core Platform Specification,* "Discovery and Join") 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.5.2 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.5.3 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.5.4 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 Technology Core Platform Specification,* "Discovery and Join" 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 is initiated as soon as an instance of this class is created. However, if an 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. 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 the `DiscoveryListener` (or the `DiscoveryChangeListener`) interface; an implementation that defines the actions to take upon the receipt of those types of events.

If a listener registered with the `LookupDiscoveryManager` is also an instance of `DiscoveryChangeListener`, then in addition to receiving events related to dis-

covered 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.

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.

The 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.

DU.5.5 Supporting Interfaces and Classes

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

DU.5.5.1 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.2, "The Discovery Management Interfaces" for more information on those interfaces.

DU.5.5.2 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.2, "The Discovery Management Interfaces").

DU.6 Low-Level Discovery Protocol Utilities

THE 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.6.1 The `Constants` Class

DU.6.1.1 Overview

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

DU.6.1.2 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.6.1.3 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.6.1.4 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.6.2 The `OutgoingMulticastRequest` Utility

DU.6.2.1 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.6.2.2 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.6.2.3 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.6.2.4 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.6.3 The `IncomingMulticastRequest` Utility

DU.6.3.1 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.6.3.2 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.6.3.3 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.6.3.4 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.6.4 The `OutgoingMulticastAnnouncement` Utility

DU.6.4.1 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.6.4.2 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.6.4.3 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.6.4.4 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.6.5 The `IncomingMulticastAnnouncement` Utility

DU.6.5.1 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.6.5.2 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.6.5.3 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.6.5.4 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.6.6 The `OutgoingUnicastRequest` Utility

DU.6.6.1 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.6.6.2 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.6.6.3 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.6.6.4 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.6.7 The `IncomingUnicastRequest` Utility

DU.6.7.1 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.6.7.2 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.6.7.3 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.6.7.4 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.6.8 The `OutgoingUnicastResponse` Utility

DU.6.8.1 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 a entity—such as a lookup service—that wishes to transmit responses to unicast requests in order to be discovered through direct, unicast communication.

DU.6.8.2 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.6.8.3 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.6.8.4 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.6.9 The `IncomingUnicastResponse` Utility

DU.6.9.1 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.6.9.2 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.6.9.3 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.6.9.4 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.

EU

Jini Entry Utilities Specification

EU.1 Entry Utilities

ENTRIES 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 Technology Core Platform Specification, "Entry"*.

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
`AbstractEntry`	5071868345060424804L	*none*

LM

Jini Lease Utilities Specification

LM.1 Introduction

THIS 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

THE 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 network technology (Jini technology), defined in *The Jini Technology Core Platform Specification, "Leasing and Distributed Systems"*, requires renewed proof of interest to continue the existence of a leased resource. Thus, any Jini technology-enabled client (Jini client) or Jini technology-enabled service (Jini service) that requests the use of the leased resources provided by another 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 Technology Core Platform Specification,* "Transaction" 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 manager*) 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 and 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.3 The Interface

THE public methods provided by the LeaseRenewalManager class are:

```
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.4 The Semantics

THE term *client* is used in this specification to refer to the local entity that is using the `LeaseRenewalManager` to manage a collection of leases on its behalf. This collection is referred to as the *managed set*.

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 Technology Core Platform Specification, "Distributed Leasing"*).

Each lease in the managed set also has two other associated attributes: a *renewal duration* and a *remaining desired duration*. The remaining desired duration is always the desired expiration less the current time. The renewal duration is usually a positive number and is the new duration that will be requested when the renewal manager renews the lease, unless the renewal duration is greater than the remaining desired duration. If the renewal duration is greater than the remaining desired duration, then the remaining desired duration will be requested when renewing the lease. One exception is that when the desired expiration is `Lease.FOREVER`, the renewal duration may be `Lease.ANY`, in which case `Lease.ANY` will be requested when renewing the client lease, regardless of the value of the remaining desired duration.

For example, if the renewal duration associated with a given lease is 360,000 milliseconds, then when the renewal manager renews the lease, it will ask for a new duration of 360,000 milliseconds—unless the lease is going to reach its desired expiration in less than 360,000 milliseconds. If the lease's desired expiration is within 360,000 milliseconds, the renewal manager will ask for the difference between the current time and the desired expiration. If the renewal duration

had been `Lease.ANY`, the renewal manager would have asked for a new duration of `Lease.ANY`.

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, bad invocation exceptions, and `LeaseExceptions` are considered to be definite exceptions (see *Introduction to Helper Utilities and Services*, Section US.2.6, "What Exceptions Imply about Future Behavior").

The `LeaseRenewalManager` generates two kinds of local events. The first kind is a *renewal failure event* that is generated when the renewal manager finds that it can't renew a lease. The second kind is a *desired expiration reached event,* which is generated when a lease's desired expiration is reached. Each event signals that the renewal manager has removed a lease from the managed set without an explicit request by the client. When placing a lease in the managed set, the client can provide either a `LeaseListener` object that will receive any renewal failure events associated with the lease, or a `DesiredExpirationListener` (a subinterface of `LeaseListener`) object that will receive both renewal failure and desired expiration reached events associated with the lease. Both kinds of event are represented by `LeaseRenewalEvent` objects.

The `LeaseRenewalManager` makes a concurrency guarantee. 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. Because of these concurrency guarantees, it is not possible for the various methods that remove leases from the managed set (for example, `remove`, `cancel`, and `clear`) to guarantee that the renewal manager will not attempt to renew leases that have just been removed. Similarly, it is not possible for the methods that change the desired expiration or renewal duration associated with a lease (for example, `renewUntil`, `renewFor`, and `setExpiration`) to guarantee that the next renewal of the lease will request a duration that is consistent with the new desired expiration and/or renewal duration (it will be consistent with either the old pair or the new pair). However, implementations should keep the window where such renewals could occur as small as possible.

The `LeaseRenewalManager` makes a similar reentrancy guarantee with respect to `LeaseListener` and `DesiredExpirationListener` 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. One implication of this guarantee is that the delivery of events is asynchronous with respect to any call (or sequence of calls) made on the renewal manager after the event occurs; this allows events to be delivered after they have been made