moot by intervening calls on the renewal manager. For example, the renewal manager may deliver events regarding leases that were removed from the managed set after the calls that removed the leases in question completed. Implementations should keep the window where such notifications could occur as small as possible.

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 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 form of the 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 events associated with the lease.

Creating a `LeaseRenewalManager` using the second 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 events associated with 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 expiration time is reached; this will generate a desired expiration reached event.
- An explicit removal of the lease from the set is requested via a `cancel`, `clear`, or `remove` call on the renewal manager.
- The lease's actual expiration time is reached before its desired expiration; this will generate a renewal failure event.
- The renewal manager tries to renew the lease and gets a definite exception; this will generate a renewal failure event.

The `renewUntil` method interprets 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 `renewDuration` 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 the listener implements the interface `DesiredExpirationListener` it will also receive notification if the lease's desired expiration is reached while the lease is still in the set.

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's `notify` method.

If an indefinite exception (see *Introduction to Helper Utilities and Services*, Section US.2.6, "What Exceptions Imply about Future Behavior") 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's `notify` method.

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's `notify` method. 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 as the value of the `LeaseListener` parameter, then no notifications will be delivered.

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 three- and four-argument forms. 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 events associated with this 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 is that, in the context of `renewFor`, the value of the `renewDuration` parameter may be `Lease.ANY` only 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`, a value of `Lease.ANY` for the `desiredDuration` parameter 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, causing the lease to be immediately dropped from the managed set. The method will not throw an exception in this circumstance. A renewal failure event will be generated if the actual expiration is before the desired expiration; otherwise a desired expiration reached event will be generated.

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 the 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.5 Supporting Interfaces and Classes

THE `LeaseRenewalManager` utility class depends on the interfaces `LeaseListener` and `DesiredExpirationListener`. Both of these interfaces reference one class, `LeaseRenewalEvent`.

LM.5.1 The `LeaseListener` Interface

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

```
package net.jini.lease;

public interface LeaseListener extends EventListener
{
    void notify(LeaseRenewalEvent e);
}
```

The `LeaseListener` interface defines the mechanism through which the client receives notification of renewal failure events generated by the renewal manager. These events are delivered using the `notify` method. Renewal failure events are generated when the `LeaseRenewalManager` has failed to renew one of the leases that it 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 renewal failure event notification. When one of the above conditions occurs, the `LeaseRenewalManager` will send an instance of `LeaseRenewalEvent` to that listener object.

LM.5.1.1 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 the `notify` method, the `LeaseRenewalManager` removes the lease that could not be renewed from the managed set of leases. Note also that because of the reentrancy guarantee made by the `LeaseRenewalManager`, new leases can be added safely from within the `notify` method.

LM.5.2 The `DesiredExpirationListener` Interface

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

```
package net.jini.lease;

public interface DesiredExpirationListener
    extends LeaseListener
{
    void expirationReached(LeaseRenewalEvent e);
}
```

The `expirationReached` method receives desired expiration reached events. These are generated when the `LeaseRenewalManager` removes a lease from the managed set because the lease's desired expiration has been reached. Note that any object that has been registered to receive desired expiration reached events will also receive renewal failure events.

It is the responsibility of the client to pass into the `LeaseRenewalManager` a reference to an object that implements the `DesiredExpirationListener` inter-

face, which defines the actions to take upon receipt of a desired expiration reached event notification.

LM.5.2.1 The Semantics

The `expirationReached` method is invoked by the `LeaseRenewalManager` when a lease in the managed set reaches its desired expiration. This method takes one parameter: an instance of the `LeaseRenewalEvent` class, which contains information about the lease who's desired expiration has been reached.

Note that prior to invoking the `expirationReached` method, the `LeaseRenewalManager` removes the affected lease from the managed set of leases. Note also that because of the reentrancy guarantee made by the `LeaseRenewalManager`, callbacks into the renewal manager can be made safely from within the `expirationReached` method.

LM.5.3 The `LeaseRenewalEvent` Class

This class defines the local event that is sent by the `LeaseRenewalManager` to the client's registered listener when the `LeaseRenewalManager` generates a renewal failure event or desired expiration reached event. As previously stated, a renewal failure event typically occurs because the actual expiration time of a lease has been reached before a successful renewal request could be made, or a renewal request resulted in a definite exception. A desired expiration reached event occurs when a lease reaches its desired expiration time at or before its actual expiration. The `LeaseRenewalEvent` class encapsulates information about the lease on which such an event occurs and, if it is a renewal failure, the cause.

```
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 `EventObject` class, adding the following additional items of abstract state: a reference to the associated `Lease` object; a `long` value representing the desired expiration of the 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.

LM.5.3.1 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 associated with this event
- The desired expiration time of the 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, if any, that is associated with the event. This method takes no arguments. If the `LeaseRenewalEvent` represents a desired expiration reached event this method will return `null`.

If the `LeaseRenewalEvent` represents a renewal failure event the `getException` method will return the exception that caused the event to be sent. The conditions under which a renewal failure event 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 a indefinite exception, the `LeaseRenewalManager` will continue to attempt to renew the

affected lease at the appropriate times until: the renewal succeeds, the lease's actual 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.

LM.5.4 Serialized Forms

Class	serialVersionUID	Serialized Fields

`LeaseRenewalEvent`	-6263993416463483O2L	`Lease lease`
		`long expiration`
		`Throwable ex`

JU

Jini Join Utilities Specification

JU.1 Introduction

THIS 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`

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

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 by using the helper utility classes `LookupDiscoveryManager` and `LeaseRenewalManager`. (For information on these classes, refer to *The Jini Technology Core Platform Specification,* "Discovery and Join" 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 that 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*), hence 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

THE 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

THE `JoinManager` helper utility class defines a number of public methods in addition to the constructor. This utility defines an accessor method that allows the entity to retrieve the set of lookup services with which the entity has been registered (by the `JoinManager`), as well as methods that allow the entity to retrieve references to the objects the `JoinManager` uses for discovery management and lease renewal management. Additionally, the `JoinManager` class defines methods the entity may use to manage the attributes associated with the entity, and a method that allows the entity to terminate the join processing being performed on its behalf.

The `equals` method for the `JoinManager` 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 `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.

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` either to use 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 that 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 that 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 Technology Core Platform Specification,* "Lookup Service", and the `ServiceIDListener` interface is described later.

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 that 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 passed as 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.

Note that although a new array is returned by `getAttributes`, the elements of that array are *not* copies. Thus, it important that the elements of the array returned by `getAttributes` not be modified; doing so could cause the state of the `JoinManager` to become corrupted or inconsistent. This potential for corruption or inconsistency is why the effects of modifying the elements of the array returned by `getAttributes` are undefined.

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 `MarshalledObject.equals` method on field values) is equivalent 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 *Jini Lookup Attribute Schema Specification,* Section LS.4.1, "Indicating User Modifiability". 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 `MarshalledObject.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 the method `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 class `ServiceRegistration` (see *The Jini Technology Core Platform Specification,* "Lookup Service"). 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 *Jini Lookup Attribute Schema Specification,* Section LS.4.1, "Indicating User Modifiability"). 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.

The `modifyAttributes` method throws an `IllegalArgumentException` if one of the following conditions is satisfied:

- The length of the array containing the templates does not equal the length of the array containing the attributes
- Any element of either array is not an instance of a valid `Entry` class (for example, the class is not public, does not contain a no-arg constructor, or has at least one public field which is a non-static, non-final primitive)
- The class of `attrSets[`*`i`*`]` is neither the same as, nor a super class of, the class of `attrSetsTemplate[`*`i`*`]`

For the same reason as that noted above in the description of the `addAttributes` method, the effects of modifying the contents of the `attrSets` 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`. 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

THE `JoinManager` class depends on the interfaces `DiscoveryManagement` and `ServiceIDListener` discussed below.

`JoinManager` also references the concrete classes `LookupDiscoveryManager` and `LeaseRenewalManager`, each described in a separate specification.

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 rediscovered. 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 granted by the discarded lookup service, and that 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);
}
```

The intent of this interface is to allow the entity to receive the ServiceID instance assigned to it by the lookup service. It is not part of the semantics of the call that the return from the ServiceIDNotify method can be delayed while the recipient of the call processes the information delivered by the method. 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 notify a separate thread, operating asynchronously, which is designed to place the ServiceID instance in persistent storage.

SD

Jini Service Discovery Utilities Specification

SD.1 Introduction

THIS specification defines helper utility classes, along with supporting interfaces and classes, that encapsulate functionality that can help a Jini technology-enabled service or client (*Jini service* or *Jini 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 services that the entity may employ in some task. This process, in which Jini services as well as 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, the terms *entity* and *client-like entity* will be used interchangeably throughout this specification to refer to Jini clients or services that create an instance of the `ServiceDiscoveryManager` (from the package `net.jini.lookup`) and use the public methods of that class to perform and manage their service discovery duties.

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 that 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 magnified 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 that 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 *service 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 Technology Core Platform Specification,* "Lookup Service". This method may be useful to entities that need 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, because of 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 Technology Core Platform Specification,* "Lookup Service". 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.ServiceEvent
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

THE 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

THE ServiceDiscoveryManager makes certain concurrency guarantees with respect to the methods it defines. When a method of 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.

SD.4.1 The Methods

The ServiceDiscoveryManager helper utility class defines a number of public methods in addition to its constructor. This utility defines a factory method that allows the entity to create a local cache for storing references to desired services that have been previously discovered. Additionally, this class defines a set of methods that the entity may use to query (remotely) each discovered lookup service for other services that are of interest to the entity.

The equals method for the ServiceDiscoveryManager 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.4.1.1 The Constructor

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.

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`, an instance of the `LookupDiscoveryManager` utility class will be constructed to discover 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.

SD.4.1.2 The `createLookupCache` Method

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 definition of the `LookupCache` interface is presented later in this specification.

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 that need 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 interfaces `ServiceItemFilter` and `ServiceDiscoveryListener` 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 that are identical to the semantics described in *The Jini Technology Core Platform Specification,* "ServiceTemplate and Item Matching") 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 Section SD.5.2, "The ServiceItemFilter Interface"). Furthermore, once an instance of the cache is created, the filter associated with that instance will not change during the life of that particular cache. If the filter is changed so that its original behavior is modified, the effect on the cache is undefined.

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.

Events and the Cache

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. 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 at any of the those lookup services:

- The service, matching the template, is registered with one of the lookup services.
- The lease of the matching service is cancelled or expires.
- An attribute set associated with the matching service 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. Second, 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 each of three different references to a service that matches the input criteria is 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 each of the service's attribute sets is 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 unregister with the cache's event mechanism.

SD.4.1.3 The `lookup` Method

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).

The `lookup` method has four versions, each version falling 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`.

Within each category, the versions of `lookup` differ only in whether or not a particular version provides what is referred to as a "wait" (or blocking) feature. That is, each category contains both a non-blocking version of `lookup` which returns immediately when unable to find the desired service, and a blocking version which returns only after waiting a specified amount of time for the desired service to be discovered. The particular version of `lookup` that an entity employs is typically determined by the entity's intended usage pattern.

The descriptions that follow refer to all versions of the `lookup` 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` parameter is treated as the equivalent of a "wildcarded" `ServiceTemplate`.
- If `null` is the value for the `filter` parameter, only template matching will be employed to find the desired services.