Jini™ Entry Specification

A Jini™ entry provides a way to store a collection of related objects in a way amenable to simple exact-match searches. This specification describes the types involved and their operational semantics, including matching semantics

THE NETWORK IS THE COMPUTER™

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

Revision 1.0
January 25, 1999

Contents

Entries and Templates 1

Entries are designed to be used in distributed algorithms for which exact-match lookup semantics are useful. An entry is a typed set of objects, each of which may be tested for exact match with a template.

1.1 Operations

A service that uses entries will support methods that let you use entry objects. In this document we will use the term “operation” for such methods. There are three types of operations:

- *Store Operations*—operations that store one or more entries, usually for future matches.
- *Match Operations*—operations that search for entries that match one or more templates.
- *Fetch Operations*—operations that return one or more entries.

It is possible for a single method to provide more than one of the operation types: for example, consider a method that returns an entry that matches a given template. Such a method can be logically split into two operation types (match and fetch), so any statements made in this specification about either operation type would apply to the appropriate part of the method’s behavior.

1.2 Entry

An entry is a typed group of object references represented by a class that implements the marker interface `net.jini.core.entry.Entry`. Two different entries have the same type if and only if they are of the same class.

```
package net.jini.core.entry;
public interface Entry extends java.io.Serializable { }
```

For the purpose of this specification, the term “field” when applied to an entry will mean fields that are public, non-static, non-transient, and non-final. Other fields of an entry are not affected by entry operations. In particular, when an entry object is created and filled in by a fetch operation, only the public non-static, non-transient, and non-final fields of the entry are set. Other fields are not affected, except as set by the class’s no-arg constructor.

Each `Entry` class must provide a public no-arg constructor. Entries may not have fields of primitive type (`int`, `boolean`, etc.), although the objects they refer to may have primitive fields and non-public fields. For any type of operation, an attempt to use a malformed entry type that has primitive fields or does not have a no-arg constructor throws `IllegalArgumentException`.

1.3 Serializing `Entry` Objects

`Entry` objects are typically not stored directly by an entry-using service (one that supports one or more entry operations). The client of the service will typically turn an `Entry` into an implementation-specific representation that includes a serialized form of the entry’s class and each of the entry’s fields. (This transformation is typically not explicit, but done by a client-side proxy object for the remote service.) It is these implementation-specific forms that are typically stored and retrieved from the service. These forms are not directly visible to the client, but their existence has important effects on the operational contract. The semantics of this section apply to all operation types, whether the above assumptions are true or not for a particular service.

Each entry has its fields serialized separately. In other words, if two fields of the entry refer to the same object (directly or indirectly), the serialized form that is compared for each field will have a separate copy of that object. This is only true of different fields of an entry; if an object graph of a particular field refers to the same object twice, the graph will be serialized and reconstituted with a single copy of that object.

A fetch operation returns an entry that has been created using the entry type's no-arg constructor, and whose fields have been filled in from such a serialized form. Thus, if two fields, directly or indirectly, refer to the same underlying object, the fetched entry will have independent copies of the original underlying object.

This behavior, although not obvious, is both logically correct, and practically advantageous. Logically, the fields can refer to object graphs, but the entry is not itself a graph of objects, and so should not be reconstructed as one. An entry (relative to the service) is a set of separate fields, not a unit of its own. From a practical standpoint, viewing an entry as a single graph of objects requires a matching service to parse and understand the serialized form, because the ordering of objects in the written entry will be different from that in a template that can match it.

The serialized form for each field is a `java.rmi.MarshalledObject` object instance, which provides an `equals` method that conforms to the above matching semantics for a field. `MarshalledObject` also attaches a codebase to class descriptions in the serialized form, so classes written as part of an entry can be downloaded by a client when they are retrieved from the service. In a store operation, the class of the entry type itself is also written with a `MarshalledObject`, ensuring that it, too, may be downloaded from a codebase.

1.4 UnusableEntryException

A `net.jini.core.entry.UnusableEntryException` will be thrown if the serialized fields of an entry being fetched cannot be deserialized for any reason:

```
package net.jini.core.entry;
public class UnusableEntryException extends Exception {
    public Entry partialEntry;
    public String[] unusableFields;
    public Throwable[] nestedExceptions;
    public UnusableEntryException(Entry partial,
                String[] badFields, Throwable[] exceptions) {…}
    public UnusableEntryException(Throwable e) {…}
}
```

The `partialEntry` field will refer to an entry of the type that would have been fetched, with all the usable fields filled in. Fields whose deserialization caused an exception will be `null` and have their names listed in the

unusableFields string array. For each element in unusableFields, the corresponding element of nestedExceptions will refer to the exception that caused the field to fail deserialization.

If the retrieved entry is corrupt in such a way as to prevent even an attempt at field deserialization (such as being unable to load the exact class for the entry), partialEntry and unusableFields will both be null, and nestedExceptions will be a single element array with the offending exception.

The kinds of exceptions that can show up in nestedExceptions are:

- ClassNotFoundException: The class of an object that was serialized cannot be found.
- InstantiationException: An object could not be created for a given type.
- IllegalAccessException: The field in the entry was either inaccessible or final.
- java.io.ObjectStreamException: The field could not be deserialized because of object stream problems.
- java.rmi.RemoteException: When a RemoteException is the nested exception of an UnusableEntryException, it means that a remote reference in the entry's state is no longer valid (more below). Remote errors associated with a method that is a fetch operation (such as being unable to contact a remote server) are not reflected by UnusableEntryException, but in some other way defined by the method (typically by the method throwing RemoteException itself).

Generally speaking, storing a remote reference to a non-persistent remote object in an entry is risky. Because entries are stored in serialized form, entries stored in an entry-based service will typically not participate in the garbage collection that keeps such references valid. However, if the reference is not persistent because the referenced server does not export persistent references, that garbage collection is the only way to ensure the ongoing validity of a remote reference. If a field contains a reference to a non-persistent remote object, either directly or indirectly, it is possible that the reference will no longer be valid when it is deserialized. In such a case, the client code must decide whether to remove the entry from the entry-fetching service, to store the entry back into the service, or to leave the service as it is.

In the 1.2 Java™ Development Kit (JDK) software, activatable object references fit this need for persistent references. If you do not use a persistent type, you will have to handle the above problems with remote references. You may choose instead to have your entries store information sufficient to look up the current reference rather than putting actual references into the entry.

1.5 Templates and Matching

Match operations use entry objects of a given type, whose fields can either have *values* (references to objects) or *wildcards* (`null` references). When considering a template *T* as a potential match against an entry *E*, fields with values in *T* must be matched exactly by the value in the same field of *E*. Wildcards in *T* match any value in the same field of *E*.

The type of *E* must be that of *T*, or be a subtype of the type of *T*, in which case all fields added by the subtype are considered to be wildcards. This enables a template to match entries of any of its subtypes. If the matching is coupled with a fetch operation, the fetched entry must have the type of *E*.

The values of two fields match if `MarshalledObject.equals` returns `true` for their `MarshalledObject` instances. This will happen if the bytes generated by their serialized form match, ignoring differences of serialization stream implementation (such as blocking factors for buffering). Class version differences that change the bytes generated by serialization will cause objects not to match. Neither entries nor their fields are matched using the `Object.equals` method or any other form of type-specific value matching.

You can store an entry that has a `null`-valued field, but you cannot match explicitly on a `null` value in that field, because `null` signals a wildcard field. If you have a field in an entry that may be variously `null` or not, you can set the field to `null` in your entry. If you need to write templates that distinguish between set and un-set values for that field, you can (for example) add a `Boolean` field that indicates whether the field is set, and use a `Boolean` value for that field in templates.

An entry that has no wildcards is a valid template.

1.6 Serialized Form

The `serialVersionUID` of `UnusableEntryException` is -2199083666668626172L. The only serialized fields are the declared public fields.