- The effects of modifying the contents of the `tmpl` parameter while the invocation is in progress are unpredictable and undefined.

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

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

Each version of `lookup` defined in the `ServiceDiscoveryManager` differs with the corresponding version of `lookup` in `ServiceRegistrar` in the following ways:

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

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

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

For each version of `lookup`, whenever a lookup service query returns a `null` service reference, the filter is bypassed, and the service reference is excluded from

the return object. On the other hand, if the query returns a non-`null` service reference in which the associated array of attribute contains one or more `null` elements, the filter is still applied and the service reference is included in the return object.

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

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

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

The Blocking Feature of `lookup`

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

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

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

Prior to blocking, each of these versions of `lookup` first queries each available lookup service in an attempt to retrieve a satisfactory number of matching ser-

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

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

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

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

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

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

SD.4.1.4 The `getDiscoveryManager` Method

The `getDiscoveryManager` method returns an object that implements the `DiscoveryManagement` interface. The object returned by this method provides the

ServiceDiscoveryManager with the ability to set discovery listeners and to discard previously discovered lookup services when they are found to be unavailable. This method takes no arguments.

SD.4.1.5 The getLeaseRenewalManager Method

The getLeaseRenewalManager method returns a LeaseRenewalManager object. The object returned by this method manages the leases requested and held by the ServiceDiscoveryManager. In general, these leases correspond to the registrations made by the ServiceDiscoveryManager with the event mechanism of each lookup service in the managed set. This method takes no arguments.

SD.4.1.6 The terminate Method

The terminate method performs cleanup duties related to the termination of the event mechanism for *lookup service* discovery, the event mechanism for *service* discovery, and the cache management duties of the ServiceDiscoveryManager. That is, the terminate method will terminate each LookupCache instance created and managed by the ServiceDiscoveryManager. Additionally, if the discovery manager employed by the ServiceDiscoveryManager was created by the ServiceDiscoveryManager itself, then the terminate method will also terminate that discovery manager.

Note that if the discovery manager was created externally and supplied to the ServiceDiscoveryManager, then any reference to that discovery manager held by the entity will remain valid, even after the ServiceDiscoveryManager has been terminated. Similarly, if the entity holds a reference to the lease renewal manager employed by the ServiceDiscoveryManager, that reference will also remain valid after termination, whether lease renewal manager was created externally or by the ServiceDiscoveryManager itself.

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

Upon completion of the termination process, the semantics of all current and future method invocations on the terminated instance of the ServiceDiscoveryManager are undefined.

SD.4.2 Defining Service Equality

The ability to accurately determine when two different service references are equal is very important to the `ServiceDiscoveryManager` in general, and the `LookupCache` in particular. Any restriction placed on that ability can result in inefficient and undesirable behavior. Storing and managing duplicate service references—that is, proxies that refer to the same version of the same back end service—is usually viewed as undesirable. In other words, when storing and managing service references, it is very desirable to be able to determine not only that two different proxies refer to the same back end service, but if they do refer to the same back end, whether or not the current version of the referenced service has been replaced with a new version.

The mechanism employed by the `LookupCache` to avoid storing duplicate service references is the `equals` method provided by the discovered services themselves. This is because an individual well-behaved service of interest will usually register with multiple lookup services, and for each lookup service with which that service registers, the `LookupCache` will receive a separate event containing a reference to the service. When the `LookupCache` receives events from multiple lookup services, the service ID (retrieved from the service reference in the event) together with the `equals` method provided by the service itself, is used to distinguish the service references from each other. In this way, when a new event arrives containing a reference associated with the same service as an already-stored reference, the `LookupCache` can determine whether the new reference is a duplicate or the service has been replaced with a new version of itself. In the former case, the duplicate would be ignored; in the latter case, the old reference would be replaced with the new reference.

Thus, the `LookupCache` relies on the provider of each service to override the `equals` method inherited from the class `Object` with an implementation that allows for the identification of duplicate service proxies. In addition to the `equals` method, each service should also provide a proper implementation of the `hashCode` method. This is because even if an entity never explicitly calls on the `equals` method to compare service references, those references may still be stored in container classes (for example, `Hashtable`) where such comparisons are made "under the covers." From the point of view of the `ServiceDiscoveryManager` and the `LookupCache`, providing an appropriate implementation for both the `equals` method and the `hashCode` method is a key characteristic of good behavior in a Jini service.

Note that there is no need to override either the `equals` method or the `hashCode` method if the service is implemented as a purely remote object in which the service proxy is an RMI stub. In this case, appropriate implementations for both methods are already provided in the stub.

SD.4.3 Exporting `RemoteEventListener` Objects

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

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

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

How each of the affected methods handle the `RemoteException` is dependent on the reason for the exception. If a `RemoteException` (or any other non-fatal exception or error) is thrown during an attempt to register for events from a lookup service, that lookup service will be discarded and made eligible for rediscovery. On the other hand, if a `RemoteException` occurs during an attempt to export the listener, the method from which that attempt is made will re-throw the same exception.

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

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

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

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

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

SD.5 Supporting Interfaces and Classes

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

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

SD.5.1 The `DiscoveryManagement` Interface

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

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

objects registered with the event mechanism of the discovery process (including all listeners registered by the `ServiceDiscoveryManager`).

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

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

SD.5.2 The `ServiceItemFilter` Interface

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

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

```
package net.jini.lookup;

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

SD.5.2.1 The Semantics

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

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

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

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

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

SD.5.3 The `ServiceDiscoveryEvent` Class

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

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

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

```
package net.jini.lookup;

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

```
                                                        {…}

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

SD.5.3.1 The Semantics

The constructor of `ServiceDiscoveryEvent` takes three arguments:

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

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

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

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

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

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

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

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

Because making a copy can be a very expensive process, neither accessor method returns a copy of the service reference associated with the event. Rather, each method returns the appropriate service reference from the cache itself. Due to this cost, listeners (see Section SD.5.4, "The ServiceDiscoveryListener Interface" below) that receive a `ServiceDiscoveryEvent` must not modify the contents of the object returned by these methods; doing so could cause the state of the cache to become corrupted or inconsistent because the objects returned by these methods are also members of the cache. This potential for corruption or inconsistency is why the effects of modifying the object returned by either accessor method are undefined.

SD.5.4 The `ServiceDiscoveryListener` Interface

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

```
package net.jini.lookup;

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

SD.5.4.1 The Semantics

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

ing to the event, including representations of the service's state both before and after the event.

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

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

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

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

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

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

quickly as possible. For example, one strategy might be to simply note the occurrence of the `ServiceDiscoveryEvent` and perform any time-consuming event handling asynchronously.

SD.5.5 The LookupCache Interface

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

```
package net.jini.lookup;

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

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

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

    public void discard(Object serviceReference);

    public void terminate();
}
```

SD.5.5.1 The Semantics

Depending on which version is invoked, the `lookup` method of the `LookupCache` interface returns one or more elements—each matching the input criteria—that were stored in the associated cache. The object that is returned is either a single instance of `ServiceItem` or a set of service references in the form of an array of `ServiceItem` objects. Each service item that is returned by either form of this method must have been previously discovered both to be registered with one or

more of the lookup services in the managed set and to match criteria defined by the entity.

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

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

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

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

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

The `addListener` method will register a `ServiceDiscoveryListener` object with the event mechanism of a `LookupCache`. This listener object will receive a

ServiceDiscoveryEvent upon the discovery, removal, or modification of one of the cache's services, as described previously in "Events and the Cache." This method takes one argument: a reference to the ServiceDiscoveryListener object to register.

If null is input to the addListener method, a NullPointerException is thrown. If the object input is a duplicate (using the equals method) of another element in the set of listeners, no action is taken.

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

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

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

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

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

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

With respect to discarding services, there is a situation that must be handled by all implementations of the `LookupCache`. Because the `LookupCache` discovers a service through a lookup service rather than through the service itself, there is a danger that, unless the `LookupCache` takes action (described below), once a service has been discarded, it may never be rediscovered. This can happen because even though a service may be discarded from the cache, it may not be discarded from the lookup services with which it is registered.

To understand this situation, it might help to first consider the conditions under which a service is normally discarded from the cache and then rediscovered. An entity typically discards a service when the entity determines that the service has become unavailable. Recall that a service usually becomes unavailable to an entity when the service crashes, the service is shut down, or the link between the entity and the service experiences a *network partition*. Under normal circumstances, when a well-defined service becomes unavailable because it has crashed or has been shut down, and the entity—after determining that the service is unavailable—discards the service, the cache will rediscover the service when the service comes back on line. The service is rediscovered because a well-behaved service will typically reregister with each lookup service with which it was registered prior to crashing or shutting down. Note that such a service will reregister even when its original lease with a lookup service is still valid. When the service reregisters with a lookup service, the lookup service notifies the cache's listener that a reregistration has occurred, and the service is then rediscovered.

A special case of the scenario just described involves services that choose to persist their leases. Typically, when a service that persists its leases comes back on line after a crash or a shutdown, the service will *not* reregister with any lookup service for which the associated lease is still valid. If none of the service's leases expire during the period in which the service is down, then when the service comes back on line, it will never reregister with any of the desired lookup services, and the cache will never be notified that the discarded service has become available once again.

Therefore it is important to note that there are conditions that may hinder rediscovering certain types of services that were discarded as a result of a crash or shutdown. This situation should not occur with any frequency because services that persist their leases are expected to be less common than other types of services. However, there is a common scenario in which *any* type of service may be discarded but never rediscovered. This new scenario is characterized not by service crashes or shutdowns, but by communication failures. In this situation, communication failures cause only the entity to view the service as unavailable; that is *each lookup service in the managed set can still communicate with the service*.

As with service crashes or shutdowns, communication failures between the entity and the service can also cause the entity to discard the service. But prob-

lems can arise when the communication failures occur between the entity and the service, but not between the service and any of the lookup services in the managed set. Although the service never goes down, it is still discarded by the entity because the inability to communicate with the service causes the entity to view the service as unavailable. But because the service can still communicate with the lookup services, the service will continue renewing its residency in each lookup service. Thus, since none of the service's leases expire, the service never reregisters with any of the lookup services, and the lookup services will never send events to the cache's listener that cause the service to be rediscovered.

To address the scenarios described above, all implementations must do the following when a service is discarded from the cache:

- Place the reference to the discarded service in separate storage, and remove the reference from the cache's storage (to guarantee that subsequent queries of the cache do not return that same unavailable reference).
- Wait an implementation-dependent amount of time that is likely to exceed the typical service lease duration.
- If a `ServiceEvent` with a transition equal to `TRANSITION_MATCH_NOMATCH` is received (indicating that the service's lease has expired), then the service reference that was set aside can be flushed, and the service is then truly discarded.
- If such a `ServiceEvent` is not received (indicating that a transient communication failure probably occurred), the service reference that was set aside should be placed back in the cache's local storage, and if the entity is registered for events from the cache, the appropriate event should be sent to the entity's registered listener.

The `terminate` method performs cleanup duties related to the termination of the processing being performed by a particular instance of `LookupCache`. For that instance, this method cancels all event leases granted by the lookup services that supplied the contents of the cache, and unexports all remote listener objects registered with those lookup services. The `terminate` method is typically called when the entity is no longer interested in the contents of the `LookupCache`. Upon completion of the termination process, the semantics of all current and future method invocations on the current instance of `LookupCache` are undefined.

LS

Jini Lookup Attribute Schema Specification

LS.1 Introduction

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

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

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

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

LS.1.1 Terminology

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

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

LS.1.2 Design Issues

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

Matching Cannot Always Be Automated

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

Attributes Are Mostly Static

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

Humans Need to Understand Most Attributes

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

- We must provide a mechanism to deal with localization of attributes
- Multiple-valued attributes must provide a way for humans to see only one value (see Section LS.2, “Human Access to Attributes”)

We will cover human accessibility of attributes soon.

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

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

Attributes Must Interoperate with JavaBeans Components

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

LS.1.3 Dependencies

This document relies on the following other specifications:

- *The Jini Technology Core Platform Specification, “Entry”*
- *Jini Entry Utilities Specification*
- *JavaBeans Specification*

LS.2 Human Access to Attributes

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

CONSIDER the following entry class:

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

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

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

- Provide a property editor class of the appropriate type, as described in Section 9.2 of the *JavaBeans Specification*.
- Extend the `java.awt.Component` class; this allows a value to be represented by a JavaBeans component or some other "active" object.
- Provide either a non-default implementation of the `Object.toString` method or inherit directly or indirectly from a class that does so (since the default implementation of `Object.toString` is not useful).

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

LS.3 JavaBeans Components and Design Patterns

LS.3.1 Allowing Display and Modification of Attributes

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

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

LS.3.1.1 Using JavaBeans Components with Entry Classes

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

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

LS.3.2 Associating JavaBeans Components with Entry Classes

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

```
package org.example.foo;

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

public class Size {
    public int value;
}

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

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

```
            return assoc;
        }
    }
```

From the above, the pattern should be relatively clear:

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

LS.3.3 Supporting Interfaces and Classes

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

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

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

    public static Class getBeanClass(Class c)
        throws ClassNotFoundException {…}
}

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

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

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

The `getBeanClass` method returns the class of the JavaBeans component associated with the given attribute class. If the class passed in does not implement the `net.jini.core.entry.Entry` interface, an `IllegalArgumentException` is thrown. If the given attribute class cannot be found, a `ClassNotFoundException` is thrown.

The `EntryBean` interface must be implemented by all JavaBeans components that are intended to be linked to entry objects. The `makeLink` method establishes a link between a JavaBeans component object and an entry object, and the `followLink` method returns the entry object linked to by a particular JavaBeans component. Note that objects that implement the `EntryBean` interface should not be assumed to perform any internal synchronization in their implementations of the `makeLink` or `followLink` methods, or in the `setFoo` or `getFoo` patterns.

LS.4 Generic Attribute Classes

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

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

LS.4.1 Indicating User Modifiability

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

```
public interface ServiceControlled {
}
```

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

LS.4.2 Basic Service Information

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

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

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

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

Each service should register itself with only one instance of this class. The fields of the `ServiceInfo` class have the following meanings:

- The `name` field contains a specific product name, such as `"Ultra 30"` (for a particular workstation) or `"JavaSafe"` (for a specific configuration management service). This string should not include the name of the manufacturer or vendor.
- The `manufacturer` field provides the name of the company that "built" this service. This might be a hardware manufacturer or a software authoring company.
- The `vendor` field contains the name of the company that sells the software or hardware that provides this service. This may be the same name as is in the `manufacturer` field, or it could be the name of a reseller. This field exists so that in cases in which resellers relabel products built by other companies, users will be able to search based on either name.

- The `version` field provides information about the version of this service. It is a free-form field, though we expect that service implementors will follow normal version-naming conventions in using it.
- The `model` field contains the specific model name or number of the product, if any.
- The `serialNumber` field provides the serial number of this instance of the service, if any.

LS.4.3 More Specific Information

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

```
public class ServiceType extends AbstractEntry
        implements ServiceControlled
{
    public ServiceType() {…}
    public java.awt.Image getIcon(int iconKind) {…}
    public String getDisplayName() {…}
    public String getShortDescription() {…}
}
```

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

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

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

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

LS.4.4 Naming a Service

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

```
public class Name extends AbstractEntry {
    public Name() {…}
    public Name(String name) {…}

    public String name;
}

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

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

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

LS.4.5 Adding a Comment to a Service

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

```
public class Comment extends AbstractEntry {
    public Comment() {…}
    public Comment(String comment) {…}

    public String comment;
}

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

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

LS.4.6 Physical Location

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

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

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

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

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

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

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

```
                    String country) {…}

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

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

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

LS.4.7 Status Information

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

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

```
    public StatusType severity;
}

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

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

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

LS.4.8 Serialized Forms

Class	serialVersionUID	Serialized Fields
Address	2896136903322046578L	*all public fields*
AddressBean	4491500432084550577L	Address asoc
Comment	7138608904371928208L	*all public fields*
CommentBean	5272583409036504625L	Comment asoc
Location	–3275276677967431315L	*all public fields*
LocationBean	–4182591284470292829L	Location asoc
Name	27432151480713072O1L	*all public fields*
NameBean	–6026791845102735793L	Name asoc

Class	serialVersionUID	Serialized Fields
ServiceInfo	–1116664185758541509L	*all public fields*
ServiceInfoBean	8352546663361067804L	ServiceInfo asoc
ServiceType	–6443809721367395836L	*all public fields*
Status	–5193075846115040838L	*all public fields*
StatusBean	–1975539395914887503L	Status asoc
StatusType	–8268735508512712203L	int type

LD

Jini Lookup Discovery Service

LD.1 Introduction

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

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

The facilities of the lookup discovery service are of particular value in a scenario in which a new lookup service is added to a long-lived djinn containing mul-

tiple inactive services. Without the use of a lookup discovery service, the time frame over which the new lookup service is fully populated can be both unpredictable and unbounded.

To understand why this time frame can be unpredictable, consider the fact that an inactive service has no way of discovering a new lookup service. This means that each inactive service in the djinn that wishes to discover and join a new lookup service must first activate. Since activation of a service occurs when some client attempts to use the service, the amount of time that passes between the arrival of the new lookup service and the activation of the service can vary greatly over the range of services in the djinn. Thus, the time frame over which the lookup service becomes fully populated cannot be predicted because it could take arbitrarily long before all of the services activate and then discover and join the new lookup service.

In addition to being unpredictable, the time it takes for the lookup service to fully populate can also be unbounded. This is because there is no guarantee that the lookup service will send multicast announcements between the time the service activates and the time it deactivates. If the timing is right, it is possible that one or more of the services in the djinn may never discover and join the new lookup service. Thus, without the use of the lookup discovery service, the new lookup service may never fully populate.

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

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

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