



JavaSpacesTM Specification

The JavaSpacesTM package provides a distributed persistence and object exchange mechanism for code written in the JavaTM programming language. Objects are written in entries that provide a typed grouping of relevant fields. Clients can perform simple operations on a JavaSpaces server to write new entries, lookup existing entries, and remove entries from the space. Using these tools, you can write systems to store state, and also write systems that use flow of data to implement distributed algorithms and let the JavaSpaces system implement distributed persistence for you.

Revision 1.0 Beta, July 17, 1998 5:19 pm



Copyright © 1998 Sun Microsystems, Inc.
901 San Antonio Road, Palo Alto, CA 94303 USA

All rights reserved. Copyright in this document is owned by Sun Microsystems, Inc.

Sun Microsystems, Inc. (SUN) hereby grants to you at no charge a nonexclusive, nontransferable, worldwide, limited license (without the right to sublicense) under SUN's intellectual property rights that are essential to practice the Jini 1.0 Specification ("Specification") to use the Specification for internal evaluation purposes only. Other than this limited license, you acquire no right, title, or interest in or to the Specification and you shall have no right to use the Specification for productive or commercial use.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-1(a).

This software and documentation is the confidential and proprietary information of Sun Microsystems, Inc. ("Confidential Information"). You shall not disclose such Confidential Information and shall use it only in accordance with the terms of the license agreement you entered into with Sun.

SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.

TRADEMARKS

Sun, the Sun logo, Sun Microsystems, Jini, JavaSoft, JavaBeans, JDK, Java, HotJava, HotJava Views, Visual Java, Solaris, NEO, Joe, Netra, NFS, ONC, ONC+, OpenWindows, PC-NFS, EmbeddedJava, PersonalJava, SNM, SunNet Manager, Solaris sunburst design, Solstice, SunCore, SolarNet, SunWeb, Sun Workstation, The Network Is The Computer, ToolTalk, Ultra, Ultracomputing, Ultraserver, Where The Network Is Going, Sun WorkShop, XView, Java WorkShop, the Java Coffee Cup logo, and Visual Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

Table of Contents



About This Document	5
0.1 Status	5
0.2 Comments	5
Introduction	1
1.1 Overview	1
1.2 The JavaSpaces Model and Terms	2
1.3 Benefits	4
1.4 JavaSpaces and Databases	6
1.5 JavaSpaces Design and Linda Systems	6
1.6 Goals & Requirements	8
1.7 Dependencies	9
Entries, Templates, and Operations	11
2.1 Entry and AbstractEntry	11
2.2 net.jini.space.JavaSpace	13
2.3 Serializing Entry Objects	15



2.4	Templates and Matching	17
2.5	write.....	18
2.6	readIfExists and read	19
2.7	takeIfExists and take	20
2.8	snapshot	20
2.9	notify	21
2.10	Operation Ordering	23
	Transactions.....	25
3.1	Operations Under Transactions	25
3.2	Transactions and ACID Properties.....	26
	References and Further Reading	29
4.1	References	29
4.2	Further Reading	29

About This Document



0.1 Status

This is the 1.0 Beta version of the JavaSpaces specification.

0.2 Comments

Please direct comments to js-comments@jse.east.sun.com



1.1 Overview

Distributed systems are hard to build. They require careful thinking about problems that do not occur in local computation. The primary problems are those of partial failure, greatly increased latency, and language compatibility[1]. The Java™ programming language¹ has a remote method invocation system called RMI[2] that lets you approach general distributed computation in the Java programming language using techniques natural to the Java programming language and application environment. This is layered on the Java platform's object serialization mechanism[3] to marshal parameters of remote methods into a form that can be shipped across the wire and unmarshalled in a remote server's Java Virtual Machine(JVM).

This specification describes the JavaSpaces architecture, which is designed to help you solve two related problems: distributed persistence and the design of distributed algorithms. JavaSpaces implementations use RMI and the serialization feature of the Java programming language to accomplish these goals.

1. "JavaSpaces" and "Java" are trademarks of Sun Microsystems, Inc.



1.2 The JavaSpaces Model and Terms

A JavaSpaces server holds *entries*. An entry is a typed group of objects, expressed in a class for the Java platform that implements the interface `net.jini.space.Entry`.

An entry can be *written* into a JavaSpaces server, which creates a copy of that entry in the space² that can be used in future lookup operations.

You can look up entries in a JavaSpaces server using *templates*, which are entry objects that have some or all of its fields set to specified *values* that must be matched exactly. Remaining fields are left as *wildcards*—these fields are not used in the lookup.

There are two kinds of lookup operations: *read* and *take*. A *read* request to a space returns either an entry that matches the template on which the read is done, or an indication that no match was found. A *take* request operates like a read, but if a match is found, the matching entry is removed from the space.

You can request a JavaSpaces server to *notify* you when an entry that matches a specified template is written. This is done using the distributed event model contained in the package `net.jini.event` and described in the *Distributed Event Specification*.

All operations that modify a JavaSpaces server are performed in a transactionally secure manner with respect to that space. That is, if a write operation returns successfully, that entry was written into the space (although an intervening take may remove it from the space before subsequent lookup of yours). And if a take operation returns an entry, that entry has been removed from the space, and no future operation will read or take the same entry. In other words, each entry in the space can be taken at most once. Note, however, that two or more entries in a space may have exactly the same value.

The JavaSpaces architecture supports a simple transaction mechanism that allows multi-operation and/or multi-space updates to complete atomically. This is done using the two-phase commit model under the default transaction semantics, as defined in the package `net.jini.transaction` and described in the *Transaction Specification*.

2. The term “space” is used to refer to a JavaSpaces server implementation.



Entries written into a JavaSpaces server are governed by a lease, as defined in the package `net.jini.lease` and described in the *Distributed Lease Specification*.

1.2.1 *Distributed Persistence*

JavaSpaces implementations provide a mechanism for storing a group of related objects and retrieving them based on a value-matching lookup for specified fields. This allows a JavaSpaces server to be used to store and retrieve objects on a remote system.

1.2.2 *Distributed Algorithms as Flows of Objects*

Many distributed algorithms can be modeled as a flow of objects between participants. This is different from the traditional way of approaching distributed computing, which is to create method-invocation-style protocols between participants. In the JavaSpaces architecture's "flow of objects" approach, protocols are based on the movement of objects into and out of JavaSpaces implementations.

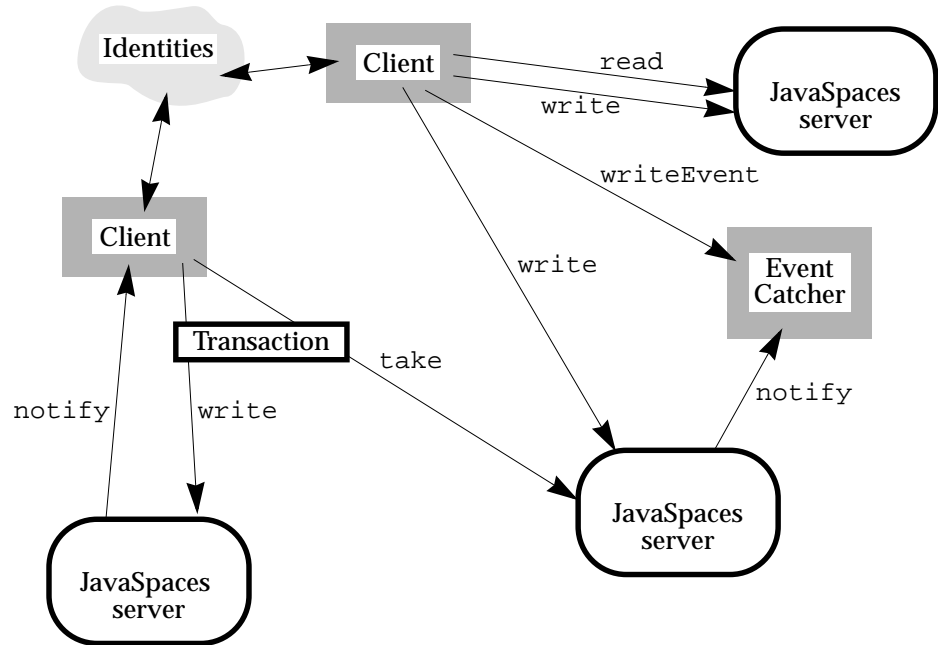
For example, a book ordering system might look like this:

- ◆ A book buyer wants to buy 100 copies of a book. They write a request for bids into a particular public JavaSpaces server.
- ◆ The broker runs a server that takes those requests out of the space and writes them into a JavaSpaces server for each book seller who registered with the broker for that service.
- ◆ A server at each book seller removes the requests from its JavaSpaces server, presents the request to a human to prepare a bid, and writes the bid into the space specified in the book buyer's request for bids.
- ◆ When the bidding period closes, the buyer takes all the bids from the space and presents them to a human to select the winning bid.

A method-invocation-style would create particular remote interfaces for these interactions. With a flow-of-objects approach, only one interface is required—the `net.jini.space.JavaSpace` interface.



In general, the JavaSpaces application world looks like this:



Clients perform operations that map entries or templates onto JavaSpaces applications. These can be singleton operations (as with the upper client), or contained in transactions (as with the lower client) so that all or none of the operations take place. A single client can interact with as many spaces as it needs to. Identities are accessed from the security subsystem and passed as parameters to method invocations. Notifications go to event catchers, which may be clients themselves, or proxies for a client (such as a store-and-forward mailbox).

1.3 Benefits

JavaSpaces implementations are tools for building distributed protocols. They are designed to work with applications that can model themselves as flows of objects through one or more servers. If your application can be modeled this way, JavaSpaces technology will provide many benefits.



JavaSpaces implementations can provide a reliable distributed storage system for the objects. In the book buying example, the designer of the system had to define the protocol for the participants and design the various kinds of entries that must be passed around. This effort is akin to designing the remote interfaces that an equivalent customized service would require. Both the JavaSpaces system solution and the customized solution would require someone to write the code that presented requests and bids to humans in a GUI. And in both systems, someone would have to write code to handle the seller's registrations of interest with the broker.

The server for the model that uses the JavaSpaces API would be implemented at that point.

The customized system would need to implement the servers. These servers would have to handle concurrent access from multiple clients. Someone would need to design and implement a reliable storage strategy that guaranteed the entries written to the server would not be lost in an unrecoverable or undetectable way. If multiple bids needed to be made atomically, a distributed transaction system would have to be implemented.

All these concerns are solved in the JavaSpaces servers. They handle concurrent access. They store and retrieve entries atomically. And they provide an implementation of the distributed transaction mechanism.

This is the power of the JavaSpaces architecture — many common needs are addressed in a simple platform that can be easily understood, and used in powerful ways.

JavaSpaces applications also help with data that would traditionally be stored in a file system, such as user preferences, email messages, images, and so on. Actually this is not a different use of a JavaSpaces server. Such uses of a file system can equally be viewed as passing objects that contain state from one external object (the image editor) to another (the window system that uses the image as a screen background). And JavaSpaces implementations enhance this functionality because they store objects, not just data, so the image can have abstract behavior, not just information that must be interpreted by some external application(s).

JavaSpaces implementations can provide distributed *object* persistence with objects in the Java programming language. Because code written in the Java programming language is downloadable, JavaSpaces entries can store objects whose behavior will be transmitted from the writer to the readers, just as in



Java RMI. An entry in a space may, when fetched, cause some active behavior in the reading client. This is the benefit of storing objects, not just data, in an accessible repository for distributed cooperative computing.

1.4 *JavaSpaces and Databases*

A JavaSpaces server can store persistent data which is later searchable. But a JavaSpaces implementation is not a relational or object database. JavaSpaces applications are designed to help solve problems in distributed computing, not to be used primarily as a data repository (although there are many data storage uses for JavaSpaces applications). Some important differences are:

- ◆ Relational databases understand the data they store and manipulate it directly via query languages. JavaSpaces servers store entries that they understand only by type and the serialized form of each field. There are no general queries in the JavaSpaces design, only “exact match” or “don’t care” for a given field. You design your flow of objects so that this is sufficient and powerful.
- ◆ Object databases provide an object oriented image of stored data that can be modified and used, nearly as if it were transient memory. JavaSpaces systems do not provide a nearly-transparent persistent/transient layer, and work only on copies of entries.

These differences exist because JavaSpaces applications are designed for a different purpose than either relational or object databases. A JavaSpaces server can be used for simple persistent storage, such as storing a user’s preferences that can be looked up by the user’s ID or name. JavaSpaces application functionality is somewhere between that of a filesystem and a database, but it is neither.

1.5 *JavaSpaces Design and Linda³ Systems*

The JavaSpaces design is strongly influenced by Linda systems, which support a similar model of entry-based shared concurrent processing. Our references (§4) include several that describe Linda-style systems.

3. “Linda” is the name of a public domain technology originally propounded by Dr. David Gelertner of Yale University. “Linda” is also claimed as a trademark for certain goods by Scientific Computing Associates, Inc. This discussion refers to the public domain “Linda” technology.



No knowledge of Linda systems is required to understand this specification. This section discusses the relationship of JavaSpaces systems with respect to Linda systems for the benefit of those already familiar with Linda programming. Other readers should feel free to skip ahead.

JavaSpaces systems are similar to Linda systems in that they store collections of information for future computation and are driven by value-based lookup. They differ in some important ways:

- ◆ Linda systems have not used rich typing. JavaSpaces systems take a deep concern with typing from the Java platform type-safe environment. In JavaSpaces systems, entries themselves, not just their fields, are typed — two different entries with the same field types but with different data types for the Java programming language are different entries. For example, an entry that had a string and two double values could be either a named point or a named vector. In JavaSpaces systems these two entry types would have specific different classes for the Java platform, and templates for one type would never match the other, even if the values are compatible (§2.4).
- ◆ Entries are typed as objects in the Java programming language, so they may have methods associated with them. This provides a way of associating behavior with entries (§2.4).
- ◆ As another result of typed entries, JavaSpaces implementations allow matching of subtypes — a template match can return a type that is a subtype of the template type (§2.4). This means that the read or take may return more state than anticipated. In combination with the previous point, this means that entry behavior can be polymorphic in the usual object-oriented style that the Java platform provides.
- ◆ The fields of JavaSpaces entries are objects in the Java programming language. Any object data type for the Java programming language can be used as a template for matching entry lookups as long as it has certain properties (§2.4). This means that computing systems constructed using the JavaSpaces API are object-oriented from top to bottom, and behavior-based (agent-like) applications can use JavaSpaces implementations for coordination.
- ◆ Most environments will have more than one JavaSpaces server. Most Linda tuple spaces have one tuple space for all cooperating threads. So JavaSpaces transactions can span multiple spaces (and even non-JavaSpaces transaction participants).



- ◆ Entries written into a JavaSpaces server are leased. This helps keep the space free of debris left behind due to system crashes and network failures.
- ◆ The JavaSpaces API does not provide an equivalent of “eval” because it would require the server to execute arbitrary computation on behalf of the client. Such a general compute server system has its own large number of requirements (such as security and fairness).

On the nomenclature side, the JavaSpaces API uses a more accessible set of terms than the traditional Linda terms. The term mappings are “entry” for “tuple”, “value” for “actual”, “wildcard” for “formal”, “write” for “out”, and “take” for “in”. So the Linda sentence “When you ‘out’ a tuple make sure that actuals and formals in ‘in’ and ‘read’ can do appropriate matching” would be translated to “When you write an entry make sure that values and wildcards in ‘take’ and ‘read’ can do appropriate matching.”

1.6 Goals & Requirements

The goals of the JavaSpaces design are:

- ◆ Provide a platform for designing distributed computing systems that simplifies the design and implementation of those systems.
- ◆ The client side should have few classes, both to keep the client-side model simple, and to make downloading of the client classes quick.
- ◆ The client side should have a small footprint, since it will run on computers with limited local memory.
- ◆ A variety of server implementations should be possible, including relational database storage and object-oriented database storage.
- ◆ It should be possible to create replicated JavaSpaces implementations.

The requirements for JavaSpaces applications are:

- ◆ The client side of JavaSpaces must be 100% Pure Java.
- ◆ Clients must be oblivious to the implementation details of the server. The same entries and templates must work in the same ways no matter which server is used.



1.7 Dependencies

This document relies upon the following other specifications:

- ◆ RMI
- ◆ Object Serialization
- ◆ Distributed Events
- ◆ Distributed Leasing
- ◆ Transactions



Entries, Templates, and Operations



There are four primary kinds of operations that you can invoke on a JavaSpaces server. Each operation has parameters that are entries, including some that are templates, which are a kind of entry. This chapter describes entries, templates, and the details of the operations, which are:

- ◆ `write`— Write the given entry into this JavaSpaces implementation.
- ◆ `read`— Read an entry from this JavaSpaces implementation that matches the given template.
- ◆ `take`— Read an entry from this JavaSpaces implementation that matches the given template, removing it from this JavaSpaces implementation.
- ◆ `notify`— Notify a specified object when entries that match the given template are written into this JavaSpaces implementation.

As used in this document, the term “operation” refers to a single invocation of a method; thus, for example, two different `take` operations may have different templates.

2.1 Entry and AbstractEntry

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

```
public interface Entry extends Serializable { }
```



Only the public fields of an entry are considered by this specification, so the term “fields” in this specification means “public fields”. A read or take of an entry that has non-public fields will result in an object with the default initial values for those fields. Entry fields must all be references to `Serializable` objects.

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 and non-public fields. Any attempt to use a malformed entry type that has primitive fields or does not have a no-arg constructor throws `IllegalArgumentException`.

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

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

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



2.2 *net.jini.space.JavaSpace*

All operations are invoked on an object that implements the `JavaSpace` interface. For example, the following code fragment would write an entry of type `AttrEntry` into the `JavaSpaces` implementation referred to by the identifier `space`:

```
JavaSpace space = getSpace();
AttrEntry e = new AttrEntry();
e.name = "Duke";
e.value = new GIFImage("dukeWave.gif");
space.write(e, null, 60 * 60 * 1000); // one hour
// lease is ignored -- one hour will be enough
```

The `JavaSpace` interface is:

```
import java.rmi.*;
import net.jini.event.*;
import net.jini.transaction.*;
import net.jini.lease.*;

public interface JavaSpace {
    Lease write(Entry e, Transaction txn, long lease)
        throws RemoteException, TransactionException;
    public final long NO_WAIT = -1; // don't wait at all
    Entry read(Entry tmpl, Transaction txn, long timeout)
        throws TransactionException, UnusableEntryException,
        RemoteException, InterruptedException;
    Entry readIfExists(Entry tmpl, Transaction txn, long timeout)
        throws TransactionException, UnusableEntryException,
        RemoteException, InterruptedException;
    Entry take(Entry tmpl, Transaction txn, long timeout)
        throws TransactionException, UnusableEntryException,
        RemoteException, InterruptedException;
    Entry takeIfExists(Entry tmpl, Transaction txn, long timeout)
        throws TransactionException, UnusableEntryException,
        RemoteException, InterruptedException;
    EventRegistration notify(Entry tmpl, Transaction txn,
        RemoteEventListener listener, long lease,
        MarshalledObject handback)
        throws RemoteException, TransactionException;
    Entry snapshot(Entry e) throws RemoteException;
}
```



The `Transaction` and `TransactionException` types in the above signatures are imported from `net.jini.transaction`. The `Lease` type is imported from `net.jini.lease`. The `RemoteEventListener` and `EventRegistration` types are imported from `net.jini.event`.

In all methods that have the parameter, `txn` may be `null`, which means that no `Transaction` object is managing the operation (§3).

The `JavaSpace` interface is not a remote interface. Each implementation of a `JavaSpaces` server exports objects that implement the `JavaSpace` interface locally on the client, talking to the actual `JavaSpaces` server through an implementation-specific interface. An implementation of any `JavaSpace` method may communicate with a remote `JavaSpaces` server to accomplish its goal; hence, each method throws `RemoteException` to allow for possible failures. Unless noted otherwise in this specification, when you invoke `JavaSpace` methods you should expect `RemoteExceptions` on method calls in the same cases where you would expect them for methods invoked directly on an RMI remote reference. For example, invoking `snapshot` may require talking to the remote `JavaSpaces` server, and so may get a `RemoteException` should the server crash during the operation.

The details of each `JavaSpace` method are given in the sections that follow.

2.2.1 *InternalSpaceException*

The exception `InternalSpaceException` may be thrown by a `JavaSpaces` implementation that encounters an inconsistency in its own internal state or is unable to process a request because of internal limitations (such as storage space being exhausted). This exception is a subclass of `RuntimeException`. The exception has two constructors: one that takes a `String` description and the other that takes a `String` and a nested exception; both constructors simply invoke the `RuntimeException` constructor that takes a `String` argument.

```
public class InternalSpaceException extends RuntimeException {
    public final Throwable nestedException;
    public InternalSpaceException(String msg) {...}
    public InternalSpaceException(String msg, RemoteException e) {...}
    public printStackTrace() {...}
    public printStackTrace(PrintStream out) {...}
    public printStackTrace(PrintWriter out) {...}
}
```



The `nestedException` field is the one passed to the second constructor, or `null` if the first constructor was used. The overridden `printStackTrace` methods print out the stack trace of the exception, and if `nestedException` is not `null`, print out that stack trace as well.

2.3 Serializing *Entry* Objects

`Entry` objects are not stored directly in the space. The `JavaSpaces` client will 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. It is these forms that are stored and retrieved from the `JavaSpaces` server. These forms are not directly visible to the client, but their existence has important effects on the operational contract.

The primary consequence is that each entry used with a `JavaSpaces` implementation 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.) These serialization semantics regularize the serialized form of entries and templates, which may have `null` fields where a matching entry has values. Otherwise, a template that had a wildcard for the first object would have the full, serialized object where the second reference appears, but the matching entry would have only a back-reference to the first object in that place. For details, see the specification for object serialization.

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 description in the serialized form, so classes written as part of an entry can be downloaded by a client when they are retrieved from the `JavaSpaces` server. The class of the entry type itself is also written with a `MarshalledObject`, ensuring that it, too, may be downloaded from a codebase.

2.3.1 *UnusableEntryException*

If the serialized fields of an entry being returned cannot be deserialized for any reason, `read` or `take` will throw an `UnusableEntryException`:



```
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 returned, 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.
- ◆ `ObjectStreamException`: The field could not be deserialized because of object stream problems.
- ◆ `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 on the read call itself are passed through by `read`, and so arrive as `RemoteException` objects, not `UnusableEntryException` objects.

Generally speaking, writing a remote reference to a non-persistent remote object into a space is risky. Because entries are stored in serialized form, entries in a `JavaSpaces` server will not participate in the garbage collection that keeps



such references valid. However, if the reference is not persistent because the 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 is no longer valid when it is deserialized. In such a case, the client code must decide whether to `take` the entry out of the space (if the original operation was a `read`), or to `write` the entry back into the JavaSpaces server (if the original operation was a `take`) or to leave the space as it is. This `take` option, of course, assumes that you can find lookup fields that match the offending entry uniquely.

In the 1.2 Java Development Kit (JDK), 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—such as the registry’s host name and the remote object’s name in the registry—rather than putting raw references into the space. This might be a good strategy for other reasons.

2.4 *Templates and Matching*

The lookup operations (`read`, `take`, and the notification requests) 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* in the space, 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 values of two fields *match* if `MarshaledObject`’s `equals` method returns `true` for their `MarshaledObject` 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. JavaSpaces applications do not use `Object`’s `equals` method or any other form of type-specific value matching.

You can write an entry that has a `null`-valued field, but you cannot match explicitly on a `null` value in that field, since `null` signals a wildcard field. If you have a field that may be variously `null` or not in entries, and want to match on whether that field is set, you can set the field to `null` in your entry.



But if you will need to write templates that distinguish between set and un-set values for that field, you will have to add a `Boolean` field that indicates whether the field is set, and use a `Boolean` value for that field in templates.

The type of *E* can 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.

An entry that has no wildcards is a valid template.

2.5 *write*

A `write` places a copy of an entry into the given `JavaSpaces` implementation. The `Entry` passed to the `write` is not affected by the operation. Each `write` operation places a new entry into the specified space, even if the same `Entry` object is used in more than one `write`.

Each field of the entry is independent, and is therefore serialized separately. This means that if two fields of an entry refer to the same object, directly or indirectly, the entry that gets written into the space will have two independent copies of that object. Consequently, when the entry is read back from the space, the reconstituted entry will have independent copies 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.)

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 (as used with respect to a `JavaSpaces` system) 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 `JavaSpaces` application 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 entry as a whole is not serialized when used with `JavaSpaces` implementations.

Each `write` invocation returns a `Lease` object that is `lease` milliseconds long. If the requested time is longer than the space is willing to grant, you will get a lease with a reduced time. When the lease expires, the entry is removed from the space. You will get an `IllegalArgumentException` if the lease time requested is negative.



If a `write` returns without throwing an exception, that entry is committed to the space, possibly within a transaction (§3). If a `RemoteException` is thrown, the `write` may or may not have been successful. If any other exception is thrown, the entry was not written into the space.

Writing an entry into a JavaSpaces implementation may generate notifications to registered objects (§2.9).

2.6 *readIfExists* and *read*

A `read` request will search the JavaSpaces server for an entry that matches (§2.4) the template provided as an `Entry`. If a match is found, a reference to a copy of the matching entry is returned. If no match is found, `null` is returned. Passing a `null` reference for the template will match any entry.

Any matching entry can be returned. Successive `read` requests with the same template in the same JavaSpaces server may or may not return equivalent objects, even if no intervening modifications have been made to the space. Each invocation of `read` may return a new object even if the same entry is matched in the JavaSpaces server.

A `readIfExists` request will return a matching entry, or `null` if there is currently no matching entry in the space. If the only possible matches for the template have conflicting locks from one or more other transactions, the `timeout` value specifies how long the client is willing to wait for interfering transactions to settle before returning a value. If at the end of that time no value can be returned that would not interfere with transactional state, `null` is returned. Note that, due to the remote nature of JavaSpaces implementations, `read` and `readIfExists` may throw a `RemoteException` if the network or server fails prior to the timeout expiration.

A `read` request acts like a `readIfExists` except that it will wait until a matching entry is found or until transactions settle, whichever is longer, up to the timeout period.

In both `read` methods, a timeout of `NO_WAIT` means to return immediately, with no waiting. A timeout of zero means to wait indefinitely until a definitive value can be returned.

(See the discussion in `write` (§2.5) about independent serialization and deserialization of fields. Also see §2.3.1 about `UnusableEntryException`.)



2.7 *takeIfExists* and *take*

The `take` requests perform exactly like the corresponding `read` requests (§2.6), except that the matching entry is removed from the JavaSpaces server. Two `take` operations will never return copies of the same entry, although if two equivalent entries were in the JavaSpaces server the two `take` operations may return equivalent entries.

If a `take` returns a non-null value, the entry has been removed from the space, possibly within a transaction (§3). This modifies the claims to once-only retrieval—A `take` is only considered to be successful if all enclosing transactions commit successfully. If a `RemoteException` is thrown, the `take` may or may not have been successful. If any other exception is thrown, the `take` did not occur, and no entry was removed from the space.

With a `RemoteException`, an entry can be removed from a JavaSpace and yet never returned to the client that performed the `take`, thus losing the entry in between. In circumstances where this is unacceptable, the `take` can be wrapped inside a transaction that is committed by the client when it has the requested entry in hand.

(See the discussion in `write` (§2.5) about independent serialization and deserialization of fields. Also see §2.3.1 about `UnusableEntryException`.)

2.8 *snapshot*

The process of serializing an entry for transmission to a JavaSpaces server will be identical if the same entry is used twice. This is most likely to be an issue with templates that may be used repeated to search for entries with `read` or `take`. The client-side implementations of `read` and `take` cannot reasonably avoid this duplicated effort, since they have no efficient way of checking whether the same template is being used without modification.

The `snapshot` method gives the JavaSpaces implementor a way to reduce the impact of repeated use of the same entry. Invoking `snapshot` with an `Entry` will return another `Entry` object that contains a *snapshot* of the original entry. Using `snapshot` is equivalent to using the unmodified original entry in all operations on the same JavaSpaces server. Modifications to the original entry will not affect the snapshot. You can `snapshot` a null template — `snapshot` may or may not return null given a null template.



An `Entry` object returned from `snapshot` on a particular `JavaSpaces` server is only guaranteed to work with that `JavaSpaces` server. Using the snapshot with any other `JavaSpaces` server will generate an `IllegalArgumentException` unless the other space can use it because of knowledge about the original `JavaSpaces` server. The entry returned from `snapshot` will only be equivalent to the original unmodified object when used with the `JavaSpaces` server. It will be a different object from the original, and may or may not have the same hash code, and `equals` may or may not return `true` when invoked with the original object, even if the original object is unmodified.

A snapshot is only guaranteed to work within the virtual machine in which it was generated. If a snapshot is passed to another virtual machine (for example, in a parameter of an RMI call), using it — even with the same `JavaSpaces` server— may generate an `IllegalArgumentException`.

We expect that a `JavaSpaces` implementation will return a specialized `Entry` object that represents a pre-serialized version of the object, either in the object itself, or as an identifier for the entry that has been cached on the server. Although the client may cache the snapshot on the server it must guarantee that the snapshot returned to the client code is always valid—the implementation may not throw any exception that indicates that the snapshot has become invalid because it has been evicted from a cache. An implementation that uses a server-side cache must, therefore, guarantee that the snapshot is valid as long as it is reachable (not garbage) in the client, such as storing enough information in the client to be able to re-insert the snapshot into the server-side cache.

No other method returns a snapshot. Specifically, the return values of the `read` and `take` methods are not snapshots, and are usable with any `JavaSpaces` implementation.

2.9 *notify*

A `notify` request invoked on a template registers interest in future incoming entries, to the specified `JavaSpaces` server, that match the template. Matching is done as it is for `read`. The `notify` method is a particular registration method under the *Distributed Event Specification*. When matching entries arrive, the specified `RemoteEventListener` will be notified. When you invoke `notify` you provide an upper bound on the lease time, which is how long you want the registration to be remembered by the server. The server decides the actual time for the lease. You will get an `IllegalArgumentException` if the lease



time requested is not `Lease.ANY` and is negative. The lease time is expressed in the standard millisecond units, although actual lease times will usually be of much larger granularity. A lease time of `Lease.FOREVER` is a request for an indefinite lease; if the server chooses not to grant an indefinite lease it will return a bounded (non-zero) lease.

Each `notify` returns an `net.jini.event.EventRegistration` object. When an object is written that matches the template supplied in the `notify` invocation, the listener's `notify` method is invoked, with a `RemoteEvent` object whose `evID` is the value returned by the `EventRegistration` object's `getEventID` method, `fromWhom` being the `JavaSpaces` server, `seqNo` being a monotonically increasing number, and whose `getRegistrationObject` being that passed as the `handback` parameter to `notify`. If you get a notification with a `seqNo` of 103 and the `EventRegID` object's `currentSeqNum` is 100, there will have been two previous calls to you with values 101 and 102 (although you may not have received them, or you may receive notification number 103 more than once).

If the transaction parameter is `null`, the listener will be notified when matching entries are written either under a `null` transaction or when a transaction commits. If an entry is written under a transaction and then taken under that same transaction before the transaction is committed, listeners registered under a `null` transaction will not be notified of that entry.

If the transaction parameter is not `null`, the listener will be notified of matching entries written under that transaction in addition to the notifications it would receive under a `null` transaction. A `notify` made with a non-`null` transaction is implicitly dropped when the transaction completes.

The request specified by a successful `notify` is as persistent as the entries of the space. They will be remembered as long as an un-taken entry would be, until the lease expires, or until any governing transaction completes, whichever is shorter.

The server will make a "best effort" attempt to deliver notifications. The server will retry at least until the notification request's lease expires. Notifications may be delivered in any order.

See the *Distributed Event Specification* for details on the event types.



2.10 Operation Ordering

Operations on a JavaSpaces implementation are unordered. The only view of operation order can be a thread's view of the order of the operations it performs. A view of inter-thread order can be imposed only by cooperating threads that use an application-specific protocol to prevent two or more operations being in progress at a single time on a single JavaSpaces server. Such means are outside the purview of this specification.

For example, given two threads *T* and *U*, if *T* performs a `write` operation and *U* performs a `read` with a template that would match the written entry, the `read` may not find the written entry even if the `write` returns before the `read`. Only if *T* and *U* cooperate to ensure that the `write` returns before the `read` commences would the `read` be ensured the opportunity to find the entry written by *T* (although it still may not do so because of an intervening `take` from a third entity).



The JavaSpaces API uses the package `net.jini.transaction` to provide basic atomic transactions that group multiple operations across multiple JavaSpaces implementations into a bundle that acts as a single atomic operation. JavaSpaces servers are actors in these transactions; the client can be an actor as well, as can any remote object that implements the appropriate interfaces.

Transactions wrap together multiple operations. Either all modifications within the transactions will be applied or none will, whether the transaction spans one or more operations and/or one or more JavaSpaces implementations.

The transaction semantics described here conform to the default transaction semantics defined in the *Transaction Specification*.

3.1 Operations Under Transactions

Any `read`, `write`, or `take` operations that have a `null` transaction act as if they were in a committed transaction that contained exactly that operation. For example, a `take` with a `null` transaction parameter performs as if a transaction was created, the `take` performed under that transaction, and then the transaction was committed. Any `notify` operations with a `null` transaction apply to `write` operations that are committed to the full JavaSpaces implementation.



Transactions affect operations in the following ways:

- ◆ **write**: An entry written is not visible outside its transaction until the transaction successfully commits. If the entry is taken within the transaction, the entry will never be visible outside the transaction and will not be added to the space when the transaction commits. Specifically, the entry will not generate notifications to listeners not registered under the writing transaction. Entries written under a transaction that aborts are discarded.
- ◆ **read**: A `read` may match any entry written under that transaction or in the full JavaSpaces implementation. A JavaSpaces implementation is not required to prefer matching entries written inside the transaction to those in the full JavaSpaces implementation. When read, an entry is added to the set of entries read by the provided transaction. Such an entry may be read in any other transaction to which the entry is visible, but cannot be taken in another transaction.
- ◆ **take**: A `take` matches like a `read` with the same template. When taken, an entry is added to the set of entries taken by the provided transaction. Such an entry may not be read or taken by any other transaction.
- ◆ **notify**: A `notify` performed under a `null` transaction applies to `write` operations that are committed to the full JavaSpaces implementation. A `notify` performed under a non-`null` transaction additionally provides notification of writes performed within that transaction. When a transaction completes, any registrations under that transactions are implicitly dropped. When a transaction commits, any entries that were written under the transaction (and not taken) will cause appropriate notifications for registrations that were made under a `null` transaction.

If a transaction aborts while an operation is in progress under that transaction, the operation will terminate with a `TransactionException`. Any statement made in this chapter about `read` or `take` apply equally to `readIfExists` or `takeIfExists`, respectively.

3.2 Transactions and ACID Properties

The ACID properties traditionally offered by database transactions are preserved in transactions on JavaSpaces systems. The ACID properties are:

- ◆ **Atomicity**: All the operations grouped under a transaction occur or none of them do.



- ◆ *Consistency*: The completion of a transaction must leave the system in a consistent state. Consistency includes issues known only to humans, such as that an employee should always have a manager. The enforcement of consistency is outside of the transaction—a transaction is a tool to allow consistency guarantees, and not itself a guarantor of consistency.
- ◆ *Isolation*: Ongoing transactions should not affect each other. Any observer should be able to see other transactions executing in some sequential order (although different observer may see different orders).
- ◆ *Durability*: The results of a transaction should be as persistent as the entity on which the transaction commits.

The timeout values in `read` and `take` allow a client to trade full isolation for liveness. For example, if a `read` request has only one matching entry, and that entry is currently locked in a `take` from another transaction, `read` would block indefinitely if the client wanted to preserve isolation. Since completing the transaction could take an indefinite amount of time, a client may choose instead to put an upper bound on how long it is willing to wait for such isolation guarantees, and instead proceed to either abort its own transaction or ask the user whether to continue or whatever else is appropriate for the client.

Persistence is not a required property of JavaSpaces implementations. A transient JavaSpaces implementation that does not preserve its contents between system crashes is a proper implementation of the `JavaSpace` contract, and may be quite useful. If you choose to perform operations on such a space, your transactions will guarantee as much durability as the JavaSpaces implementation allows for all its data, which is all that any transaction system can guarantee.



References and Further Reading



4.1 References

- [1] *A Note on Distributed Computing*, Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. Sun Microsystems Laboratories technical report SMLI TR-94-29, <http://www.sunlabs.com/technical-reports/1994/abstract-29.html>
- [2] *Java Remote Method Invocation Specification*, <http://java.sun.com/products/jdk/rmi>
- [3] *Java Object Serialization Specification*, <http://java.sun.com/products/jdk/rmi>
- [4] *Jini Event and Notification Specification*
- [5] *Jini Leasing Specification*
- [6] *Jini Transaction Specification*
- [7] *Jini Distributed Security Specification*

4.2 Further Reading

4.2.1 Linda Systems

- [8] *How to Write Parallel Programs: A Guide to the Perplexed*, Nicholas Carriero and David Gelernter, *ACM Computing Surveys*, Sept., 1989.



-
- [9] *Generative Communication in Linda*, David Gelernter, *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, pp. 80-112 (January 1985)
 - [10] *Persistent Linda: Linda + Transactions + Query Processing*, Brian G. Anderson, Dennis Shasha,
 - [11] *Adding Fault-tolerant Transaction Processing to LINDA*, Scott R. Cannon, David Dunn, *Software—Practice and Experience*, Vol. 24(5), pp. 449-446 (May 1994)
 - [12] *ActorSpaces: An Open Distributed Programming Paradigm*, Gul Agha, Christian J. Callsen, University of Illinois at Urbana-Champaign, UILU-ENG-92-1846,

4.2.2 *The Java Platform*

- [13] *The Java Language Specification*, James Gosling, Bill Joy, Guy Steele, Addison-Wesley
- [14] *The Java Virtual Machine Specification*, Tim Lindholm, Frank Yellin, Addison-Wesley
- [15] *The Java Class Libraries*, Patrick Chan, Rosanna Lee, Addison-Wesley

4.2.3 *Distributed Computing*

- [16] *Distributed Systems*, Sape Mullender, Addison-Wesley
- [17] *Distributed Systems: Concepts and Design*, George Coulouris, Jean Dollimore, Tim Kindberg, Addison-Wesley
- [18] *Distributed Algorithms*, Nancy A. Lynch, Morgan Kaufmann