Content Repository for Java™ Technology API 2.0

Specification

JCR 2.0 Specification
Java Specification Request (JSR) 283
10 August 2009

1 Preface

The Content Repository API for Java™ Technology Specification, Version 2.0 (JCR 2.0 Specification) consists of a normative part and a non-normative part.

The normative part consists of:

- This document, excluding the appendix.
- The source code of `javax.jcr` and its subpackages.
- The Javadoc reference.

In case of a conflict this document takes precedence over the source code and the source code takes precedence over the Javadoc.

The non-normative part consists of:

- The appendix of this document.
- The `jar` file of `javax.jcr` and its subpackages.

The JCR 2.0 Specification was created and released through the Java Community Process (JCP) under Java Specification Request 283 (JSR 283).

1.1 Previous Versions

The Content Repository for Java™ Technology API Specification, Version 1.0 (JCR 1.0 Specification) was created and released through the Java Community Process (JCP) under Java Specification Request 170 (JSR 170).

1.2 Coverage

This document describes the abstract repository model and Java API of JCR. The API is described from a high-level, functional perspective. Consult the accompanying Javadoc for full information on signature variants and exceptions.

1.2.1 Classes and Interfaces

Unless otherwise indicated, all Java classes and interfaces mentioned are in the package `javax.jcr` and its subpackages. Non-JCR classes mentioned are always fully qualified. The only exception is `java.lang.String`, which is used throughout and written simply as `String`.

1.2.2 Null Parameters

When describing JCR API methods, this specification and the Javadoc assume that all parameters passed are non-`null`, unless otherwise stated. If `null` is passed as parameter and its behavior is not explicitly described in this specification or in the Javadoc, then the behavior of the method in that case is implementation-specific.

1.3 Typographical Conventions

A `monospaced font` is used for JCR names and paths, and all instances of machine-readable text (Java code, XML, grammars, JCR-SQL2 examples, URIs, etc.).

1.3.1 String Literals in Syntactic Grammars

Formal grammars are used at various places in the specification to define the syntax of string-based entities such as names, paths, search languages and other notations.

When a *string literal* appears as a terminal symbol within a grammar, each character literal in that string corresponds to exactly one Unicode code point.

The intended code point of such a character literal must be determined only by reference to the Unicode Basic Latin code chart[1] and no other part of the Unicode character set.

Any code point outside the Basic Latin set *cannot* be the intended code point of such a character literal, even if the grapheme of the code point superficially resembles that of the character literal.

For example, in the following production (excerpted from §3.2.2 *Local Names*).

```
InvalidChar ::= '/' | ':' | '[' | ']' | '|' | '*'
```

The code points indicated by the character literals are, respectively, `U+002F` ("`/`"), `U+003A` ("`:`"), `U+005B` ("`[`"), `U+005D` ("`]`"), `U+007C` ("`|`") and `U+002A` ("`*`").

1.4 System Requirements

The JCR 2.0 requires Java Runtime Environment (JRE) 1.4 or greater.

1.5 License

---

[1] See `http://unicode.org/charts/PDF/U0000.pdf`.

NOTICE; LIMITED LICENSE GRANTS

1. License for Purposes of Evaluation and Developing Applications. Licensor hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under Licensor's applicable intellectual property rights to view, download, use and reproduce the Specification only for the purpose of internal evaluation. This includes developing applications intended to run on an implementation of the Specification provided that such applications do not themselves implement any portion(s) of the Specification.

2. License for the Distribution of Compliant Implementations. Licensor also grants you a perpetual, non-exclusive, non-transferable, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights or, subject to the provisions of subsection 4 below, patent rights it may have covering the Specification to create and/or distribute an Independent Implementation of the Specification that: (a) fully implements the Specification including all its required interfaces and functionality; (b) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented; and (c) passes the Technology Compatibility Kit (including satisfying the requirements of the applicable TCK Users Guide) for such Specification ("Compliant Implementation"). In addition, the foregoing license is expressly conditioned on your not acting outside its scope. No license is granted hereunder for any other purpose (including, for example, modifying the Specification, other than to the extent of your fair use rights, or distributing the Specification to third parties).

3. Pass-through Conditions. You need not include limitations (a)-(c) from the previous paragraph or any other particular "pass through" requirements in any license You grant concerning the use of your Independent Implementation or products derived from it. However, except with respect to Independent Implementations (and products derived from them) that satisfy limitations (a)-(c) from the previous paragraph, You may neither: (a) grant or otherwise pass through to your licensees any licenses under Licensor's applicable intellectual property rights; nor (b) authorize your licensees to make any claims concerning their implementation's compliance with the Specification.

4. Reciprocity Concerning Patent Licenses. With respect to any patent claims covered by the license granted under subparagraph 2 above that would be infringed by all technically feasible implementations of the Specification, such license is conditioned upon your offering on fair, reasonable and non-discriminatory terms, to any party seeking it from You, a perpetual, non-exclusive, non-transferable, worldwide license under Your patent rights that are or would be infringed by all technically feasible implementations of the Specification to develop, distribute and use a Compliant Implementation.

5. Definitions. For the purposes of this Agreement: "Independent Implementation" shall mean an implementation of the Specification that neither derives from any of Licensor's source code or binary code materials nor, except

with an appropriate and separate license from Licensor, includes any of Licensor's source code or binary code materials; "Licensor Name Space" shall mean the public class or interface declarations whose names begin with "java", "javax", "javax.jcr" or their equivalents in any subsequent naming convention adopted by Licensor through the Java Community Process, or any recognized successors or replacements thereof; and "Technology Compatibility Kit" or "TCK" shall mean the test suite and accompanying TCK User's Guide provided by Licensor which corresponds to the particular version of the Specification being tested.

6. Termination. This Agreement will terminate immediately without notice from Licensor if you fail to comply with any material provision of or act outside the scope of the licenses granted above.

7. Trademarks. No right, title, or interest in or to any trademarks, service marks, or trade names of Licensor is granted hereunder. Java is a registered trademark of Sun Microsystems, Inc. in the United States and other countries.

8. Disclaimer of Warranties. The Specification is provided "AS IS". LICENSOR MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT (INCLUDING AS A CONSEQUENCE OF ANY PRACTICE OR IMPLEMENTATION OF THE SPECIFICATION), OR THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE. This document does not represent any commitment to release or implement any portion of the Specification in any product.

The Specification could include technical inaccuracies or typographical errors. Changes are periodically added to the information therein; these changes will be incorporated into new versions of the Specification, if any. Licensor may make improvements and/or changes to the product(s) and/or the program(s) described in the Specification at any time. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

9. Limitation of Liability. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL LICENSOR BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

10. Report. If you provide Licensor with any comments or suggestions in connection with your use of the Specification ("Feedback"), you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Licensor a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for

any purpose related to the Specification and future versions, implementations, and test suites thereof.

1.6 Acknowledgements

The following people and organizations have contributed to this specification:

David Nuescheler (Specification Lead)

Peeter Piegaze (Principal Author)

Razmik Abnous
Tim Anderson
Gordon Bell
Tobias Bocanegra
Al Brown
Dave Caruana
Geoffrey Clemm
David Choy
Jeff Collins
Cornelia Davis
Chenggang Duan
Roy Fielding
Xaver Fischer
Gary Gershon
Stefan Guggisberg
Florent Guillaume
Berry van Halderen
Rich Howarth
Jens Huebel
Volker John
Alison Macmillan
Ryan McVeigh
Stefano Mazzocchi
James Myers
John Newton
James Owen
Franz Pfeifroth
David Pitfield
Nicolas Pombourcq
Corprew Reed
Julian Reschke
Marcel Reutegger
Celso Rodriguez
Steve Roth
Angela Schreiber
Victor Spivak
Paul Taylor
David B. Victor
Dan Whelan
Kevin Wiggen
Jukka Zitting

Alfresco
Apache Software Foundation
BEA
Day Software
Documentum
EMC
FileNet
Fujitsu
Greenbytes
Hippo
Hummingbird
IBM
Imerge
Intalio

Mobius	Pacific Northwest National Laboratory
Nuxeo	Saperion
Opentext	Vignette
Oracle	Xythos

2 Introduction

The JCR specification defines an abstract model and a Java API for data storage and related services commonly used by content-oriented applications. The target domain encompasses not only traditional content management systems but, more broadly, any application that must handle both unstructured digital assets and structured or semi-structured information.

The repository model enables efficient access to both large binary objects and finely-structured hierarchical data through a simple, generic API and a robust and extensible object typing system. Additionally, many features that are traditionally custom-built on top of RDBMSs and file systems have been incorporated into the repository itself. These include support for query, access control, versioning, locking and observation. Standardized support for these services further enables applications that might not normally have access to such advanced features to take advantage of them, since they are built-in at the infrastructure level.

3 Repository Model

This section describes the objects, types and structures that compose a JCR repository. The description is language-neutral and focuses on the static aspects of the data model. Discussion of the behavioral aspects of the repository, and in particular the Java API for performing operations on the model, is found in subsequent sections. The full repository model is described here, though an implementation may support only a subset of this model, in accordance with §24 *Repository Compliance.*

3.1 Overview

3.1.1 Persistent Workspaces

A JCR *repository* is composed of one or more *persistent workspaces*, each consisting of a directed acyclic graph of *items* where the edges represent the parent-child relation.

Each persistent workspace is identified by a unique name within the repository, which is a string.

3.1.2 Items

An item is either a *node* or a *property*. A node can have zero or more child items. A property cannot have child items but can hold zero or more *values*.

The nodes of a workspace form the structure of the stored data while the actual content is stored in the values of the properties.

Each workspace contains at least one item, the *root node*. The root node is the only item in the workspace without a parent node; all other items have at least one parent.

3.1.2.1 Shared Nodes

In the simplest case, a workspace is a *tree* of items. However, strictly speaking, the more general term *graph* should be used to cover those cases where a repository supports the optional *shareable nodes* feature, which allows an item to have more than one parent (see §3.9 *Shareable Nodes Model*).

3.1.3 Names

The name of the root node of a workspace is always "" (the empty string). Every other item in a workspace has a name, which must be a *JCR name* (see §3.2 *Names*).

3.1.3.1 Same-Name Siblings

In the simplest case, every child item of a given parent has a unique name. However, child nodes with identical names can only occur if a repository supports *same-name siblings* (see §22 *Same-Name Siblings*). Additionally, some repositories may support a node and sibling property having the same name (see

§5.1.8 *Node and Property with Same Name*). However, two sibling properties can never have the same name.

To distinguish sibling nodes with the same name an integer index, starting at 1, is used. A node with no same-name siblings has an implicit index of 1 and a node name without an index is understood to have an index of 1.

3.1.4 Paths

The location of an item in the workspace graph can be described by the path from the root node to that item. The path consists of the name (and index in cases of same-name siblings) of each interceding node in order from root to target item, much like a file system path. Relative paths can also be used to describe the location of one item with respect to another (see §3.4 *Paths*).

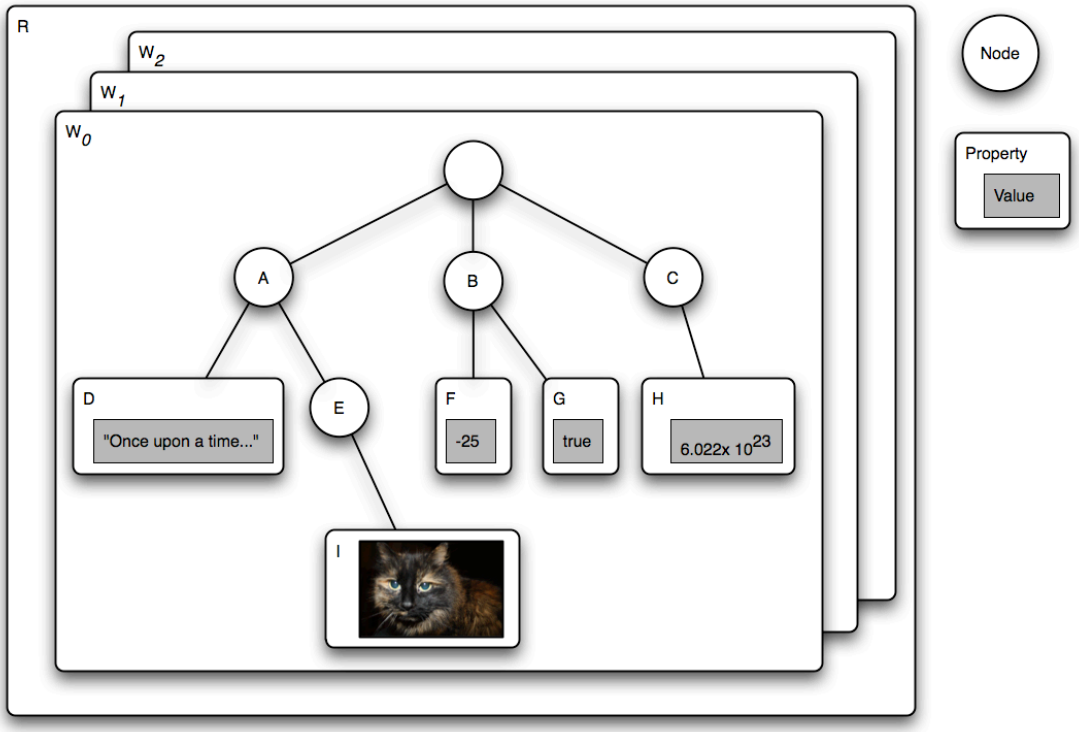
3.1.5 Identifiers

In addition to a path, every node also has an identifier. In some implementations, the identifier may be independent of the path and provide an identity to the node that is stable across moves within the workspace. In simpler repositories the identifier may be implemented as a reflection of the path and therefore not provide any additional semantics (see §3.3 *Identifiers*).

3.1.6 Property Types

Properties can be either single or multi-valued. Each value has one of the 12 possible types (see §3.6 *Properties*). These types include familiar data storage types such as strings, numbers, booleans, binaries and dates, as well as types that hold pointers to other nodes in the workspace.

3.1.6.1 Repository Diagram



The above diagram depicts a repository `R` with workspaces `W0`, `W1` and `W2`. The item graph of `W1` contains a root node with child nodes `A`, `B` and `C`. `A` has a property `D` of type `STRING` and a child node `E`, which in turn has a property `I` of type `BINARY`. `B` has the properties `F` (a `LONG`) and `G` (a `BOOLEAN`). `C` has a property `H` of type `DOUBLE`.

3.1.7 Node Types

Every node has a type. A node's type The names, types and other attributes of its child items. Node types can be used to define complex storage objects consisting of multiple subnodes and properties, possibly many layers deep.

3.1.8 Sessions

A *user* connects to a repository by passing a set of credentials and the name of the workspace that the user wishes to access. The repository returns a *session* which binds the user to the requested persistent workspace with a level of authorization determined by that user's credentials. A session is always bound to exactly one persistent workspace, though a single persistent workspace may be bound to multiple sessions.

3.1.8.1 User

A user is any agent bound to a session. This may be a human user, an external software process, or anything else that holds and controls the session.

3.1.8.2 Current Session and Workspace

Through a session, the user can access, read and write the nodes and properties of the bound workspace, to the extent allowed by that user's authorization and the capabilities of the repository. Any object acquired, directly, or indirectly through a chain of interceding objects, from a particular session, is said to be within the scope of that session and any method called on such object is also within the scope of the same session.

In the context of discussing a particular object or method call, the session within whose scope that object or method call lies is referred to as the *current session*, and the workspace to which that session is bound is referred to as the *current workspace*.

3.2 Names

A *JCR name* is an ordered pair of strings:

(*N*, *L*)

where *N* is a *JCR namespace* and *L* is a *JCR local name*.

3.2.1 Namespaces

A *JCR namespace* is either the empty string or a Universal Resource Identifier[2].

```
Namespace ::= EmptyString | Uri

EmptyString ::= /* The empty string */

Uri ::= /* A URI, as defined in Section 3 in
           http://tools.ietf.org/html/rfc3986#section-3 */
```

3.2.2 Local Names

A *JCR local name* is a string that conforms to the grammar below.

```
LocalName ::= ValidString - SelfOrParent
              /* Any ValidString except SelfOrParent */

SelfOrParent ::= '.' | '..'

ValidString ::= ValidChar {ValidChar}

ValidChar ::= XmlChar - InvalidChar
              /* Any XmlChar except InvalidChar */

InvalidChar ::= '/' | ':' | '[' | ']' | '|' | '*'

XmlChar ::= /* Any character that matches the Char production
               at http://www.w3.org/TR/xml/#NT-Char */
```

[2] See `http://tools.ietf.org/html/rfc3986#section-3`.

```
/* See §1.3.1 String Literals in Syntactic Grammars for details
   on the interpretation of string literals in this grammar */
```

3.2.3 Use of JCR Names

JCR names are used to name items, node types and other entities throughout the repository.

3.2.3.1 Item Names

Every item has one JCR name. If the item has more than one parent it has the same name relative to each, though in most cases an item will have only one parent (see §3.9 *Shareable Nodes Model*).

3.2.3.2 Paths

JCR names are combined into JCR paths which indicate the location of an item within a workspace either in relation to the root node or relative to another item in the workspace (see §4.5 *Workspace*).

3.2.3.3 NAME and PATH Values

JCR names appear as the values of `NAME` properties (see §3.6.1.9 *NAME*) and within the values of `PATH` properties (see §3.6.1.10 *PATH*).

3.2.3.4 Node Types

JCR names are used to name node types (see §3.7 *Node Types*).

3.2.3.5 Constants

JCR names are used to identify other types of entities such as *privileges*, *access control policies* (see §16 *Access Control Management*), *retention policies*, *holds* (see §20 *Retention and Hold*), *activities* (see §15.12 *Activities*) and *configurations* (see §15.13 *Configurations and Baselines*).

3.2.4 Naming Restrictions

This definition of *JCR name* represents the *least restrictive* set of constraints permitted for the naming of items and other entities. A repository *may* further restrict the names of entities to a subset of JCR names and in most cases is encouraged to do so.

In a read-only repository, any such restrictions will stem trivially from the fact that the repository controls the set of entity names exposed. A writable repository *may* enforce any implementation-specific constraint by causing an exception to be thrown on an invalid JCR write method call.

The characters declared invalid within a local name (“`/`”, “`:`”, “`[`”, “`]`”, “`|`”, “`*`”) represent only those characters which are used as metacharacters in JCR names, paths and name-matching patterns (see §5.2.2 *Iterating Over Child Items*). These restrictions are not necessarily sufficient to enforce best practices in the creation of JCR names. In particular, the minimal grammar defined here permits

JCR names with leading and trailing whitespace as well as characters which may appear superficially identical while representing different code points, creating a potential security issue.

Though this specification does not attempt to define good naming practice, implementers are discouraged from permitting names with these and other problematic characteristics when possible. However, there may be cases where the latitude provided by the minimal grammar is useful, for example, when a JCR implementation is built on top of an existing data store with an unconventional naming scheme.

3.2.5 Lexical Form of JCR Names

While a JCR name is an ordered pair of strings, (*N*, *L*), it is not itself a string. There are, however, two lexical forms (string serializations) that a JCR name can take when used in the JCR API: the *expanded form* and the *qualified form*. A JCR name that is converted to either lexical form is said to have been *lexicalized*.

3.2.5.1 Expanded Form

The expanded form of a JCR name is defined as:

```
ExpandedName ::= '{' Namespace '}' LocalName

Namespace ::= /* see §3.2.1 Namespaces */

LocalName ::= /* see §3.2.2 Local Names */

/* See §1.3.1 String Literals in Syntactic Grammars for details
   on the interpretation of string literals in this grammar */
```

3.2.5.2 Qualified Form

The qualified form of a JCR name is defined as:

```
QualifiedName ::= [Prefix ':'] LocalName

Prefix ::= /* Any string that matches the NCName production in
              http://www.w3.org/TR/REC-xml-names */

LocalName ::= /* see §3.2.2 Local Names */

/* See §1.3.1 String Literals in Syntactic Grammars for details
   on the interpretation of string literals in this grammar */
```

A qualified name is only interpretable in the context of a *namespace mapping*, which provides a one-to-one mapping between prefixes and namespaces.

When a qualified name *Q* is passed to a JCR method within the scope of the `Session` *S* then the JCR name *J* represented by *Q* is *(N, L)* where *N* is the *namespace* corresponding to *P* in the *local namespace mapping* of *S*. See §3.4 *Namespace Mapping*.

When a qualified name occurs in a string serialization of repository content or a node type definition, the namespace mapping is either provided within the

serialized form (see, for example, §7 *Export* and §25.2 *Compact Node Type Definition Notation*) or implied by the context of use.

3.2.5.3 Qualified Form with the Empty Namespace

The qualified form of a name (“”, *L*) (i.e., with the empty string as namespace) is not written as

:*L*

but simply as

L

The former is not a valid qualified JCR name.

3.2.5.4 Exposing Non-JCR Names

An implementation that exposes a non-JCR data store through the JCR API may wish to expose names containing JCR-illegal characters by using a substitution or escaping scheme. If so, it must do so by substituting private-use Unicode characters for the JCR-illegal characters according to the following mapping.

JCR-Illegal character	**Substitution character**
* (U+002A)	U+F02A
/ (U+002F)	U+F02F
: (U+003A)	U+F03A
[(U+005B)	U+F05B
] (U+005D)	U+F05D
| (U+007C)	U+F07C

The mapping must be used bi-directionally. When the repository wishes to return the name of an entity whose native name contains a JCR-illegal character, that character must be replaced with its corresponding substitution character in the returned string.

Conversely, when a name containing one of the substitution characters is passed to the repository through the JCR API, that character must be replaced with its corresponding non-JCR character before further processing is done within the native layer, whether writing the name to storage or using the name to access an entity.

In the unlikely event that one of the substitution characters appears literally in a native name, that character will be returned unchanged through the JCR API.

In repositories that do not expose non-JCR names and therefore do not need to use the substitution scheme, any private-use substitution character passed to the

API is stored and returned unchanged. However, such use of a private-use substitution character within a JCR name is strongly discouraged.

3.2.6 Use of Qualified and Expanded Names

When a JCR name is passed as an argument to a JCR method it may be in either expanded or qualified form. When a repository returns a JCR name it must be in qualified form. The qualified form of a name depends upon the prevailing local namespace mapping of the current session (see §3.5 *Namespace Mapping*).

3.2.7 Equality of Names

Two JCR names (N_1, L_1) and (N_2, L_2) are *equal* if and only if N_1 is equal to N_2 and L_1 is equal to L_2, according to the definition of string equality used in the `String.compareTo` method. This definition applies both in the general context of using an API method that takes or returns a JCR name and in the specific case of comparing values of type `NAME` (see §3.6.5.8 *NAME*).

3.3 Identifiers

Every node has an *identifier*. An identifier is a string whose format is not defined by this specification but which adheres to the following constraints:

- The identifier of a non-shared node is unique within a workspace. The identifier of a shared node is common to each member of that node's share-set (see §3.9 *Shareable Nodes Model*).
- An identifier *must* be the most stable one available to the implementation. For example, in some implementations this might be nothing more than the node path itself. Other implementations might support node identifiers that are partly or entirely independent of the path.

3.3.1 Identifier Assignment

The identifier must be assigned at the latest when the node is first persisted, though it may be assigned earlier, when the node is first created in transient storage in the session (see §10.4.1 *Adding a Node*).

3.3.2 Referenceable Identifiers

In implementations that support referenceable nodes, these nodes have more stringent requirements on their identifiers (see §3.8 *Referenceable Nodes*).

3.3.3 Correspondence by Identifier

Identifiers are also used for *node correspondence* across multiple workspaces (see §3.10 *Corresponding Nodes*).

3.4 Paths

A *JCR path P*,

$$P = (S_0, S_1, ..., S_n),$$

is an ordered list with at least one element, where each element S_i, for $0 \le i \le n$, is a *path segment*.

3.4.1 Path Segment

A path segment is one of:

- a *name segment*, (J, I), where J is a JCR name and I is an integer index ($I \ge 1$).
- an *identifier segment*, U, where U is a JCR identifier.
- the *root segment*.
- the *self segment*.
- the *parent segment*.

The root, self and parent segments are logical constants distinct from each other and from all name segments.

3.4.1.1 Position of Segments in a Path

Name, self and parent segments can occur at any position in a path.

A root segment can occur only as the first segment of a path.

An identifier segment can occur only as the first and sole segment in a path. No other segments may follow an identifier segment.

3.4.2 Path Resolution

The successive path segments of JCR path $P = (S_0, S_1, ..., S_n)$ define a route through workspace W to a *target* item as follows:

- If S_0 is the root segment then the path is absolute and the *current* item is the root node of W.
- If S_0 is an identifier segment U, then the path is absolute and the *current* item is the node in W with the identifier U.
- Otherwise, the path is relative and the *current* item is determined by the context of use.
- For each segment S in path P:
    - If S is a self segment then the *current* item does not change.
    - If S is a parent segment then the new *current* item is a parent of the old *current* item (see §3.4.2.1 *Parent Resolution*).
    - If S is a name segment then the new current item is the child of the old *current* item identified by S (see §3.4.2.2 *Child Resolution*).
- Once all segments have been traversed, the *current* item is the *target* item.

3.4.2.1 Parent Resolution

In most cases an item will have only one parent, in such a case, parent resolution is trivial. In repositories that support shareable nodes, a node may share its child nodes and properties with other nodes. A child item of a shared node therefore has more than one parent. In such a case the parent resolved depends upon the *deemed path* of the item, which is an implementation-specific issue (see §3.9.5 *Deemed Path*). An attempt to resolve the parent of a workspace root node always fails.

3.4.2.2 Child Resolution

Given a name segment *S* = (*J*, *I*), *J* is the name of the child item indicated by that segment while *I* indicates the index of the item. The index is an integer greater than or equal to `1` and is used to distinguish between sibling child nodes with the same name. If there is only one child node with the name *J* then its index is always `1`. If there is more than one node with the name *J* then each has a unique index (see §22 *Same-Name Siblings*).

The child item indicated by *S* is determined as follows:

- If the *S is not* the last segment of the path then, if a child node with name *J* and index *I* exists, *S* resolves to that node. Otherwise, resolution fails.
- If *S is* the last segment of the path then,
    - if *S* is constrained to resolve to a *gettable node* (as in the case of `Node.getNode`) and a child node with name *J* and index *I* is retrievable, *S* resolves to that node. Otherwise,
    - if *S* is constrained to resolve to an *addable node* (as in the case of `Node.addNode`) and a child node named *J* can be legally added and *I* is equal to 1, then *J* is used as the name of the new node which, if necessary, is given an appropriate index. Otherwise,
    - if *S* is constrained to resolve to a *gettable property* (as in the case of `Node.getProperty`) then, if a property with name *J* is retrievable, and *I* is equal to 1, *S* resolves to that property. Otherwise,
    - if *S* is constrained to resolve to a *settable property* (as in the case of `Node.setProperty`) then, if a property with name *J* or if a property named *J* can be legally added, and *I* is equal to 1, *S* resolves to that property. Otherwise,
    - if *S* is constrained to resolve to a *gettable item* (as in the case of `Session.getItem`) then if a node with name *J* and index *I* is retrievable, *S* resolves to that node. Otherwise, if there exists a property with name J and *I* is equal to 1, then *S* resolves to that property.
    - Otherwise, resolution fails.

3.4.3 Lexical Forms

Given a JCR path `P = (S0, S1, ..., Sn)`, its lexical form `L` can be constructed according to the following algorithm, where `=` is the assignment operator, `+=` is the string append operator, `or` indicates an arbitrary choice between alternative operations and `nothing` is the null operation.

```
L = ""
for each S in P
  if S is the root segment
    L += "/"
  else
    if S is an identifier segment U
      L += "[" + U + "]"
    else if S is a self segment
      L += "."
    else if S is a parent segment
      L += ".."
    else if S is a name segment (J, I)
      L += the qualified form of J
      or L += the expanded form of J //optional syntax
      if I > 1
        L += "[" + I + "]"
      else
        nothing
        or L += "[1]" //optional syntax
      end if
    end if
    if S is not the last segment of P
      L += "/"
    else
      nothing
      or L += "/" //optional syntax
    end if
  end if
end for
```

The resulting `L` is a lexical form of `P`. As indicated by the steps marked *optional syntax*, a JCR path may have multiple equivalent lexical forms depending on the use of qualified vs. expanded names, the optional `[1]` index indicator and the optional trailing forward slash ("`/`").

3.4.3.1 Standard Form

A string constructed without any of the optional syntax shown in the algorithm is called the *standard form* of a JCR path. Such a lexical path has the following characteristics:

- It consists of either one identifier segment or one or more name segments.
- All name segments are in qualified form, none are in expanded form.
- No name segment has a `[1]` index.
- There is no trailing forward slash ("`/`").

The following are examples of standard form lexical paths:

- `/`
- `/ex:document`
- `/ex:document/ex:paragraph[2]`
- `[f81d4fae-7dec-11d0-a765-00a0c91e6bf6]`

3.4.3.2 Non-Standard Form

A string constructed *with one or more optional steps* is a *non-standard form* JCR path. A non-standard form lexical path has *at least one* of the following features:

- One or more name segments are in expanded form.
- One or more name segments has a `[1]` index.
- The path has a trailing forward slash ("`/`").

The following are examples of non-standard form lexical paths:

- `/ex:document[1]`
- `/ex:document/`
- `/{http://example.com/ex}document/ex:paragraph[2]`

3.4.3.3 Lexical Path Grammar

A JCR path in lexical form conforms to the following grammar

```
Path ::= AbsolutePath | RelativePath

AbsolutePath ::= '/' [RelativePath] | '[' Identifier ']'

RelativePath ::= [RelativePath '/'] PathSegment ['/']

PathSegment ::= ExpandedName [Index] |
                QualifiedName [Index] |
                SelfOrParent

Index ::= '[' Number ']'

Identifier ::= /* See §3.3 Identifiers */

Number ::= /* An integer > 0 */

ExpandedName ::= /* See §3.2.5.1 Expanded Form */

QualifiedName ::= /* See §3.2.5.2 Qualified Form */

SelfOrParent ::= /* see §3.2.2 Local Names */

/* See §1.3.1 String Literals in Syntactic Grammars for details
   on the interpretation of string literals in this grammar */
```

3.4.3.4 Parsing Lexical Paths

When parsing a lexical path, the parser must distinguish between name segments that are in expanded form and those that are in qualified form (see §3.2.5 *Lexical Form of JCR Names*). When making this determination, the repository cannot assume that every namespace URI encountered in an expanded name will be registered within the repository.

An otherwise valid path containing an expanded name with an unregistered *namespace URI* will always resolve into a valid internal representation of a path (i.e., an ordered list of path segments, see §3.4 *Paths*). Any errors that arise from passing such a path must therefore be as a result of further processing (not merely parsing) that depends on the semantics of the path and the context of use.

However, a path containing a qualified name with an unregistered *prefix* will *not* resolve into a valid internal path representation. An attempt to pass such a path will therefore fail at the parsing stage.

3.4.4 Absolute and Relative Paths

An abstract JCR path is either *absolute* or *relative*.

3.4.4.1 Absolute Path

An absolute JCR path is either *root-based* or *identifier-based*.

3.4.4.1.1 Root-Based Absolute Paths

A root-based absolute path begins with the root segment. Its lexical form therefore begins with a forward slash, for example,

```
/A/B/C
```

3.4.4.1.2 Identifier-Based Absolute Paths

An identifier-based absolute path consists of a single identifier segment. Its lexical form therefore consists of square brackets delimiting an identifier, for example,

```
[f81d4fae-7dec-11d0-a765-00a0c91e6bf6]
```

3.4.4.2 Relative Path

A relative JCR path is one which begins with a segment that is neither a root segment nor an identifier segment. Its lexical form therefore begins with either a JCR name, `..` or `.`, for example,

```
D/E/F
```

or

```
../E/F/G
```

3.4.5 Normalized Paths

A JCR path is normalized by the following steps:

- All self segments are removed.
- All redundant parent segments are collapsed. A redundant parent segment is one which can be removed by also removing a preceding name segment while preserving the location indicated by the path. For example, the path `/A/B/C/../..` can be collapsed to `/A`. Note therefore, that if a normalized path contains any parent segments, they must all precede the first name segment.
- If the path is an identifier-based absolute path, it is replaced by a root-based absolute path that picks out the same node in the workspace as the identifier it replaces.

3.4.6 Passing Paths

When a JCR path is passed as an argument to a JCR method it may be normalized or non-normalized and in standard or non-standard form.

3.4.7 Returning Paths

When a repository returns a JCR path it must be normalized (see §3.4.5 *Normalized Paths*), unless the repository is returning the value of a `PATH` property, in which case the original, possibly non-normalized form of the path is preserved and returned. In all cases the returned path must be in standard form (see §3.4.3.1 *Standard Form*).

3.4.8 Equality of Paths

Two types of path equality are defined: *segment equality* and *semantic equality*.

3.4.8.1 Segment Equality

Two paths P_1 and P_2 are segment-equal if and only if:

- They contain the same number of segments.
- Each segment in P_1 is equal to the segment at the same position in P_2.

Two name segments are equal if and only if their JCR names are equal (see §3.2.7 *Equality of Names*) and their integer indexes are equal.

Equality for identifier segments is as defined for identifiers in general, that is, by standard Java `String` equality.

Equality for root, self and parent segments is simple type identity; every instance of a root, parent or self segment is equal to every other instance of the same type.

3.4.8.2 Semantic Equality

For two paths P_1 and P_2 semantic equality is defined as follows:

- If P_1 and P_2 are normalized then they are semantically equal if and only if they are *segment-equal.*
- If P_1 and P_2 are non-normalized then they are semantically equal if and only if their normalized forms are semantically equal.

3.4.8.3 Application of Path Equality

When a JCR path is passed to a JCR API method that must resolve that path the applicable definition of path equality is that of semantic equality. Semantic equality of two paths means that, given identical contexts, the two paths will resolve to the same item.

However, values of type `PATH` are not normalized upon storage or retrieval, so the when comparing two such values, the applicable definition of equality is that of segment equality. (see §3.6.5.9 *PATH*).

3.5 Namespace Mapping

For compactness and legibility in documentation, XML and Java code, JCR names are usually expressed in qualified form.

The use of qualified form, however, depends upon a context that supplies a mapping from prefix to namespace. In documentation this context is provided either by convention or explicit statement. In XML serialization it is supplied by `xmlns` attributes (see §7 *Export*) and in a running JCR repository is provided by the local namespace mapping of each individual `Session`.

3.5.1 Namespace Registry

The local namespace mapping of a session is determined by the initial set of mappings copied from the *namespace registry* and any session-local changes made to that set.

The namespace registry is a single, persistent, repository-wide table that contains the default namespace mappings. It may contain namespaces that are not used in repository content, and there may be repository content with namespaces that are not included in the registry. The namespace registry always contains at least the following built-in mappings between prefix (on the left) and namespace (on the right):

1. `jcr = http://www.jcp.org/jcr/1.0`
   Reserved for items defined within built-in node types (see §3.7 *Node Types*).
2. `nt = http://www.jcp.org/jcr/nt/1.0`
   Reserved for the names of built-in primary node types.
3. `mix = http://www.jcp.org/jcr/mix/1.0`
   Reserved for the names of built-in mixin node types.
4. `xml = http://www.w3.org/XML/1998/namespace`
   Reserved for reasons of compatibility with XML.

5. *(the empty string)* = *(the empty string)*
   The default namespace is the *empty namespace*.

3.5.1.1 Empty Prefix and Empty Namespace

The permanent default namespace in JCR is the empty string, also referred to as the *empty namespace*. This permanence is reflected in the immutable default namespace mapping in the namespace registry. By definition, the prefix in this mapping is the empty string, also referred to as the *empty prefix*.

3.5.1.2 Additional Built-in Namespaces

A repository may provide additional built-in mappings other than those defined in this section. All mappings must be one-to-one, meaning that for a given namespace in the registry exactly one prefix is mapped to it, and for a given prefix in the registry exactly one namespace is mapped to it.

3.5.2 Session-Local Mappings

A local set of namespace mappings is associated with each session. When a new session is acquired, the mappings present in the persistent namespace registry are copied to the local namespace mappings of that session. A user can then add new mappings or change existing ones. The resulting mapping table applies only within the scope of that session (see §5.11 *Namespace Mapping*).

If a JCR method returns a name from the repository with a namespace URI for which no local mapping exists, a prefix is created automatically and a mapping between that prefix and the namespace URI in question is added to the set of local mappings. The new prefix must differ from those already present among the set of local mappings. If a JCR method is passed a name or path containing a prefix which does not exist in the local mapping an exception is thrown.

3.5.2.1 Effect of Session Namespace Mappings

All methods that take or return names or paths must use the current session (see §3.1.8.2 *Current Session and Workspace*) namespace mappings to dynamically interpret or produce those names or paths according to the current local namespace mapping of the current session.

Though the precise mechanism of this behavior is an implementation detail, its behavior must be equivalent to that of a system where names and paths are stored internally in expanded form and converted dynamically to and from qualified JCR names or paths as necessary.

3.5.3 Namespace Conventions

Names and paths determined by an application provider should be assigned namespace URIs under the control of the provider organization. Because the space of URIs is universally managed, this ensures that naming collisions will not occur between applications from providers that observe this convention.

3.6 Properties

All data stored within a JCR repository is ultimately stored as the values of properties.

3.6.1 Property Types

Every property is of one of the following types: `STRING`, `URI`, `BOOLEAN`, `LONG`, `DOUBLE`, `DECIMAL`, `BINARY`, `DATE`, `NAME`, `PATH`, `WEAKREFERENCE` or `REFERENCE`.

3.6.1.1 STRING

`STRING` properties store instances of `java.lang.String`.

3.6.1.2 URI

`URI` properties store instances of `java.lang.String` that conform to the syntax of a URI-reference as defined in RFC 3986[3].

3.6.1.3 BOOLEAN

`BOOLEAN` properties store instances of the Java primitive type `boolean`.

3.6.1.4 LONG

`LONG` properties store instances of the Java primitive type `long`.

3.6.1.5 DOUBLE

`DOUBLE` properties store instances of the Java primitive type `double`.

3.6.1.6 DECIMAL

`DECIMAL` properties store instances of `java.math.BigDecimal`.

3.6.1.7 BINARY

`BINARY` properties store instances of `javax.jcr.Binary` (see §5.10.5 *Binary Object*).

3.6.1.8 DATE

`DATE` properties store instances of `java.util.Calendar`. Note that an implementation may not support `DATE` values that cannot be represented in the ISO 8601-based notation defined in §3.6.4.3 *From DATE To*. In such cases an attempt to set a property to such a value will throw a `ValueFormatException`.

[3] See `http://www.ietf.org/rfc/rfc3986.txt`.

3.6.1.9 NAME

`NAME` properties store instances of JCR names.

3.6.1.10 PATH

`PATH` properties store instances of JCR paths and serve as pointers to locations within the workspace. `PATH` properties *do not* enforce referential integrity.

3.6.1.11 WEAKREFERENCE

`WEAKREFERENCE` properties serve as pointers to referenceable nodes by storing their identifiers. `WEAKREFERENCE` properties *do not* enforce referential integrity (see §3.8.2 *Referential Integrity*).

3.6.1.12 REFERENCE

`REFERENCE` properties serve as pointers to referenceable nodes by storing their identifiers. `REFERENCE` properties *do* enforce referential integrity (see §3.8.2 *Referential Integrity*).

3.6.2 Undefined Type

The `UNDEFINED` keyword, while not specifying an actual type, may be supported by some repositories as a valid property type attribute value in property definitions within node types. In that context it indicates that the specified property may be of *any* type. No actual existing property in the repository ever has the type `UNDEFINED`.

3.6.3 Single and Multi-Value Properties

A property may be a single-value or a multi-value property.

A single-value property, if it exists, must have a value. There is no such thing as a null value. A multi-value property can have zero or more values. Again there is no such thing as a null value, however a multi-value property can be empty, just as an array can be empty.

The values stored within a multi-valued property are all of the same type and are ordered.

Whether a particular property is a multi-valued property is governed by the property definition applicable to it, which is determined by the node type of the property's parent node (see §3.7 *Node Types*).

Accessing the value of a property is done with `Property.getValue` which returns a single `Value` object. Accessing the set of values of a multi-value property is done through `Property.getValues` which returns a (possibly empty) array of `Value` objects (see §5.10 *Reading Properties*).

3.6.4 Property Type Conversion

When the value of a property is read or written using a type different from that declared for the property, the repository attempts a type conversion according to

the following rules. Note that even in cases where the JCR type conversion is defined in terms of standard JDK type conversion method, failure of conversion must only ever cause a JCR `ValueFormatException` to be thrown and never any exception defined in the JDK API.

3.6.4.1 From STRING To

BINARY: The string is encoded using UTF-8.

DATE: If the string is in the format described in §3.6.4.3 *From DATE To*, it is converted directly, otherwise a `ValueFormatException` is thrown.

DOUBLE: The string is converted using `java.lang.Double.valueOf(String)`.

DECIMAL: The string is converted using the constructor `java.math.BigDecimal(String)`.

LONG: The string is converted using `java.lang.Long.valueOf(String)`.

BOOLEAN: The string is converted using `java.lang.Boolean.valueOf(String)`.

NAME: If the string is a syntactically valid qualified JCR name with a registered prefix, it is converted directly. If it is a syntactically valid expanded JCR name with a registered namespace URI, it is returned in qualified form. If it is a syntactically valid expanded JCR name with an *unregistered* namespace URI, a prefix is created automatically, the mapping added to the local namespace mappings (see §3.5.2 *Session-Local Mappings*), and the name is returned in qualified form. Otherwise a `ValueFormatException` is thrown.

PATH: If the string is a valid JCR path then each name segment is converted as per `NAME` conversion above, and all other segments are converted directly. If one or more name conversions fails or if the string is not a valid path then a `ValueFormatException` is thrown. The presence of an item in the current workspace at that path is not required.

URI: If the string is a syntactically valid URI-reference, it is converted directly, otherwise a `ValueFormatException` is thrown. The string is parsed as described in RFC 3986. In particular, the first colon (":") encountered is interpreted as the scheme delimiter and the string as a whole is assumed to already be in percent-encoded form. This means that if a non-URI-legal character is encountered it is not percent-encoded, but is instead regarded as an error and a `ValueFormatException` is thrown.

REFERENCE or WEAKREFERENCE: If the string is a syntactically valid identifier, according to the implementation, it is converted directly, otherwise a `ValueFormatException` is thrown. The identifier is not required to be that of an existing node in the current workspace.

3.6.4.2 From BINARY To

STRING: An attempt is made to interpret the stream as a UTF-8 encoded string. If the string is not a legal UTF-8 byte sequence then the behavior is implementation-specific.

All Others: The binary stream is first converted to a string, as described above. If this is successful, the resulting string is converted according to the appropriate conversion as described in §3.6.4.1 *From STRING To*.

3.6.4.3 From DATE To

STRING: The date is converted to the following format:

```
sYYYY-MM-DDThh:mm:ss.sssTZD
```

where:

`sYYYY`
Four-digit year with optional leading positive ('`+`') or negative ('`-`') sign. `0000` , `-0000` and `+0000` all indicate the year 1 BCE. `-YYYY` where `YYYY` is the number *y* indicates the year (*y*+1) BCE. The absence of a sign or the presence of a positive sign indicates a year CE. For example, `-0054` would indicate the year 55 BCE, while `+1969` and `1969` indicate the year 1969 CE.

`MM`
Two-digit month (01 = January, etc.)

`DD`
Two-digit day of month (01 through 31)

`hh`
Two digits of hour (00 through 23, or 24 if `mm` is 00 and `ss.sss` is 00.000)

`mm`
Two digits of minute (00 through 59)

`ss.sss`
Seconds, to three decimal places (00.000 through 59.999 or 60.999 in the case of leap seconds)

`TZD`
Time zone designator (either *Z* for Zulu, i.e. UTC, or *+hh:mm* or *-hh:mm*, i.e. an offset from UTC)

Note that the "`T`" separating the date from the time and the separators "`-`"and "`:`" appear literally in the string.

This format is a subset of the format defined by ISO 8601:2004.

If the `DATE` value cannot be represented in this format a `ValueFormatException` is thrown.

BINARY: The date is converted to a string, as described in §3.6.4.2 *From BINARY To*, and this string is encoded in UTF-8.

DOUBLE: The date is converted to the number of milliseconds since 00:00 (UTC) 1 January 1970 (`1970-01-01T00:00:00.000Z`). If this number is out-of-range for a double, a `ValueFormatException` is thrown.

DECIMAL: The date is converted to the number of milliseconds since 00:00 (UTC) 1 January 1970 (`1970-01-01T00:00:00.000Z`).

LONG: The date is converted to the number of milliseconds since 00:00 (UTC) 1 January 1970 (`1970-01-01T00:00:00.000Z`). If this number is out-of-range for a long, a `ValueFormatException` is thrown.

All Others: A `ValueFormatException` is thrown.

Since the string and number formats into which a `DATE` may be converted can hold only a subset of the information potentially contained within a `java.util.Calendar`, conversion from `DATE` to `STRING`, `BINARY`, `DOUBLE`, `DECIMAL` or `LONG` may result in loss of information.

3.6.4.4 From DOUBLE To

STRING: The double is converted using `java.lang.Double.toString()`.

BINARY: The double is converted to a string, as described in §3.6.4.2 *From BINARY To*, and this string is encoded in UTF-8.

DECIMAL: The double is converted using the constructor `java.math.BigDecimal(double)`.

DATE: The double is coerced to a long using standard Java type coercion and interpreted as the number of milliseconds since 00:00 (UTC) 1 January 1970 (`1970-01-01T00:00:00.000Z`). If the resulting value is out of range for a date, a `ValueFormatException` is thrown.

LONG: Standard Java type coercion is used.

All Others: A `ValueFormatException` is thrown.

3.6.4.5 From DECIMAL To

STRING: The decimal is converted using `java.math.BigDecimal.toString()`.

BINARY: The decimal is converted to a string, as described in §3.6.4.2 *From BINARY To*, and this string is encoded in UTF-8.

DOUBLE: The decimal is converted using `java.math.BigDecimal.doubleValue()`.

DATE: The decimal is converted to a long and interpreted as the number of milliseconds since 00:00 (UTC) 1 January 1970 (`1970-01-01T00:00:00.000Z`). If the resulting value is out of range for a date, a `ValueFormatException` is thrown.

LONG: The decimal is converted using `java.math.BigDecimal.longValue()`.

All Others: A `ValueFormatException` is thrown.

3.6.4.6 From LONG To

STRING: The long is converted using `java.lang.Long.toString()`.

BINARY: The long is converted to a string, as described in §3.6.4.2 *From BINARY To*, and this string is encoded in UTF-8.

DECIMAL: The double is converted using the method `java.math.BigDecimal.valueOf(long)`.

DATE: The long is interpreted as the number of milliseconds since 00:00 (UTC) 1 January 1970 (`1970-01-01T00:00:00.000Z`). If the resulting value is out of range for a date, a `ValueFormatException` is thrown.

DOUBLE: Standard Java type coercion is used.

All Others: A `ValueFormatException` is thrown.

3.6.4.7 From BOOLEAN To

STRING: The boolean is converted using `java.lang.Boolean.toString()`.

BINARY: The boolean is converted to a string, as described in §3.6.4.2 *From BINARY To*, and this string is encoded in UTF-8.

All Others: A `ValueFormatException` is thrown.

3.6.4.8 From NAME To

STRING: The name is converted to qualified form according to the current local namespace mapping (see §3.2.5.2 *Qualified Form*).

BINARY: The name is converted to a string, as described in §3.6.4.2 *From BINARY To*, and then encoded using UTF-8.

PATH: The name becomes a relative path of length one.

URI: The name becomes a URI-reference consisting of "`./`" followed by the name in qualified form. For example, the name `foo:bar` becomes the URI-reference `./foo:bar`. The addition of the leading "`./`" is done to ensure that a colon-delimited prefix is not interpreted as a URI scheme name. If the name includes characters that are illegal within a URI-reference (such as any non-ASCII character), the UTF-8 byte representations of these characters are percent-encoded, as described in RFC 3986[4].

All Others: A `ValueFormatException` is thrown.

3.6.4.9 From PATH To

STRING: Each path is converted to standard form according to the current local namespace mapping (see §3.4.3.1 *Standard Form*).

---

[4] See `http://www.ietf.org/rfc/rfc3986.txt`.

BINARY: The path is converted to a string, as described in §3.6.4.2 *From BINARY To*, and then encoded using UTF-8.

NAME: If the path is a relative path of length one it is converted directly, otherwise a `ValueFormatException` is thrown.

URI: If the path is root-based absolute (that is, it has a leading "`/`", see §3.4.4.1.1 *Root-Based Absolute Paths*), it is directly converted into a URI-reference. If the path is identifier-based absolute (§3.4.4.1.2 *Identifier-Based Absolute Paths*) or relative, it becomes a URI-reference consisting of "`./`" followed by the path in standard form. For example, the path `foo:bar/foo:baz` becomes the URI-reference `./foo:bar/foo:baz`. The addition of the leading "`./`" is done to ensure that a colon-delimited prefix is not interpreted as a URI scheme name. If the path includes characters that are illegal within a URI-reference (such as any non-ASCII character), the UTF-8 byte representations of these characters are percent-encoded, as described in RFC 3986[6].

All Others: A `ValueFormatException` is thrown.

3.6.4.10 From URI To

STRING: The `URI` is converted directly into a `STRING`.

BINARY: Since a URI is guaranteed to already contain only ASCII characters it is converted directly to a `BINARY` resulting in series of octets that is a UTF-8 encoding of the character sequence comprising the URI.

NAME: If the `URI` consists of a single path segment without a colon (for example, simply `bar`) it is converted to a `NAME` by decoding any percent-escaped sequences into octet sequences and then decoding those into character sequences using UTF-8. If it has a redundant leading "`./`" followed by a single segment (with or without a colon, like `./bar` or `./foo:bar`) the redundant "`./`" is removed and the remainder is converted to a `NAME` in the same way. Otherwise a `ValueFormatException` is thrown.

PATH: If the `URI` begins with a "`/`" it is converted a `PATH` by decoding any percent-escaped sequences into octet sequences and then decoding those into character sequences using UTF-8. If it consists of a path whose first segment is without a colon (for example, `bar`, `bar/baz` or `bar/foo:baz`) it is converted in the same way. If it consists of a path with a redundant leading "`./`" (for example `./bar/baz`, or `./foo:bar/foo:baz`) the redundant "`./`" is removed and the remainder is converted to a `PATH` as described in §3.6.4.1 *From STRING To*. Otherwise a `ValueFormatException` is thrown.

All Others: A `ValueFormatException` is thrown.

3.6.4.11 From REFERENCE To

STRING: The identifier is converted directly to a string.

BINARY: The identifier is converted directly to a string and then converted to `BINARY` as described in §3.6.4.1 *From STRING To*.

WEAKREFERENCE: The `REFERENCE` is simply turned into a `WEAKREFERENCE`.

All Others: A `ValueFormatException` is thrown.

3.6.4.12 From WEAKREFERENCE To

STRING: The identifier is converted directly to a string.

BINARY: The identifier is converted directly to a string and then converted to `BINARY` as described in §3.6.4.1 *From STRING To*.

REFERENCE: The `WEAKREFERENCE` is simply turned into a `REFERENCE`.

All Others: A `ValueFormatException` is thrown.

3.6.5 Comparison of Values

For any values V_1 and V_2 both of the same property type, the relations *is equal to*, *is ordered before* and *is ordered after* are defined in §3.6.5.1 *CompareTo Semantics*.

Note that the definition of these relations here does not necessarily imply that JCR API methods for testing these relations are supported for every property type. In particular, `Value.equals` is not required to work on `BINARY` values and JCR query is not required to support testing the equality or ordering of `BINARY` values.

3.6.5.1 CompareTo Semantics

For the property types *other than* `BOOLEAN`, `NAME`, `PATH` and `BINARY`, comparison relations are defined in terms of the result of the `compareTo` method on instances `V1` and `V2` of the Java class corresponding to the JCR property type (see each section below for the relevant class). For those types:

- V_1 *is equal to* V_2 if and only if `V1.compareTo(V2) == 0`.
- V_1 *is ordered before* V_2 if and only if `V1.compareTo(V2) < 0`.
- V_1 *is ordered after* V_2 if and only if `V1.compareTo(V2) > 0`.

3.6.5.2 STRING, URI, REFERENCE and WEAKREFERENCE

If V_1 and V_2 are values of type `STRING`, `URI`, `REFERENCE` or `WEAKREFERENCE` then the repository *should* use the semantics of `java.lang.String.compareTo`, as described in §3.6.5.1 *CompareTo Semantics*.

3.6.5.3 DATE

If V_1 and V_2 are values of type `DATE` then the repository *must* use the semantics of `java.lang.Calendar.compareTo`, as described in §3.6.5.1 *CompareTo Semantics*.

3.6.5.4 DOUBLE

If V_1 and V_2 are values of type `DOUBLE` then the repository *must* use the semantics of `java.lang.Double.compareTo`, as described in §3.6.5.1 *CompareTo Semantics*.

3.6.5.5 LONG

If V_1 and V_2 are values of type `LONG` then the repository *must* use the semantics of `java.lang.Long.compareTo`, as described in §3.6.5.1 *CompareTo Semantics*.

3.6.5.6 DECIMAL

If V_1 and V_2 are values of type `DECIMAL` then the repository *must* use the semantics of `java.lang.BigDecimal.compareTo`, as described in §3.6.5.1 *CompareTo Semantics*.

3.6.5.7 BOOLEAN

If V_1 and V_2 are values of type `BOOLEAN` then

- *V_1 is equal to V_2* if and only if `V1 == V2`.
- *V_1 is ordered before V_2* if and only if `V1 == false` and `V2 == true`.
- *V_1 is ordered after V_2* if and only if `V1 == true` and `V2 == false`.

3.6.5.8 NAME

If V_1 and V_2 are values of type `NAME` and $V_1 = (N_1, L_1)$ and $V_2 = (N_2, L_2)$ where N_1 and N_2 are JCR namespaces and L_1 and L_2 are JCR local names then

- *V_1 is equal to V_2* if and only if N_1 is equal to N_2 and L_1 is equal to L_2, according to the semantics of `String.compareTo` (see §3.2.7 *Equality of Names*).
- Ordering is implementation-specific. The only requirement is that a *total order* on values of type `NAME` must be defined, meaning that if V_1 and V_2 are not equal then either V_1 is ordered before V_2 or V_1 is ordered after V_2.

3.6.5.9 PATH

If V_1 and V_2 are values of type `PATH` then

- *V_1 is equal to V_2* if and only if V_1 and V_2 are *segment-equal* (see §3.3.8 *Equality of Paths*).
- Ordering is implementation-specific. The only requirement is that a *total order* on values of type `PATH` must be defined, meaning that if V_1 and V_2 are not equal then either V_1 is ordered before V_2 or V_1 is ordered after V_2.

3.6.5.10 BINARY

If V_1 and V_2 are values of type `BINARY` and given,

- *V_1 is equal to V_2* if and only if V_1 and V_2 are bitwise equivalent.
- Ordering is implementation-specific.

3.6.6 Value.equals Method

An implementation of the `Value` interface must override the inherited method `Object.equals(Object)` so that, given `Value` instances `V1` and `V2`, `V1.equals(V2)` will return `true` if:

- `V1` and `V2` were acquired from the same `Session`, and
- the contents of `V1` and `V2` have not yet been accessed, and
- `V1` and `V2` are of the same type, and
- `V1` *is equal to* `V2` as defined in §3.6.5 *Comparison of Values*.

`V1.equals(V2)` will return `false` otherwise.

In addition:

- The equality comparison must not change the state of either `V1` or `V2` and
- support for `Value.equals` in the case of `BINARY` values is optional.

3.6.7 Length of a Value

The *length* of a value is defined as follows:

- For a `BINARY` value, its length is equal to its length in bytes. This number is returned both by `Binary.getSize` (see §5.10.5 *Binary Object*) and by `Property.getLength` and `Property.getLengths` (see §5.10.3 *Value Length*).
- For other types, the length is the same value that would be returned by calling `java.lang.String.length()` on the `String` resulting from standard JCR property type conversion (see §3.6.4 *Property Type Conversion*). This number is returned by `Property.getLength` and `Property.getLengths`.

For single value properties, the length of a property's value is often referred to as the *property length*.

3.7 Node Types

Node types are used to enforce structural restrictions on the nodes and properties in a workspace by defining for each node, its required and permitted child nodes and properties.

Every node has one declared *primary node type* and zero or more *mixin node types*. Primary node types are typically used to defined the core characteristics of a node, while mixin node types are used to add additional characteristics often related to specific repository functions or to metadata.

In a writable repository a node's primary type is first assigned upon node creation, while mixin types may be assigned on creation or during a node's lifetime. Repository implementations may vary as to how flexible they are in allowing changes to the primary or mixin node types assigned to a node.

Each repository has a single, system-wide registry of node types. Typically, a repository will come with some implementation-determined set of built-in node types. Some of these types may be vendor-specific while others may be standard node types defined by JCR to support common use-cases (see §3.7.11 *Standard Application Node Types*) or repository features. Some repositories may further allow users to register new node types programmatically (see §19 *Node Type Management*).

3.7.1 Node Type Definition Attributes

A node type definition consists of the following attributes:

3.7.1.1 Node Type Name

Every registered node type has a JCR *name*, unique within the repository.

3.7.1.2 Supertypes

A node type has *zero* or more *supertypes*. Supertypes are specified by name.

3.7.1.3 Abstract

A node type may be declared *abstract*, meaning that it cannot be directly assigned to a node, though it may act as a supertype to other node types. The abstract flag is a boolean.

3.7.1.4 Mixin

A node type may be declared a mixin node type. A mixin node type can be assigned to a node during that node's lifetime, not just upon node creation, as is the case with primary node types. The mixin flag is a boolean.

3.7.1.5 Queryable Node Type

A node type may be declared *queryable*, meaning that the node type can be used in a query selector and that the query-related attributes of properties defined in that node type take effect. The *queryable node type* attribute is a boolean.

3.7.1.5.1 Interaction with Property Definitions

If a node type is declared queryable, then the *available query operators*, *full-text searchable* and *query-orderable* attributes of its property definitions take effect (see §3.7.3.3 *Available Query Operators*, §3.7.3.4 *Full-Text Searchable*, §3.7.3.5 *Query-Orderable*). If a node type is declared non-queryable then these attributes of its property definitions have no effect.

3.7.1.6 Orderable Child Nodes

A node type may declare its child nodes orderable, meaning that for all nodes of that type, the order that the child nodes are iterated over can be programmatically controlled by the user (see §23 *Orderable Child Nodes*). The orderable child nodes flag is a boolean.

3.7.1.7 Primary Item

A node type can declare one of its child items as primary, meaning that for all nodes of that type, that child item is accessible through a dedicated API method which does not require the name of the item. (see §5.1.7 *Primary Item Access*). This feature can help generic API clients intelligently traverse an unknown node structure. The primary item may be an item name, which must be a JCR name, or null, meaning that there is no primary item.

3.7.1.7.1 Primary Item and Same-Name Siblings

In cases where the primary child item specifies the name of a set of same-name sibling child nodes, the node with index `[1]` will be regarded as the primary item.

3.7.1.7.2 Property and Child Node With Same Name

In cases where this node has both a child node and a property with the same name and where that name is specified as the primary item name, the child node will be regarded as the primary item (see §22.4 *Property and Node with Same Name*).

3.7.1.8 Property Definitions

A node type may contain a list of *property definitions*, which specify the properties that nodes of that type are permitted or required to have and the characteristics of those properties. The list of property definitions may be empty.

3.7.1.9 Child Node Definitions

A node type may contain a list of child node definitions, which specify the permitted or required child nodes and their characteristics. The list of child node definitions may be empty.

3.7.2 Item Definition Attributes

Property and child node definitions have some attributes in common, while others are specific to either property definitions or child nodes in particular (this is reflected in the API interfaces, see §8.3 *ItemDefinition Object*). The common attributes are:

3.7.2.1 Item Definition Name

The name attribute specifies the set of child nodes or properties to which the definition applies. This set is called the *scope* of the definition. An item within the scope of a given definition is called a *scoped item* (scoped property, scoped child node) of that definition. The definition within whose scope a given item falls is called the *scoping definition* of that item.

In the standard case the scope consists of the single item named by the attribute and must be a JCR name.

3.7.2.1.1 Item Definition Name and Same-Name-Siblings

In a repository that supports *same-name siblings* (see §22 *Same-Name Siblings*), the name attribute of a child node definition will have scope over all the child nodes of that name. In this case the attribute must also be a JCR name.

3.7.2.1.2 Item Definition Name and Residual Definitions

In a repository that supports *residual definitions* the name attribute may be "`*`" (asterisk), specifying that the definition is residual, meaning that its scope consists of all other properties (child nodes), which are not otherwise scoped by any of the other property (child node) definitions in the effective node type of the node (see §3.7.6.5 *Effective Node Type*).

3.7.2.1.3 Multiple Item Definitions with the Same Name

A node type may have two or more item definitions with identical *name* attributes. On `Node.setProperty` or `Node.addNode`, the repository must choose among the available definitions for one which matches the name and possible type information specified in the method call. If this information is insufficient to select a single definition unambiguously, the repository may choose a definition based on some implementation-specific criteria or fail the operation (see §10.4 *Adding Nodes and Setting Properties*).

3.7.2.2 Protected

If an item *I* is declared protected it is *repository-controlled*.

If *I* is a node then, through the *core write methods of JCR* (see §10.2 *Core Write Methods*),

- *I* cannot be removed,
- child nodes of *I* cannot be added, removed, or reordered,
- properties of *I* cannot be added or removed,
- the values of existing properties of *I* cannot be changed,
- the primary node type of *I* cannot be changed and
- mixin node types cannot be added to or removed from *I*.

If I is a property then, through the *core write methods of JCR* (see §10.2 *Core Write Methods*),

- *I* cannot be removed and
- the value of *I* cannot be changed.

Additionally, if *I* is a property, its being repository-controlled also implies that its value is under the control of the repository and can change at any time, before or after save. See §3.7.2.3.2 *Auto-Created and Protected*.

3.7.2.3 Auto-Created

An item may be declared *auto-created*, meaning that it is automatically created upon creation of its parent node. The auto-created attribute is a boolean.

3.7.2.3.1 Auto-Created and Non-Protected

If an item is auto-created but not protected then it *must* be *immediately* created in transient space when its parent node is created. Creation of auto-created non-protected items must never be delayed until save (see §10.11 *Saving*).

3.7.2.3.2 Auto-Created and Protected

If an item is both auto-created and protected, then it *should* be immediately created in transient space when its parent node is created. Creation of auto-created protected items should not be delayed until save, though doing so does not violate JCR compliance. In some implementations the value of an auto-created property may be assigned upon save, in such cases the creation of the property may also be delayed until save (see, for example, §3.7.1 *Identifier Assignment*).

3.7.2.3.3 Auto-created and Same-Name Siblings

In a repository that supports same-name siblings (see §22 *Same-Name Siblings*), a child node definition may specify that a node be both auto-created and allow same-name siblings. In that case the repository must create at least one such child node with the specified name upon parent node creation, though it may create more than one.

3.7.2.3.4 Auto-created and Residual Definitions

In repositories that support residual definitions, an item cannot be both auto-created and residual (see §3.7.2.1.2 *Item Definition Name and Residual Definitions*).

3.7.2.3.5 Chained Auto-creation

An auto-created node may itself have auto-created child items, resulting in the automatic creation of a tree of items. However, chaining that produces an infinite loop of item creation is not permitted. A repository must ensure that at no time does it have a set of registered node types that could result in such behavior (see §19 *Node Type Management*).

3.7.2.4 Mandatory

An item may be declared *mandatory*, meaning that the item must exist before its parent node is saved.

3.7.2.4.1 Mandatory and Multi-Value Properties

Since single-value properties either have a value or do not exist (there being no concept of the null value, see §10.4.2.4 *No Null Values*), a mandatory single-

value property must have a value. A mandatory multi-value property, on the other hand, may have zero or more values.

3.7.2.4.2 Mandatory and Same-Name Siblings

In a repository that supports same-name siblings, a child node definition may specify that a node be both mandatory and allow same-name siblings. In that case at least one child node must exist upon save of the parent node (see §22 *Same-Name Siblings*).

3.7.2.4.3 Mandatory and Residual Definitions

In repositories that support residual definitions, an item cannot be both mandatory and residual (see §3.7.2.1.2 *Item Definition Name and Residual Definitions*).

3.7.2.5 On-Parent-Version

In a repository that supports *simple* or *full versioning* the on-parent-version attribute governs the behavior of the child item when its parent node is checked-in (see §15.2 *Check-In: Creating a Version*). In repositories that do not support *simple* or *full versioning* this attribute has no effect.

3.7.3 Property Definition Attributes

A property definition has all the attributes of a generic item definition as well as the following property-specific attributes:

3.7.3.1 Property Type

A property definition must specify a property type. This must be one of the JCR property types (see §3.6.1 *Property Types*) or, in repositories that support it, the `UNDEFINED` keyword, indicating that the property scoped by this definition can be of any type (see §3.6.2 *Undefined Type*). An attempt to save a property with a type different from that required by its definition will fail if conversion to that type is not possible (see §10.4.2 *Setting a Property* and §3.6.4 *Property Type Conversion*).

3.7.3.2 Default Values

The *default values* attribute of a property definition defines the values assigned to property if it is *auto-created*. If the property is single-valued this attribute will hold a single value. If it is multi-valued this attribute will hold an array of values. A default values setting of `null` indicates that the property does not have a single static default value. It may have no default value at all or it may have a parameterized default value defined externally to this specification. If the scoped property is not *auto-created* then this attribute has no effect.

3.7.3.3 Available Query Operators

A property definition declares the set of query comparison operators that can be validly applied to a property. The set of operators that can appear in this attribute may be limited by implementation-specific constraints that differ across property

types. For example, some implementations may permit property definitions to provide `EqualTo` and `NotEqualTo` (see §6.7.16 *Comparison*) as available operators for `BINARY` properties while others may not. However, in all cases where a JCR-defined operator *is* potentially available for a given property type, its behavior must conform to the comparison semantics defined in §3.6.5 *Comparison of Values*.

3.7.3.3.1 Interaction with Node Type Definition

This attribute only takes effect if the node type holding the property definition has a queryable setting of *true* (see §3.7.1.5 *Queryable Node Type*).

3.7.3.4 Full-Text Searchable

A property may be declared *full-text searchable*, meaning that its value is accessible through the full-text search function within a query (see §6.7.19 *FullTextSearch*). The full-text searchable flag is a boolean.

3.7.3.4.1 Interaction with Node Type Definition

This attribute only takes effect if the node type holding the property definition has a queryable setting of *true* (see §3.7.1.5 *Queryable Node Type*), otherwise this attribute is automatically set to *false*.

3.7.3.5 Query-Orderable

A property may be declared *query-orderable*, meaning that query results may be ordered by this property using the *order* clause of a query (see §6.7.37 *Ordering*). The query-orderable flag is a boolean.

3.7.3.5.1 Interaction with Node Type Definition

This attribute only takes effect if the node type holding the property definition has a queryable setting of *true* (see §3.7.1.5 *Queryable Node Type*), otherwise this attribute is automatically set to *false*.

3.7.3.6 Value Constraints

A property definition may impose constraints on the value that the property may hold. These value constraints are defined by an array of strings, whose format differs depending on the type of the property.

Each string in the returned array specifies a constraint on the values of the property. In order to be valid, *each* value of the property (since a property may be multi-valued) must independently meet *at least one* of the constraints.

If a property does not exist or, in the case of multi-value properties, contains an empty array, the constraint set is considered to have been met.

An attempt to save a property whose value or values fail to meet the constraint criteria will fail (see §10.11 *Saving*).

Reporting constraint information is optional on a per property instance level. The return of an empty array indicates that there are no *expressible* constraints,

meaning that either there are constraints but they are not expressible in the constraint-string syntax, or constraint discovery is not supported for that property.

Constraint strings have different formats depending on the type of the property in question. The following sections describe the value constraint syntax for each property type.

3.7.3.6.1 STRING and URI Constraints

For `STRING` and `URI` properties, the constraint string is a regular expression pattern according to the syntax of `java.util.regex.Pattern`.

3.7.3.6.2 PATH Constraints

For `PATH` properties, the constraint is an absolute or relative path, possibly terminating with a "`*`" as the last segment.

On assignment the constraint may be passed in any valid lexical form, with the possible addition of a trailing "`*`". The constraint, however is stored as a JCR path in normalized form plus an optional *match-descendants* indicator corresponding to the "`*`". The constraint is returned in normalized standard form (see §3.4.5.1 *Standard Form* and §3.4.5 *Normalized Paths*).

For a constraint *without* match-descendants, the constraint is met when the property value is equal to the constraint. For a constraint *with* match-descendants, the constraint is met when the property value is either equal to the constraint or equal to a descendant path of the constraint (see §3.4.8 *Equality of Paths*).

3.7.3.6.3 NAME Constraints

For `NAME` properties, the constraint is a JCR name. On assignment the constraint may be passed in any valid lexical form but is returned in qualified form (see §3.2.5.2 *Qualified Form*).The constraint is met if the property value is equal to the constraint (see §3.2.7 *Equality of Names*).

3.7.3.6.4 REFERENCE and WEAKREFERENCE Constraints

For `REFERENCE` and `WEAKREFERENCE` properties, the constraint is a JCR name. The constraint is met if the target node of the property is of the node type indicated by the constraint (see §3.7.6.3 *Is-of-Type Relation*). On assignment the constraint passed may be in any valid lexical form but is returned in qualified form (see §3.2.5.2 *Qualified Form*).

3.7.3.6.5 BINARY, DATE, LONG, DOUBLE and DECIMAL Constraints

The remaining types all have value constraints in the form of inclusive or exclusive ranges specified according to the following pattern:

```
Constraint ::= Open Min ',' Max Close

Open ::= '[' | '('
```

```
Close ::= ']' | ')'

Min ::= /* Type dependent, see below */

Max ::= /* Type dependent, see below */

/* See §1.3.1 String Literals in Syntactic Grammars for details
   on the interpretation of string literals in this grammar */
```

The brackets "`[`" and "`]`" indicate inclusivity, while "`(`" and "`)`" indicate exclusivity. A missing `min` or `max` value indicates no bound in that direction. The meaning of the *`min`* and *`max`* values themselves differ between types as follows:

`BINARY`: *`min`* and *`max`* specify the allowed size range of the binary value in bytes.

`DATE`: *`min`* and `max` are dates specifying the allowed date range. The date strings must be in the standard string serialization (see §3.6.4.3 *From DATE To)*.

`LONG, DOUBLE, DECIMAL` : *`min`*, *`max`* are valid Java language numeric literals.

The range is evaluated according to the standard value comparison rules (see §3.6.5 *Comparison of Values*).

To specify a constant value, the constant itself, "*`c`*" may be used instead of the bracket notation, though the constraint is always returned in bracket notation.

3.7.3.6.6 BOOLEAN

For `BOOLEAN` properties the constraint string can be either "`true`" or "`false`". In most cases `getValueConstraints` will return an empty array since placing a constraint on a `BOOLEAN` value is uncommon.

3.7.3.6.7 Choice Lists

Because constraints are returned as an array of disjunctive constraints, in many cases the elements of the array can serve directly as a *choice list*. This may, for example, be used by an application to display options to the end user indicating the set of permitted values.

3.7.3.7 Multi-Value

A property can be declared *multi-valued*. An attempt to set a single-value property by passing an array will fail. Similarly, an attempt to set a multi-value property by passing a non-array will also fail (see §10.4.2 *Setting a Property*).

3.7.4 Child Node Definition Attributes

A child node definition has all the attributes of a generic item definition as well as the following node-specific attributes:

3.7.4.1 Required Primary Node Types

A child node definition must declare one or more *required primary node types*.

In order to successfully save a scoped child node *N*, it must be true for each required primary type *R* that the assigned primary type *A* of *N* is of type *R* (see §3.7.6.3 *Is-of-Type Relation*).

In cases where this attribute specifies more than one required node type, any particular node instance will still have only one assigned primary type, but that type must be a subtype of *all* of the types specified by this attribute. Such a situation may arise, for example, in repositories that support multiple inheritance of node types.

3.7.4.2 Default Primary Node Type

The *default primary type* of a child node definition is a JCR name defining the node type that the child node will be given if it is auto-created or created without an explicitly specified node type. This node type must be the same as or a subclass of each of the required primary node types.

If `null` is returned this indicates that no default primary type is specified and that therefore an attempt to create this node without specifying a node type will fail.

3.7.4.3 Same-Name Siblings

The *same-name sibling* attribute of a child node definition indicates whether the child node can have sibling nodes with the same name (see §22 *Same-Name Siblings*). In repositories that do not support same-name siblings this attribute has no effect.

3.7.5 Mixin Node Types

Mixin node types are used to add additional properties or child nodes to a given node instance, typically in order to expose some aspect of a specialized repository feature. For example, referenceability is supported by the mixin `mix:referenceable` which defines the property `jcr:uuid` to expose a node's identifier (see §3.3 *Identifiers*).

3.7.5.1 Mixins Apply Per Node Instance

Mixin node types apply to specific node instances within a workspace, allowing the repository to decouple support for some repository features from the primary node type assigned to that node. In effect, mixin node types permit *per instance node type inheritance*. In a writable repository mixin node types can be assigned to a node during its lifetime, not just upon creation.

3.7.5.2 Mixins and Inheritance

A mixin node type may have one or more supertypes, which must also be mixin types. Additionally, a mixin node type can serve as a supertype of a primary type. This is typically done to build a mixin-linked feature into a primary node type. For example, if a repository requires all nodes of type `xyz:Document` to be referenceable it can specify that `mix:referenceable` as a supertype of `xyz:Document`.

3.7.5.3 Mixins Are Not Stand-Alone

A mixin node type cannot be used by itself as the node type of a node. A primary node type is always required.

3.7.6 Node Type Inheritance

A mixin node type *may* be part of an inheritance hierarchy. A primary node (other than `nt:base`) must at least be a subtype of the common base primary type, `nt:base` (see §3.7.10 *Base Primary Node Type*). The semantics of inheritance are defined by the following rules.

3.7.6.1 Supertype Relation

The *supertype* relation is transitive: If T_1 is a supertype of T_2 and T_2 is a supertype of T_3 then T_1 is a supertype of T_3.

The *supertype* relation always and only stems from explicit *supertypes* attribute declarations within the set of node types: For T_1 to be a supertype of T_2 it is *not sufficient* that the item definitions of T_2 be a superset of the item definitions of T_1. For that to be the case, T_2 must *declare* T_1 as a supertype.

3.7.6.2 Subtype Relation

The *subtype* relation is the converse of supertype: T_1 is a subtype of T_2 if and only if T_2 is a supertype of T_1. Hence, subtype is also a transitive relation.

3.7.6.3 Is-of-Type Relation

The *is-of-type* relation which holds between node instances and node types (as in, node *N* is of type *T*) is transitive across the *subtype* relation: If *N* is of type T_2 and T_2 is a subtype of T_1 then *N* is (also) of type T_1. This predicate appears in the API as the method `Node.isNodeType()` (see §8.6 *Node Type Information for Existing Nodes*). This relation is also the one that is relevant in the child node definition attribute *required primary node types* (see §3.7.4.1 *Required Primary Node Types*).

The *is of type* relation always and only stems from an explicit assignment of a node type to a node: For node *N* to be of type *T* it is *not sufficient* for *N* to have the child items declared by *T*. For that to be the case, *N* must be *explicitly assigned* the type *T*, or a subtype of *T*.

3.7.6.4 Abstract Node Types

As mentioned (see §3.7.1.3 *Abstract*), a node type may be declared abstract, meaning that it cannot be assigned as the primary or mixin node type of a node but can be used in the definition of other node types as a supertype.

3.7.6.5 Effective Node Type

The complete set of node type constraints on a particular node is referred to as that node's *effective node type*. This consists of the sum of all attributes

- declared in that node's primary type,

- inherited by that node's primary type,
- declared in that node's mixin node types, and
- inherited by that node's mixin node types.

The summing of these attributes must conform to the semantics of subtyping defined in this section.

3.7.6.6 Semantics of Subtyping

The general principle guiding inheritance is to preserve the *is-a* relation across subtyping. This implies that if *T'* is a subtype of *T* and *N* is a valid instance of *T'* then:

- *N* must be a valid instance of *T*.
- A method call that depends on the truth of the test *N is of type T* must not fail *solely* due to *N* being of type *T'*.

3.7.6.7 Node Type Attribute Subtyping Rules

If *T'* is a subtype of *T* then the following must hold:

The name of *T'* must differ from the name of *T*.

- The supertypes list of *T'* must include either *T* or a subtype of *T*.
- If *T* is a primary type, *T'* must be a primary type. However, if *T* is a mixin then *T'* may be either a mixin or a primary type.

If *T* has orderable child nodes then *T'* must have orderable child nodes.

If *T* specifies a primary item *I* then *T'* inherits that setting and must not specify a primary item other than *I*.

T' may declare any number of property definitions as long as they are not invalid (see §3.7.6.8 *Item Definitions in Subtypes*).

T' may declare any number of child node definitions as long as they are not invalid (see §3.7.6.8 *Item Definitions in Subtypes*).

3.7.6.8 Item Definitions in Subtypes

If *T* is a registered node type and *T'* is the definition of a subtype of *T* that meets the criteria in the preceding sections, then an item definition *D'* in *T'* is either *additive*, *overriding* or *invalid*, as determined by the following algorithm:

- If *D'* is not statically valid then *D'* is invalid.
- If *D'* is a residual definition then *D'* is additive.
- If there *does not* exists a definition *D* in *T* with a name and class (i.e., either *node* or *property*) identical to that of *D'* then *D'* is additive.
- If there *does* exist a definition *D* in *T* with name and class identical to that of *D'* then *D'* is overriding if:

    - The implementation supports item definition overrides in this instance (implementations are free allow or disallow overrides globally or on an instance-by-instance basis)
    - If *D* is a property definition then *D* and *D'* have identical multiple settings and any property values valid against *D'* would also be valid against *D*.
    - If *D* is a child node definition then *D* and *D'* have identical same-name sibling settings.
    - If *D* is autocreated, mandatory or protected then *D'* must be, respectively, that as well.
- Otherwise, *D'* is invalid.

If *D'* is *additive* then when *T'* is registered *D'* becomes part of *T'* alongside all item definitions inherited from *T*.

If *D' overrides D* then when *T'* is registered *D'* replaces the definition *D* that would otherwise have been inherited from *T*.

If *D'* is *invalid* then *T'* cannot be registered.

3.7.6.9 Effect of Inheritance Rules

The rules of inheritance will have most impact on repositories that allow

- a wide latitude in assigning mixins to nodes,
- registration of custom node types (see §19 *Node Type Management*) or, in particular,
- registration of custom node types with multiple super types (multiple inheritance).

In fixed node type repositories (those without support for mixin assignment or node type registration), adherence to the inheritance rules is simply a matter of ensuring that the correct relations hold among the statically defined node type that the system exposes.

3.7.7 Applicable Item Definition

Though there may be more than one definition in the parent node's type that *could* apply to the child item, the definition that does apply is determined by the implementation and remains constant through the lifetime of the item.

In writable repositories the applicable item definition is determined at item creation time.

3.7.8 Root Node Type

The node type of the root node of each workspace is implementation-determined. There are no restrictions other than those implied by the feature set of the repository. For example, a repository that exposes system data under

/jcr:system will necessarily have a root node of a type that allows a jcr:system child node.

3.7.9 Node Type Notation

The node type definitions shown in this specification use the *compact node type definition* (CND) notation (see §25.2 *Compact Node Type Definition Notation*).

3.7.9.1 Implementation Variants in Node Types

Some of the attributes of the node types defined in this specification may vary across implementations. For example, it is implementation-dependent which node types and which properties are queryable (see §3.7.1.5 *Queryable Node* and §3.7.3.3 *Available Query Operators*). Similarly, some of the standard application node types (see §3.7.11 *Standard Application Node Types*) may vary as to the *on-parent-version* and *protected* status of some properties. In the CND notation, variant attributes are indicated with either a question mark (for example, protected? and opv?) or, in the case of the queryable node type attribute, by the absence of an explicit indicator. For the queryable attribute of a node type to be non-variant it must be explicitly defined using the keywords query or noquery, (see §25.2 *Compact Node Type Definition Notation*).

3.7.10 Base Primary Node Type

All repositories must supply the *base primary node type*, nt:base, as a built-in type.

3.7.10.1 nt:base

```
[nt:base] abstract
  - jcr:primaryType (NAME) mandatory autocreated
      protected COMPUTE
  - jcr:mixinTypes (NAME) protected multiple COMPUTE
```

nt:base is an abstract primary node type that is the base type for all other primary node types. It is the only primary node type without supertypes.

nt:base exposes type information about a node through the properties jcr:primaryType, and jcr:mixinTypes.

Since every other primary type must be a subtype of nt:base (see §3.7.6.2 *Subtype Relation*), every primary node type will inherit these two type-reflective property definitions.

jcr:primaryType is a protected mandatory NAME property which holds the name of the declared primary node type of its node. The repository must maintain its value accurately throughout the lifetime of the node (see §10.10 *Node Type Assignment*). Since it is mandatory, every node will have this property.

jcr:mixinTypes is a non-mandatory protected multi-value NAME property which holds a list of the declared mixin node types of its node. It is non-mandatory but is required to be present on any node that has one or more declared mixin types. If it is present, the repository must maintain its value accurately throughout the lifetime of the node (see §10.10.3 *Assigning Mixin Node Types*).

3.7.11 Standard Application Node Types

JCR defines a number of standard application node types designed to support common application-level entities. A repository may supply zero or more of these as built-in types (see §24 *Repository Compliance*).

3.7.11.1 nt:hierarchyNode

```
[nt:hierarchyNode] > mix:created abstract
```

This abstract node type serves as the supertype of `nt:file` and `nt:folder` and inherits the item definitions of `mix:created` and so requires the presence of that node type (see §3.7.11.7 *mix:created*).

3.7.11.2 nt:file

```
[nt:file] > nt:hierarchyNode primaryitem jcr:content
  + jcr:content (nt:base) mandatory
```

Nodes of this node type may be used to represent files. This node type inherits the item definitions of `nt:hierarchyNode` and requires a single child node called `jcr:content`. The `jcr:content` node is used to hold the actual content of the file. This child node is mandatory, but not auto-created. Its node type will be application-dependent and therefore it must be added by the user. A common approach is to make the `jcr:content` a node of type `nt:resource`. The `jcr:content` child node is also designated as the primary child item of its parent.

3.7.11.3 nt:linkedFile

```
[nt:linkedFile] > nt:hierarchyNode primaryitem jcr:content
  - jcr:content (REFERENCE) mandatory
```

The `nt:linkedFile` node type is similar to `nt:file`, except that the content node is not stored directly as a child node, but rather is specified by a `REFERENCE` property. This allows the content node to reside anywhere in the workspace and to be referenced by multiple `nt:linkedFile` nodes. The content node must be referenceable. Support for this node type requires support for *referenceable nodes* with *referential integrity* (see §3.8.2 *Referential Integrity*).

3.7.11.4 nt:folder

```
[nt:folder] > nt:hierarchyNode
  + * (nt:hierarchyNode) VERSION
```

Nodes of this type may be used to represent folders or directories. This node type inherits the item definitions of `nt:hierarchyNode` and adds the ability to have any number of other `nt:hierarchyNode` child nodes with any names. This means, in particular, that it can have child nodes of types `nt:folder`, `nt:file` or `nt:linkedFile`.

3.7.11.5 nt:resource

```
[nt:resource] > mix:mimeType, mix:lastModified
  primaryitem jcr:data
  - jcr:data (BINARY) mandatory
```

This node type may be used to represent the content of a file. In particular, the `jcr:content` subnode of an `nt:file` node will often be an `nt:resource`. Note that the definition of this node type indicates multiple inheritance (see §3.7.6 *Node Type Inheritance*).

3.7.11.6 mix:title

```
[mix:title] mixin
  - jcr:title (STRING) protected? OPV?
  - jcr:description (STRING) protected? OPV?
```

This mixin node type can be used to add standardized title and description properties to a node.

3.7.11.7 mix:created

```
[mix:created] mixin
  - jcr:created (DATE) autocreated protected? OPV?
  - jcr:createdBy (STRING) autocreated protected? OPV?
```

This mixin node type can be used to add standardized creation information properties to a node. In implementations that make these properties protected, their values are controlled by the repository, which *should* set them appropriately upon the initial persist of a node with this mixin type. In cases where this mixin is added to an already existing node the semantics of these properties are implementation specific (see §10.10.3 *Assigning Mixin Node Types*).

3.7.11.8 mix:lastModified

```
[mix:lastModified] mixin
  - jcr:lastModified (DATE) autocreated protected? OPV?
  - jcr:lastModifiedBy (STRING) autocreated protected? OPV?
```

This mixin node type can be used to provide standardized modification information properties to a node. In implementations that make these properties protected, their values are controlled by the repository, which *should* set them appropriately upon a *significant modification* of the subgraph of a node with this mixin. What constitutes a significant modification will depend on the semantics of the various parts of a node's subgraph and is implementation-dependent.

3.7.11.9 mix:language

```
[mix:language] mixin
  - jcr:language (STRING) protected? OPV?
```

This mixin node type can be used to provide a standardized property that specifies the natural language in which the content of a node is expressed. The value of the `jcr:language` property should be a language code as defined in RFC

4646[5]. Examples include "en" (English), "en-US" (United States English), "de" (German) and "de-CH" (Swiss German).

3.7.11.10 mix:mimeType

```
[mix:mimeType] mixin
  - jcr:mimeType (STRING) protected? OPV?
  - jcr:encoding (STRING) protected? OPV?
```

This mixin node type can be used to provide standardized mimetype and encoding properties to a node.

If a node of this type has a primary item that is a single-value BINARY property then jcr:mimeType property indicates the media type[6] applicable to the contents of that property and, if that media type is one to which a text encoding applies, the jcr:encoding property indicates the character set[7] used.

If a node of this type does not meet the above precondition then the interpretation of the jcr:mimeType and jcr:encoding properties is implementation-dependent.

3.7.11.11 nt:address

```
[nt:address]
  - jcr:protocol (STRING)
  - jcr:host (STRING)
  - jcr:port (STRING)
  - jcr:repository (STRING)
  - jcr:workspace (STRING)
  - jcr:path (PATH)
  - jcr:id (WEAKREFERENCE)
```

This node type may be used to represent the location of a JCR item not just within a particular workspace but within the space of all workspaces in all JCR repositories.

The jcr:protocol property stores a string holding the protocol through which the target repository is to be accessed.

The jcr:host property stores a string holding the host name of the system through which the repository is to be accessed.

The jcr:port property stores a string holding the port number through which the target repository is to be accessed.

---

[5] see http://www.ietf.org/rfc/rfc4646.txt.

[6] See http://www.iana.org/assignments/media-types.

[7] See http://www.iana.org/assignments/character-sets.

The semantics of these properties are left undefined but are assumed to be known by the application. The names and descriptions of the properties are not normative and the repository does not enforce any particular semantic interpretation on them.

The `jcr:repository` property stores a string holding the name of the target repository.

The `jcr:workspace` property stores the name of a workspace.

The `jcr:path` property stores a path to an item.

The `jcr:id` property stores a weak reference to a node.

In most cases either the `jcr:path` or the `jcr:id` property would be used, but not both, since they may point to different nodes. If any of the properties other than `jcr:path` and `jcr:id` are missing, the address can be interpreted as *relative* to the current container at the same level as the missing specifier. For example, if no repository is specified, then the address can be interpreted as referring to a workspace and path or id within the current repository.

3.7.12 Entity Tags

It is often useful for an application to be able to quickly find whether the value of a `BINARY` property has changed since the last time it was checked. This is particularly useful when determining whether to invalidate a cache containing a copy of the `BINARY` value.

The `mix:etag` mixin type defines a standardized identity validator for `BINARY` properties similar to the entity tags used in HTTP/1.1[8].

3.7.12.1 mix:etag

```
[mix:etag] mixin
  - jcr:etag (STRING) protected autocreated
```

A `jcr:etag` property is an opaque string whose syntax is identical to that defined for entity tags in HTTP/1.1. Semantically, the `jcr:etag` is comparable to the HTTP/1.1 strong entity tag.

On creation of a `mix:etag` node `N`, or assignment of `mix:etag` to `N`, the repository must create a `jcr:etag` property with an implementation determined value.

The value of the `jcr:etag` property must change immediately on persist of any of the following changes to `N`:

- A `BINARY` property is added to `N`.

[8] See `http://www.ietf.org/rfc/rfc2616.txt` §3.11.

- A `BINARY` property is removed from `N`.
- The value of an existing `BINARY` property of `N` changes.

3.7.13 Unstructured Content

Support for unstructured content may be provided by supporting a free-form node type: `nt:unstructured`. Support for this node type requires support for the `UNDEFINED` property type value.

3.7.13.1 nt:unstructured

```
[nt:unstructured]
  orderable
  - * (UNDEFINED) multiple
  - * (UNDEFINED)
  + * (nt:base) = nt:unstructured sns VERSION
```

This node type is used to store unstructured content. It allows any number of child nodes or properties with any names. It also allows multiple nodes having the same name as well as both multi-value and single-value properties with any names. This node type also supports client-orderable child nodes.

3.7.14 Node Type Definition Storage

A repository may expose the definitions of its available node types in content using the node types `nt:nodeType`, `nt:propertyDefinition` and `nt:childNodeDefinition`. If a repository exposes node type definitions in content, then that repository must also support the system node (see §3.11 *System Node*) and the node type definitions should be located below `/jcr:system/jcr:nodeTypes`. Support for these node types also requires support for same-name siblings (see §22 *Same-Name Siblings*).

3.7.14.1 nt:nodeType

```
[nt:nodeType]
  - jcr:nodeTypeName (NAME) protected mandatory
  - jcr:supertypes (NAME) protected multiple
  - jcr:isAbstract (BOOLEAN) protected mandatory
  - jcr:isQueryable (BOOLEAN) protected mandatory
  - jcr:isMixin (BOOLEAN) protected mandatory
  - jcr:hasOrderableChildNodes (BOOLEAN) protected mandatory
  - jcr:primaryItemName (NAME) protected
  + jcr:propertyDefinition (nt:propertyDefinition)
      = nt:propertyDefinition protected sns
  + jcr:childNodeDefinition (nt:childNodeDefinition)
      = nt:childNodeDefinition protected sns
```

This node type is used to store a node type definition. Property and child node definitions within the node type definition are stored as same-name sibling nodes of type `nt:propertyDefinition` and `nt:childNodeDefinition`.

3.7.14.2 nt:propertyDefinition

```
[nt:propertyDefinition]
  - jcr:name (NAME) protected
  - jcr:autoCreated (BOOLEAN) protected mandatory
```

```
  - jcr:mandatory (BOOLEAN) protected mandatory
  - jcr:onParentVersion (STRING) protected mandatory
      < 'COPY', 'VERSION', 'INITIALIZE', 'COMPUTE',
        'IGNORE', 'ABORT'
  - jcr:protected (BOOLEAN) protected mandatory
  - jcr:requiredType (STRING) protected mandatory
    < 'STRING', 'URI', 'BINARY', 'LONG', 'DOUBLE',
      'DECIMAL', 'BOOLEAN', 'DATE', 'NAME', 'PATH',
      'REFERENCE', 'WEAKREFERENCE', 'UNDEFINED'
  - jcr:valueConstraints (STRING) protected multiple
  - jcr:defaultValues (UNDEFINED) protected multiple
  - jcr:multiple (BOOLEAN) protected mandatory
  - jcr:availableQueryOperators (NAME) protected mandatory
      multiple
  - jcr:isFullTextSearchable (BOOLEAN) protected mandatory
  - jcr:isQueryOrderable (BOOLEAN) protected mandatory
```

This node type used to store a property definition within a node type definition, which itself is stored as an `nt:nodeType` node.

3.7.14.3 nt:childNodeDefinition

```
[nt:childNodeDefinition]
  - jcr:name (NAME) protected
  - jcr:autoCreated (BOOLEAN) protected mandatory
  - jcr:mandatory (BOOLEAN) protected mandatory
  - jcr:onParentVersion (STRING) protected mandatory
      < 'COPY', 'VERSION', 'INITIALIZE', 'COMPUTE',
        'IGNORE', 'ABORT'
  - jcr:protected (BOOLEAN) protected mandatory
  - jcr:requiredPrimaryTypes (NAME) = 'nt:base' protected
      mandatory multiple
  - jcr:defaultPrimaryType (NAME) protected
  - jcr:sameNameSiblings (BOOLEAN) protected mandatory
```

This node type used to store a child node definition within a node type definition, which itself is stored as an `nt:nodeType` node.

3.7.14.4 Representing Null Attributes

The attributes that make up a node type definition may in some cases have no set value (for example, some child node definitions may not define a *default primary type*). To store this information (i.e., the lack of a value) in an `nt:nodeType`, `nt:childNodeDefinition` or `nt:propertyDefinition` node the property representing that attribute must simply be not present, since null values for single-value properties are not permitted (see §10.4.2.4 *No Null Values*).

3.7.14.5 Representing Residual Items

To indicate that a property or child node definition is residual, the value returned by `ItemDefinition.getName()` is "`*`". However, "`*`" is not a valid value for the property `jcr:name` in an `nt:propertyDefinition` or `nt:childNodeDefinition` node (because `jcr:name` it is a `NAME` property, not a `STRING`). As a result, an in-content definition of a residual item will simply not have a `jcr:name` property.

3.7.15 Repository Feature Node Types

JCR defines a number of node types in order to support specific repository features. Descriptions of these node types are found in their corresponding feature sections. The following list summarizes the node types and their associated features:

Referenceable Nodes: `mix:referenceable` (see §3.8 *Referenceable Nodes*).

Locking: `mix:lockable` (see §17 *Locking*).

Shareable Nodes: `mix:shareable` (see §3.9 *Shareable Nodes Model*).

Lifecycles: `mix:lifecycle` (see §18 *Lifecycle Management*).

Versioning: `mix:simpleVersionable`, `mix:versionable`, `nt:version`, `nt:versionHistory`, `nt:frozenNode`, `nt:versionLabels`, `nt:versionedChild` (see §3.13 *Versioning Model*).

3.7.16 JCR Node Type Variants

An implementation *may* provide a variant of a JCR node type as a built-in under certain conditions.

3.7.16.1.1 Replacing the Canonical Type

Such a variant must have the same name as the canonically defined type and thus replace it in that implementation's set of available node types.

3.7.16.1.2 Additions to the Hierarchy

An implementation may alter the definition of a JCR node type by adding supertypes. These additional supertypes may be either JCR mixin node types or implementation-specific mixin or primary node types. For example, a repository may require that all nodes of type `nt:file` be, additionally, `mix:versionable`. In such a repository the definition of `nt:file`, when introspected, would report an additional supertype of `mix:versionable`.

This extension mechanism is distinct from the automatic addition of mixin types that may be done on node creation (see §10.10.3.3 *Automatic Addition and Removal of Mixins*). Though the two features may both be employed in the same repository, they differ in that one affects the actual hierarchy of the supported node types, while the other works on a node-by-node basis.

3.7.16.1.3 Abstract Node Types

An implementation may make abstract a JCR node type that is not canonically abstract. For example, some implementations might use `nt:file` as is, whereas others might subtype it in order to introduce implementation specific item definitions. Such implementations would therefore designate `nt:file` as abstract.

3.7.16.1.4 Variant Attributes

An implementation may vary the value of a node type or child definition attribute that is explicitly indicated as a variant in the node type definitions given in this

specification. For example, any node type defined in this specification may be either queryable or non-queryable, depending on the implementation. Also, the protected and OPV settings of the properties of the metadata mixins (`mix:title`, `mix:created`, `mix:lastModified`, `mix:language` and `mix:mimeType`) are also variant.

3.7.17 External Node Types

An *external node type* is one defined outside this specification. It may be either an implementation-specific type built into a repository or a node type defined and registered by a user (see §19 *Node Type Management*).

3.7.17.1 Restrictions

The following restrictions apply to all external node types:

- An implementation *must not* allow external node types with node type names in the `nt`, `mix`, `jcr` or `xml` namespaces.
- An implementation may allow external node types which have item definitions in the `jcr` namespace. Such an item definition must only reuse an item definition from a JCR-defined node type.
- Any jcr namespaced item definition *D'* in an external node type *T'* must not be invalid with respect to the JCR-defined definition *D* in the JCR-defined node *T* (with *D*, *D'*, *T* and *T'* as above, see §3.7.6.8 *Item Definitions in Subtypes*).
- Any jcr namespaced item definition in an external node type must be used for a purpose equivalent to its JCR use.
- All custom node types must adhere to semantics of subtyping (see §3.7.6.6 *Semantics of Subtyping*)

3.8 Referenceable Nodes

A repository may support *referenceable nodes*. A node must be referenceable to serve as the target of a *reference property*, which is either a `WEAKREFERENCE` or `REFERENCE`. To be referenceable a node must be of type `mix:referenceable`.

3.8.1.1 mix:referenceable

```
[mix:referenceable]
  mixin
  - jcr:uuid (STRING) mandatory autocreated protected
      INITIALIZE
```

This node type adds an auto-created, mandatory, protected `STRING` property to the node, called `jcr:uuid`, which exposes the identifier of the node. Note that the term "UUID" is used for backward compatibility with JCR 1.0 and does not necessarily imply the use of the UUID syntax, or global uniqueness.

The identifier of a referenceable node must be a *referenceable identifier*. Referenceable identifiers must fulfill a number of constraints beyond the minimum required of standard identifiers (see §3.8.3 *Referenceable Identifiers*).

A reference property is a property that holds the referenceable identifier of a referenceable node and therefore serves as a pointer to that node. The two types of reference properties, `REFERENCE` and `WEAKREFERENCE` differ in that the former enforces referential integrity while the latter does not (see §3.8.2 *Referential Integrity*). A repository may support only `WEAKREFERENCE` or both `WEAKREFERENCE` and `REFERENCE` property types.

3.8.2 Referential Integrity

Given a property `P` with value `V` in workspace `W`:

If `P` is of type `REFERENCE` then there must exist a node in `W` with identifier `V`.

If `P` is of type `WEAKREFERENCE`, no such restriction exists.

In a read-only context the only difference between the types is that a workspace cannot contain a dangling `REFERENCE` while it may contain a dangling `WEAKREFERENCE`.

3.8.2.1 Exceptions to Referential Integrity

In a repository that exposes version storage in content, such as one that supports *full versioning*, an exception is made to the referential integrity rule when the `REFERENCE` property in question is part of the frozen state of a version stored in version storage. In that case the frozen `REFERENCE` property may hold the identifier of a node that is no longer in the workspace (see §3.13.3.7 *References in a Frozen Node*).

3.8.3 Referenceable Identifiers

Every node has an identifier, where an identifier is a string which is the most stable available. A *referenceable* node, however, must have a *referenceable identifier*, which is subject to a number of further constraints:

3.8.3.1 Identifier Assignment

As with any identifier, a referenceable node's identifier must be assigned *at the latest* when the node is first persisted. However, the `jcr:uuid` property of the node must be created immediately upon the node becoming referenceable, which may be upon node creation or upon a later mixin addition. Consequently, the value of the `jcr:uuid` property before the first persist is not guaranteed to be the identifier of the node.

3.8.3.2 Identifier Immutable across Move and Clone

The identifier is immutable during the lifetime of the node, that is, until the node is deleted though a `remove` operation. In particular, the identifier is immutable across `move` and `clone` operations. Note that non-referenceable identifiers are *not* required to be immutable across these operations. As in the non-referenceable case, the referenceable identifier is not immutable across `copy` operations. This operation results in the creation of a new node with a new identifier.

3.8.3.3 Implementation Variations

These are the minimum requirements for a referenceable identifier, but implementations are free to exceed these requirements.

3.9 Shareable Nodes Model

The ability to address the same piece of data via more than one path is a common feature of many content storage systems. In JCR this feature is supported through *shareable nodes*.

Two or more shareable nodes in the same workspace may belong to a shared set. Each node within that set has its own unique path within the workspace but all share exactly the same set of child nodes and properties. This means that while the shared nodes are distinct from a path perspective, they are effectively the same node for purposes of operations that access their common subgraph.

3.9.1 mix:shareable

In order to be shareable, a node must of type `mix:shareable`:

```
[mix:shareable] > mix:referenceable mixin
```

All shareable nodes are referenceable.

3.9.2 Shared Set

Given two distinct shareable nodes *A* and *B* where *A* shares with *B*, the following facts hold:

- *A* and *B* are in the same shared set.
- *B* shares with *A* (sharing is a symmetric relation).
- If *B* shares with *C* then *A* shares with *C* (sharing is a transitive relation).
- If item *I* is a child of *A* then *I* is also a child of *B* and has the same name relative to both *A* and *B*.
- *A* and *B* have the same identifier.
- *A* and *B* are in the same workspace
- *A* and *B* have distinct paths.

3.9.3 Child Nodes of Shared Nodes

Each node in a shared set shares the same child nodes. In particular, the addition or removal of a child from a shared node `N` automatically adds or removes that child from all the nodes in the shared set of `N`.

For example, suppose the following nodes exist:

```
/x
/x/y
/x/y/z
/x/y/z/n1
/x/y/z/n2
```

Suppose a shared node at `/x/a` is created and shares with the shareable node at `/x/y`. Since the children of `/x/y` are automatically added to `/x/a`, a child named "`z`" is automatically added to `/x/a`. Therefore, as a result of creating `/x/a`, the following paths are associated with nodes:

```
/x/a
/x/a/z
/x/a/z/n1
/x/a/z/n2
```

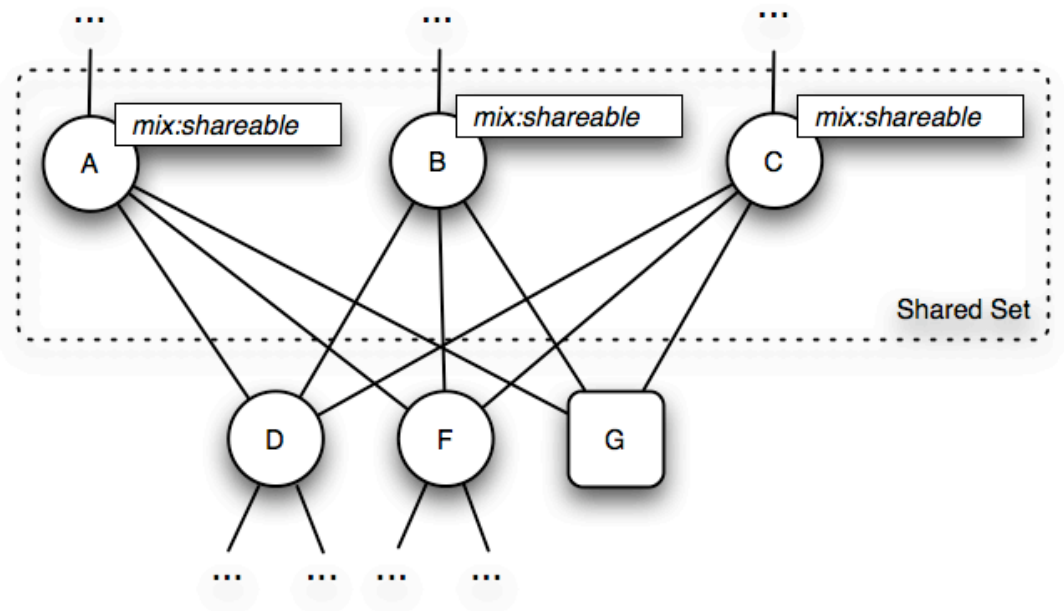
where `/x/a` is a new node that shares with `/x/y`, `/x/a/z` identifies the same node as `/x/y/z`, `/x/a/z/n1` identifies the same node as `/x/y/z/n1`, and `/x/a/z/n2` identifies the same node as `/x/y/z/n2`.

Subsequently, when a new child named "`w`" is added to either `/x/y` or `/x/a`, that child is automatically added to both `/x/y` and `/x/a`. Conversely, when a child named "`w`" is removed from either `/x/y` or `/x/a`, the child named "`w`" is removed from both `/x/y` and `/x/a`.

3.9.4 Properties of Shared Nodes

Each node in shared set shares the same properties and their respective property values. When a change, addition or removal of a property of one node in a shared set is made, that change, addition or removal is immediately reflected in the properties of each node in the shared set.

3.9.5 Shared Nodes Diagram



The above diagram shows a share set of three nodes, `A`, `B` and `C`, which share the child nodes `D` and `F` and the property `G`.

3.9.6 Deemed Path

A descendant item of a shared set will have more than one valid path (assuming the shared set has at least two members). When the parent node or path of such a descendant item is requested, an implementation must choose a *deemed path* to return.

How the deemed path is chosen and its stability both over time and across the set of descendent items is an implementation issue.

In particular, it is permissible for an implementation to choose deemed paths for two sibling items where those paths differ by more than just the last element. It is also permissible for the deemed path of an item to change from one request to the next on the same item within the same session.

Though most implementations are expected to support deemed paths which are more stable than this, flexibility of the deemed path is provided to facilitate implementations which would otherwise not be able to support shareable nodes.

3.9.7 Ancestors of Shared Nodes

Given the following situation:

- Node `A` is an ancestor of node `N`.
- `N` is in the shared set `S`.
- `N'` is also in the shared set `S`.
- `A` is an ancestor of `N`.
- `D` is a descendent of `N`.

The following terminology applies:

- `A` is an *ancestor* of `N` (as usual).
- `A` is a *share-ancestor* of the set `S` and of the individual nodes, `N'` and (trivially), `N`.
- Since `D` is a *descendent* of `N` it is also a *descendent* of every node in `S` (`N'`, for example). We also say that it is a *descendent* of the set `S`.
- Since `A` is a *share-ancestor* of `S` and `D` is a *descendent* of `S`, `A` is an *ancestor* (proper) of `D`.

Note that the term *share-ancestor* does not mean *shared* ancestor. The ancestor (proper) of a member of a shared set is not necessarily an ancestor (proper) of any other member of that set.

3.9.8 Identifiers

When a node is requested by identifier and that identifier references a shared set of nodes with more than one member the repository must return one member of that set. How this node is chosen is an implementation issue. In general, a user that interacts with repositories that support shareable nodes must be prepared to deal with different nodes having the same identifier.

3.9.9 Share Cycle

A *share cycle* occurs when a node is in the same shared set as one of its ancestors. A repository implementation *may* prevent the occurrence of share cycles. In such implementations any method call that would cause a cycle will fail.

3.10 Corresponding Nodes

In a repository with more than one workspace, a node in one workspace *may* have *corresponding nodes* in one or more other workspaces.

Given a repository R with workspaces W_0, W_1,...W_k and a node N_0 in W_0 with identifier I_0 then for each workspace W_x in R, if W_x has a node N_x with identifier I_0, N_x is a corresponding node of N_0. Some corollaries include:

- Every node corresponds to itself.
- A non-shared node has at most one corresponding node per workspace. In repositories that support *shareable nodes* the nodes within a shared-set have the same identifier and therefore a node in another workspace with that identifier will have more than one corresponding node in that workspace (see §3.9 *Shareable Nodes Model*).

Apart from having the same identifier, corresponding nodes need have nothing else in common. They can have different sets of properties and child nodes, for example.

3.10.1.1 Root Node Correspondence

The root nodes of all workspaces in a repository all have the same identifier, and therefore correspond to one another.

3.10.1.2 Correspondence Semantics

The mechanism of correspondence allows two nodes in separate workspaces to be related by a common identifier while maintaining distinct states. This relation is used to model cases where copies of a common content structure must be maintained separately.

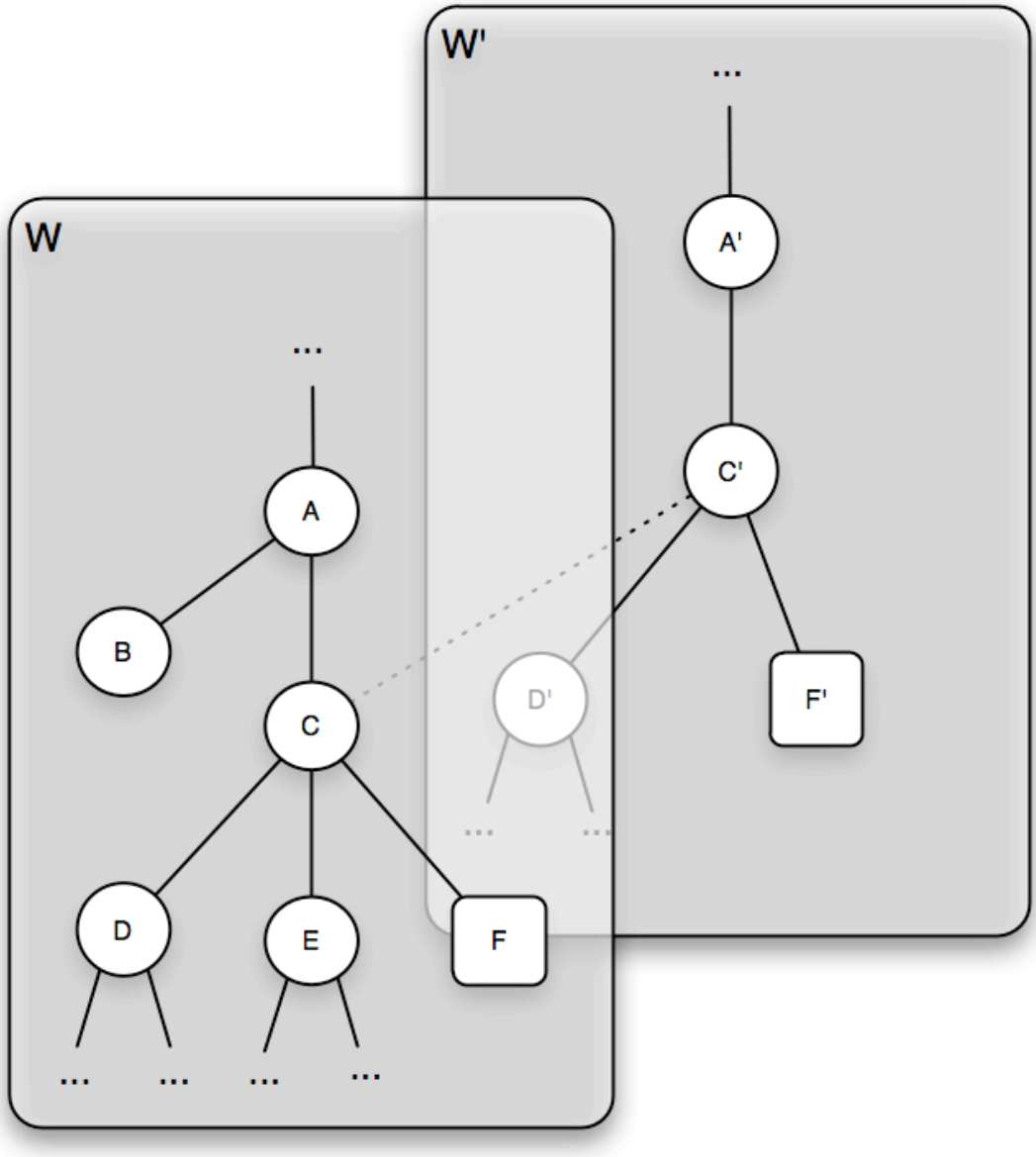
3.10.1.3 Cross-Workspace Operations

JCR provides methods for transferring state between workspaces through *clone* and *update* (see §10.7.2 *Copying Across Workspaces* and §10.8 *Cloning and Updating Nodes*).

3.10.1.4 Versioning and Corresponding Nodes

In systems that support versioning, corresponding nodes in separate workspaces share the same version history (see §3.13.7 *Versioning and Multiple Workspaces*).

3.10.1.5 Corresponding Nodes Diagram



The above diagram depicts two workspaces `W` and `W'`. Node `C` in `W` and node `C'` in `W'` are corresponding nodes. Note that the subgraphs of corresponding nodes may differ, as they do in this case.

3.11 System Node

The location `/jcr:system` is reserved for use as a "system folder". Some implementations may use this location to expose repository-internal data as content.

If a repository exposes node type definitions in content, then those node type definitions *should* be located at `/jcr:system/jcr:nodeTypes` (see §3.7.14 *Node Type Definition Storage*).

If a repository supports full versioning, then it *must* expose the version storage at `/jcr:system/jcr:versionStorage`. If it supports only simple versioning then it *may* expose the version storage (see §3.13.8 *Version Storage*).

Similarly, if an implementation supports activities (see §15.12 *Activities*) or configurations and baselines (see §15.13 *Configurations and Baselines*), the in-content representations of these entities are stored under `/jcr:system/jcr:activities` and `/jcr:system/jcr:configurations`, respectively.

If `/jcr:system` is supported, its node type is left up to the implementation.

3.12 Unfiled Content

Implementers that build a JCR repository on top of an existing repository that supports content objects which exist outside of a hierarchical structure may expose these objects as nodes below `/jcr:system/jcr:unfiled` (see §3.11 *System Node*).

The hierarchical structure below `/jcr:system/jcr:unfiled` is implementation-dependent.

JCR implementers may disallow discovery (listing) of the nodes beneath this folder. In such a case a call to `Node.getNodes`() on the `jcr:unfiled` node would throw a `RepositoryException`.

JCR implementers may expose the nodes below `jcr:unfiled` to search through the query mechanism (see §6 *Query*).

3.13 Versioning Model

Versioning enables a user to record the state of a node and its subgraph and restore that state at a later time. A repository that supports versioning may support either the complete set of versioning features, referred to as *full versioning*, or a specific subset defined here, which is referred to as *simple versioning*. This section describes the concepts, data structures and node types of the full versioning model as well as which parts of that model apply under simple versioning. Discussion of the versioning API and its behavior under both levels of support is found in §15 *Versioning*.

3.13.1 Overview

3.13.1.1 Versionable Nodes

For its state to be recorded in a version, a node must be *versionable*.

3.13.1.2 Check-In

When a versionable node is *checked-in,* a new *version* of that node is created which contains a (typically partial) copy of its subgraph. The part of a node's subgraph that is to be copied to a version is referred to as its *versionable state*. A node's versionable state is determined by the *on-parent-version* attribute of each of its subitems, as defined in its node type (see §3.7.2.5 *On-Parent-Version*).

3.13.1.3 Version History

Once created, a version is stored in a *version history*. Within a given workspace, each non-shared versionable node has its own version history which contains a *version graph* that records the position of each version in relation to its direct predecessor and direct successor versions.

3.13.1.4 Successor and Predecessor

`v'` is a *direct successor* of `v` if and only if `v` is a *direct predecessor* of `v'`.

A version `v'` is an *eventual successor* of a version `v` if and only if `v'` is a direct successor of `v` or there exists a version `v*` such that `v'` is a direct successor of `v*` and `v*` is an eventual successor of `v`.

Similarly, a version `v'` is an *eventual predecessor* of a version `v` if and only if `v'` is a direct predecessor of `v` or there exists a version `v*` such that `v'` is a direct predecessor of `v*` and `v*` is an eventual predecessor of `v`.

When the terms *successor* and *predecessor* are used without qualification they mean *direct successor* and *direct predecessor*, respectively.

3.13.1.5 Simple and Full Versioning

Under simple versioning, each new version is always added as the unique direct successor of the previous version, thus maintaining a linear series of versions.

Under full versioning, a new version may be added as the direct successor of a version that already has another direct successor, thus producing a *branch*. A new version may also be added as the direct successor of more than one existing version, thus producing a *merge*.

3.13.1.6 Version Storage

Version histories are stored in a repository-wide *version storage*. Under full versioning this store is exposed both through the Java objects of the versioning API as well as in a read-only subgraph reflected in each workspace. Within that subgraph version histories are represented as nodes of type `nt:versionHistory` and versions as nodes of type `nt:version`. Under simple versioning, the version store is exposed through the versioning API but is not required to be exposed as a node subgraph.

3.13.1.7 Check-Out

Once checked-in, a versionable node and its versionable subgraph become *read-only*. To alter a checked-in node or its versionable subgraph, the node must first be *checked-out*. It can then be changed and checked-in again, creating a new version.

3.13.1.8 Restore

A versionable node and its versionable subgraph can also be *restored* to the state recorded in one of its versions.

3.13.2 Versionable Nodes

Under simple versioning, a versionable node must be `mix:simpleVersionable`. Under full versioning, it must be `mix:versionable`.

3.13.2.1 mix:simpleVersionable

```
[mix:simpleVersionable] mixin
  - jcr:isCheckedOut (BOOLEAN) = 'true'
      mandatory autocreated protected IGNORE
```

The `mix:simpleVersionable` type exposes the node's *checked-out status* as a `BOOLEAN` property.

3.13.2.2 Checked-In or Checked-Out

A new version of a versionable node is created by *checking-in* a versionable node (see §15.2 *Check-In: Creating a Version*). In this state the node and its *versionable subgraph* are *read-only* (see §15.2.2 *Read-Only on Check-In*). The node can then be *checked-out* (see 15.3 *Check-Out*), at which point it becomes writable again.

Under both simple and full versioning, this status is accessible through `VersionManager.isCheckedOut` (see §15.3.1.1 *Testing for Checked-Out Status*) and the `BOOLEAN` property `jcr:isCheckedOut`.

3.13.2.3 mix:versionable

```
[mix:versionable] > mix:simpleVersionable, mix:referenceable
 mixin
 - jcr:versionHistory (REFERENCE) mandatory protected IGNORE
     < 'nt:versionHistory'
 - jcr:baseVersion (REFERENCE) mandatory protected IGNORE
     < 'nt:version'
 - jcr:predecessors (REFERENCE) mandatory protected multiple
     IGNORE < 'nt:version'
 - jcr:mergeFailed (REFERENCE) protected multiple ABORT
     < 'nt:version'
 - jcr:activity (REFERENCE) protected IGNORE < 'nt:activity'
 - jcr:configuration (REFERENCE) protected IGNORE
     < 'nt:configuration'
```

The mixin `mix:versionable` is a subtype of `mix:simpleVersionable` and `mix:referenceable`, and adds properties exposing a number of additional versioning-related attributes.

3.13.2.4 Version History Reference

Apart from nodes within the same shared set, which share the same version history, each versionable node within a persistent workspace has its own version history.

Under both simple and full versioning the version history of a node is accessed through `VersionManager.getVersionHistory`, which returns a `VersionHistory` object (see §15.1.1 *VersionHistory Object*). Under full versioning it is also

represented by the REFERENCE property jcr:versionHistory, which points to an nt:versionHistory node (see §3.13.5.1 *nt:versionHistory*).

3.13.2.5 Base Version Reference

Each versionable node has a *base version* within its version history. When a new version of a node is created, it is placed in that node's version history as a direct successor of the base version. That version itself then becomes the new base version (see §3.13.6.2 *Base Version*).

Under simple versioning, the base version of a versionable node is always the *most recent version* in its version history.

Under full versioning, corresponding versionable nodes in different workspaces, while having the same version history, may have different base versions *within* that history. Therefore, the base version of a full versionable node may not be the most recent version in that node's version history (see §3.13.6.2 *Base Version*).

The base version is accessed through VersionManager.getBaseVersion (see §15.1.2 *Getting the Base Version*) which returns a Version object (see 15.2.1 *Version Object*). Under full versioning the connection to the base version is also represented by the REFERENCE property jcr:baseVersion, which points to an nt:version node (see §3.13.2.3 *mix:versionable*).

3.13.2.6 Predecessors

Under full versioning, a versionable node N has one or more versions in its version history that will become direct predecessors of the new version V created on the next check-in of N. For convenience these versions can also be referred to as the direct predecessors of N (i.e., not just the direct predecessor of the hypothetical V).

The base version of N is always *one* of these direct predecessors, but N may have additional direct predecessors as well. If so, on check-in of N, all of these become direct predecessors of the newly created version V, thus forming a merge within the version graph (see §15.9 *Merge*).

A node's direct predecessors are exposed by the multi-value REFERENCE property jcr:predecessors (see §3.13.2.3 *mix:versionable*) which points to one or more nt:version nodes (see §3.13.3.1 *nt:version*).

There is no dedicated API for accessing the direct predecessors of a *versionable node*; access is provided through the property only (this should not be confused with access to the direct predecessors of a *version*, which is exposed through the API, see §3.13.2.6 *Predecessors*).

Under simple versioning, the jcr:predecessors attribute is not needed (and hence not present on mix:simpleVersionable) since a versionable node will only ever have one direct predecessor, which is its base version.

3.13.2.7 Merge Failed

Under full versioning, `jcr:mergeFailed` is a multi-value `REFERENCE` property that is used to mark merge failures (see §15.9 *Merge*). Under simple versioning, merges are not supported. There is no dedicated API for accessing merge failures; access is provided only through this property.

3.13.2.8 Activity

Under full versioning, `jcr:activity` is a `REFERENCE` property used to support the activities feature (see §15.12 *Activities*). Under simple versioning, activities are not supported. There is no dedicated API for retrieving the activity associated with a given versionable node; access is provided only through this property.

3.13.2.9 Configuration

Under full versioning, `jcr:configuration` is a `REFERENCE` property used to support the configurations and baselines feature (see §15.13 *Configurations and Baselines*). Under simple versioning, configurations are not supported. There is no dedicated API for retrieving the configuration associated with a given versionable node; access is provided only through this property.

3.13.3 Versions

Under simple versioning, a version is represented by a `Version` object (see §15.2.1 *Version Object*) and the attributes of the version are accessible only through methods of that class.

Under full versioning a version is represented by both a `Version` object and a node of type `nt:version` within the version storage (see §3.13.7 *Version Storage*). The attributes of a version are accessible both through methods of the `Version` class and through the properties of `nt:version`.

Each version has a name unique within its version history that is assigned automatically on creation of the version. The format of the name is implementation-dependant. Under full versioning this is the name of the `nt:version` node representing the version. Under simple versioning this is simply the name returned when `Item.getName()` is called on the `Version` object.

`Version` is a subclass of `Node`. However, since under simple versioning a version is not represented by a node, most of the `Node` methods inherited by `Version` are not required to function. The single exception is `Item.getName()` as mentioned above. Under full versioning the `Node` methods inherited by `Version` function as expected on the `nt:version` node.

Under full versioning the `nt:version` nodes representing the versions within a given history are always created as direct child nodes of the `nt:versionHistory` node representing that history.

3.13.3.1 nt:version

```
[nt:version] > mix:referenceable
  - jcr:created (DATE) mandatory autocreated protected
      ABORT
```

```
- jcr:predecessors (REFERENCE) protected multiple ABORT
    < 'nt:version'
- jcr:successors (REFERENCE) protected multiple ABORT
    < 'nt:version'
- jcr:activity (REFERENCE) protected ABORT
    < 'nt:activity'
+ jcr:frozenNode (nt:frozenNode) protected ABORT
```

`nt:version` inherits the `STRING` `jcr:uuid` from `mix:referenceable`, making every `nt:version` node referenceable. Additionally, it defines properties that expose the following attributes.

3.13.3.2 Creation Date

Each version records its creation date, which is accessible through `Version.getCreated` and, under full versioning, through the `jcr:created` `DATE` property of `nt:version`.

3.13.3.3 Predecessors

Each version has zero or more direct predecessor versions within its version history, accessible though `Version.getPredecessors`. Under simple versioning, a version will have either zero direct predecessors (if it is the root version of a history) or one direct predecessor. Under full versioning, a version may have zero, one, or more direct predecessors, which are exposed through the `jcr:predecessors` multi-value `REFERENCE` property of `nt:version`.

3.13.3.4 Successors

Each version has zero or more direct successor versions within its version history, accessible though `Version.getSuccessors`. Under simple versioning, a version will have either zero or one direct successors. Under full versioning, a version may have zero, one, or more direct successors, which are exposed through the `jcr:successors` multi-value `REFERENCE` property of `nt:version`.

3.13.3.5 Frozen Node

Each version records the versionable state of its versionable node at the time of check-in in a *frozen node*, attached to the version and accessed though `Version.getFrozenNode`. Under simple versioning, the frozen node is isolated, having no parent in any workspace. Under full versioning, it is a subnode of the `nt:version` node in the version storage subgraph. In both cases the node is of type `nt:frozenNode` (see §3.13.4.1 *nt:frozenNode*).

3.13.3.6 Activity

Under full versioning a version may be bound to an activity. This relationship is recorded by the `jcr:activity` `REFERENCE` property of `nt:version` pointing to `nt:activity` node that represents the activity under which this version was created (see §15.12 *Activities*). Under simple versioning activities are not supported.

3.13.4 Frozen Nodes

When a version is created, the *versionable subgraph* of its versionable node is copied to a *frozen node* within the new version. On check-in the child nodes and properties that constitute the versionable subgraph are copied and placed as child items of the frozen node under the same names as they had under the versionable node.

Which child nodes and properties are copied, and in the case of child nodes, the depth of the subgraph to be copied, constitutes the versionable state of the node and is determined by the on-parent-version settings defined in the node type of the versionable node (see §3.7.2.5 *On-Parent-Version*).

Regardless of the node type of the original versionable, all frozen nodes are of the type `nt:frozenNode`. Under both simple and full versioning, the frozen node of a version is accessible directly from the `Version` object (see §15.2.1 *Version Object*). Under full versioning, the frozen node is also accessible as the child node `jcr:frozenNode` of the `nt:version` node. Under simple versioning the frozen node is a `Node` object but does not have a parent, and consequently methods called on the frozen node that depend upon having a parent will throw an exception. Under full versioning the frozen node's parent is the `nt:version` node to which it belongs.

A frozen node always has the name `jcr:frozenNode`. Under full versioning this is the name under which it exists as a child of its `nt:version` node. Under simple versioning this is simply the name returned when `Item.getName()` is called on the frozen node.

3.13.4.1 nt:frozenNode

```
[nt:frozenNode] orderable
  - jcr:frozenPrimaryType (NAME) mandatory autocreated
      protected ABORT
  - jcr:frozenMixinTypes (NAME) protected multiple ABORT
  - jcr:frozenUuid (STRING) protected ABORT
  - * (UNDEFINED) protected ABORT
  - * (UNDEFINED) protected multiple ABORT
  + * (nt:base) protected sns ABORT
```

`nt:frozenNode` defines the following properties and child nodes.

3.13.4.2 Frozen Primary Type

`jcr:frozenPrimaryType` is a `NAME` property that stores the primary node type of the versionable node.

3.13.4.3 Frozen Mixin Types

`jcr:frozenMixinTypes` is a multi-value `NAME` property that stores the mixin types of the versionable node, if any.

3.13.4.4 Frozen Identifier

`jcr:frozenUuid` is a `STRING` property that stores the referenceable identifier of the versionable node. Note that the term "UUID" is used for backwards compatibility with JCR 1.0 and does not necessarily the use of the UUID syntax, or global uniqueness.

3.13.4.5 Residual Properties and Child Nodes

A set of residual definitions are defined for the copies of the properties and child nodes that make up the versionable state of the versionable node. In repositories that do not support residual item definitions (see §3.7.2.1.2 *Item Definition Name and Residual Definitions*) these must be implemented as a special case for the frozen node.

3.13.4.6 References in a Frozen Node

A `REFERENCE` property stored in the frozen node of a version does not enforce referential integrity. Under simple versioning this follows from the fact that a `REFERENCE` within a frozen node does not appear within the workspace in any case, it only appears when the frozen node is explicitly retrieved through the `Version` object (see §15.2.1 *Version Object*). Under full versioning a `REFERENCE` within a frozen node will appear in the workspace within the read-only version storage (see §3.13.8 *Version Storage*) so the referential integrity requirement must be lifted as a special case.

3.13.5 Version History

Under simple versioning a version history is represented by a `VersionHistory` object (see §15.1.1 *VersionHistory Object*). Under full versioning a version history is represented by both a `VersionHistory` object and a node of type `nt:versionHistory`.

A version history is created upon creation of a new versionable node. Under full versioning this results that an `nt:versionHistory` node being created automatically within the version storage subgraph in an implementation determined location and with an implementation-determined name.

`VersionHistory` is a subclass of `Node`. However, since under simple versioning a version history is not represented by a node, the `Node` methods inherited by `VersionHistory` are not required to function. Under full versioning those methods will function as expected on the `nt:versionHistory` node representing the history.

3.13.5.1 nt:versionHistory

```
[nt:versionHistory] > mix:referenceable
  - jcr:versionableUuid (STRING) mandatory autocreated
      protected ABORT
  - jcr:copiedFrom (WEAKREFERENCE) protected ABORT
      < 'nt:version'
  + jcr:rootVersion (nt:version) = nt:version mandatory
      autocreated protected ABORT
  + jcr:versionLabels (nt:versionLabels)
```

```
      = nt:versionLabels protected ABORT
  + * (nt:version) = nt:version protected ABORT
```

This type inherits the `STRING` `jcr:uuid` from `mix:referenceable`, making every `nt:versionHistory` node referenceable. It also defines properties and child nodes representing the following attributes.

3.13.5.2 Root Version

Each version history has a *root version*, which is a null version that stores no state; it simply serves as the eventual predecessor of all subsequent versions.

The root version is accessed through `VersionHistory.getRootVersion` (see 15.1.1.1 *Root Version*). Under full versioning it is also represented as an `nt:version` child node of the `nt:versionHistory` node, called `jcr:rootVersion`.

3.13.5.3 Versions

In addition to the root version, a version history also holds all the versions that have been created through check-ins of its versionable node (or nodes, in the case of corresponding or shared nodes). These versions are accessible through `VersionHistory.getAllVersions`, `VersionHistory.getAllLinearVersions` and `VersionHistory.getVersion`.

Under full versioning these versions are also accessible as `nt:version` child nodes of the `nt:versionHistory` node, under their respective implementation-determined names.

3.13.5.4 Versionable Identifier

Each version history also stores the identifier of its versionable node (or nodes in the case of corresponding or shared nodes).

This attribute is accessible through `VersionHistory.getVersionableIdentifier` and, under full versioning, through the `STRING` property `jcr:versionableUuid` of `nt:versionHistory`. Note that the term "UUID" is used for backward compatibility with JCR 1.0 and does not necessarily imply the use of the UUID syntax, or global uniqueness.

3.13.5.5 Version Labels

A version label is a JCR name, unique among the labels within a version history, that identifies a single version within that history. A version can have zero or more labels.

Labels are can be assigned, removed and retrieved through the API. Versions can also be retrieved by label (see §15.4 *Version Labels*).

Under full versioning labels are also exposed through an `nt:versionLabels` child node of `nt:versionHistory` called `jcr:versionLabels`. This node holds a set of reference properties that record all labels that have been assigned to the versions within this version history. Each label is represented by a single reference property which uses the label itself as its name and points to that `nt:version`

child node to which the label applies (see §15.4.1.2 *Adding a Version Label*). The `nt:versionLabels` node type has the following definition:

```
[nt:versionLabels]
  - * (REFERENCE) protected ABORT < 'nt:version'
```

3.13.5.6 Copied From

When a full versionable node is copied to a new location and the repository preserves the `mix:versionable` mixin (see §10.7 *Copying Nodes*), the copy gets a new version history and that history records the base version of the node from which the copy was created. This information, known as the node's *lineage*, is recorded in the `jcr:copiedFrom` `WEAKREFERENCE` property of `nt:versionHistory` (see §15.1.4 *Copying Versionable Nodes and Version Lineage*).

3.13.6 Version Graph

The versions within a version history form a *version graph* where the versions are the vertices and each *direct successor/direct predecessor* pair are joined by a directed edge.

Under simple versioning, branching and merging are not supported, so the version graph is always a linear series of successive versions.

Under full versioning branching and merging are supported, so a version may have multiple direct predecessors, multiple direct successors, or both. Also, under full versioning, the version graph is exposed in content with `nt:version` nodes as the vertices and the edges defined by the `REFERENCE` properties `jcr:successors` and `jcr:predecessors`.

3.13.6.1 Root Version

The version graph always contains at least the root version. This is a null version that stores no state and simply serves as the eventual predecessor of all subsequent versions. Its frozen node does not contain any state information about the versionable other than the node type and identifier information found in the properties `jcr:frozenPrimaryType`, `jcr:frozenMixinTypes`, and `jcr:frozenUuid`.

3.13.6.2 Base Version

For a given versionable node, one version within its version history is its *base version.* Conceptually, the base version of a versionable node is that version relative to which the current state of the checked-out versionable node constitutes a versionable change.

On check-in, the newly created version becomes a direct successor of the current base version and then itself becomes the *new* base version.

Under full versioning, corresponding versionable nodes in different workspaces, while sharing the same version history, may have distinct base versions within that history. This means that when one versionable node is checked-in, its version will become direct successor of a particular existing version but when one

of its correspondees is checked in, that new version may become direct successor to a *different* existing version within the same version history.

Under simple versioning the linearity of the version graph guarantees that the current base version is always the most recent version. On check-in it becomes the unique direct predecessor of the newly created version, which then becomes the new base version.

Under full versioning, the presence of branches and merges means that the current base version for a given versionable node is not guaranteed to be the most recent version.

3.13.7 Versioning and Multiple Workspaces

As mentioned (see 3.13.2.5 *Base Version Reference* and 3.13.6.2 *Base Version*), under full versioning, corresponding versionable nodes in different workspaces all have a single common version history, though within that history each may have a distinct base version at any given time.

The intended semantics of the correspondence relationship is that corresponding nodes represent copies of a common content structure which while identical at one level (i.e., as determined by *identifier*), can be maintained in distinct states (see §3.10.1.2 *Correspondence Semantics*).

The relation of corresponding versionable nodes to the same version history reflects these semantics in that when different copies of the same content entity are changed and checked-in from different workspaces the resulting versions are all recorded within the same version graph and, depending on their individual base versions, extending that graph at different points.

3.13.8 Version Storage

Version histories are stored in a single, repository-wide version storage mutable and readable through the versioning API.

Under full versioning the version storage data must, additionally, be reflected in each workspace as a protected (see §3.7.2.2 *Protected*) subgraph of nodes of type `nt:versionHistory` located below `/jcr:system/jcr:versionStorage` (see §3.11 *System Node*). Because it is protected, the subgraph cannot be altered through the core write methods of JCR (see §10.2 *Core Write Methods*) but only through though the methods of the versioning API.

Though the general repository-wide version history is reflected in each workspace, the access that a particular `Session` gets to that subgraph is governed by that `Session`'s authorization, just as for any other part of the workspace.

The `nt:versionHistory` nodes below `/jcr:system/jcr:versionStorage`, may all be direct children of the `jcr:versionStorage` node or may be organized in a deeper substructure of intervening subnodes that sort the version histories by some implementation-specific criteria.

The node type of the node jcr:versionStorage and any structural nodes used within the subgraph are left up to the implementation.

3.13.9 Versionable State

The *versionable state* of a versionable node N is typically a *subset* of its subgraph. The extent of this subset is defined in the node type of N through the *on-parent-version attribute* (*OPV*) of each of its child items (see §3.7.2.5 *On-Parent-Version*).

A frozen node F storing the versionable state of the node N is constructed as follows:

1. The primary type of *N* is copied from the jcr:primaryType property of *N* to the jcr:frozenPrimaryType property of *F*.
2. The mixin types of *N* (if any) are copied from the jcr:mixinTypes property of *N* to the jcr:frozenMixinTypes property of *F*.
3. The referenceable identifier of *N* is copied from the jcr:uuid property of *N* to the jcr:frozenUuid property of *F*.
4. For each property P of N other than jcr:primaryType, jcr:mixinTypes and jcr:uuid where

   - P has an OPV of COPY *or*
   - P has an OPV of VERSION,

   a copy of P is added to the frozen node, preserving its name and value (or values).
5. For each child node C of N where

   - C has an OPV of COPY,

   a copy of the entire subgraph rooted at C (regardless of the OPV values of the sub-items) is added to the frozen node, preserving the name of C and the names and values of all its sub-items.

   In a repository that supports orderable child nodes, the relative ordering of the set of child nodes C that are copied to the frozen node is preserved. As is the ordering within the subgraph of each of these child nodes C.
6. For each child node C of N where:

   - C has an OPV of VERSION

   Under simple versioning, the same behavior as COPY.

   Under full versioning, if C is not mix:versionable, the same behavior as COPY.

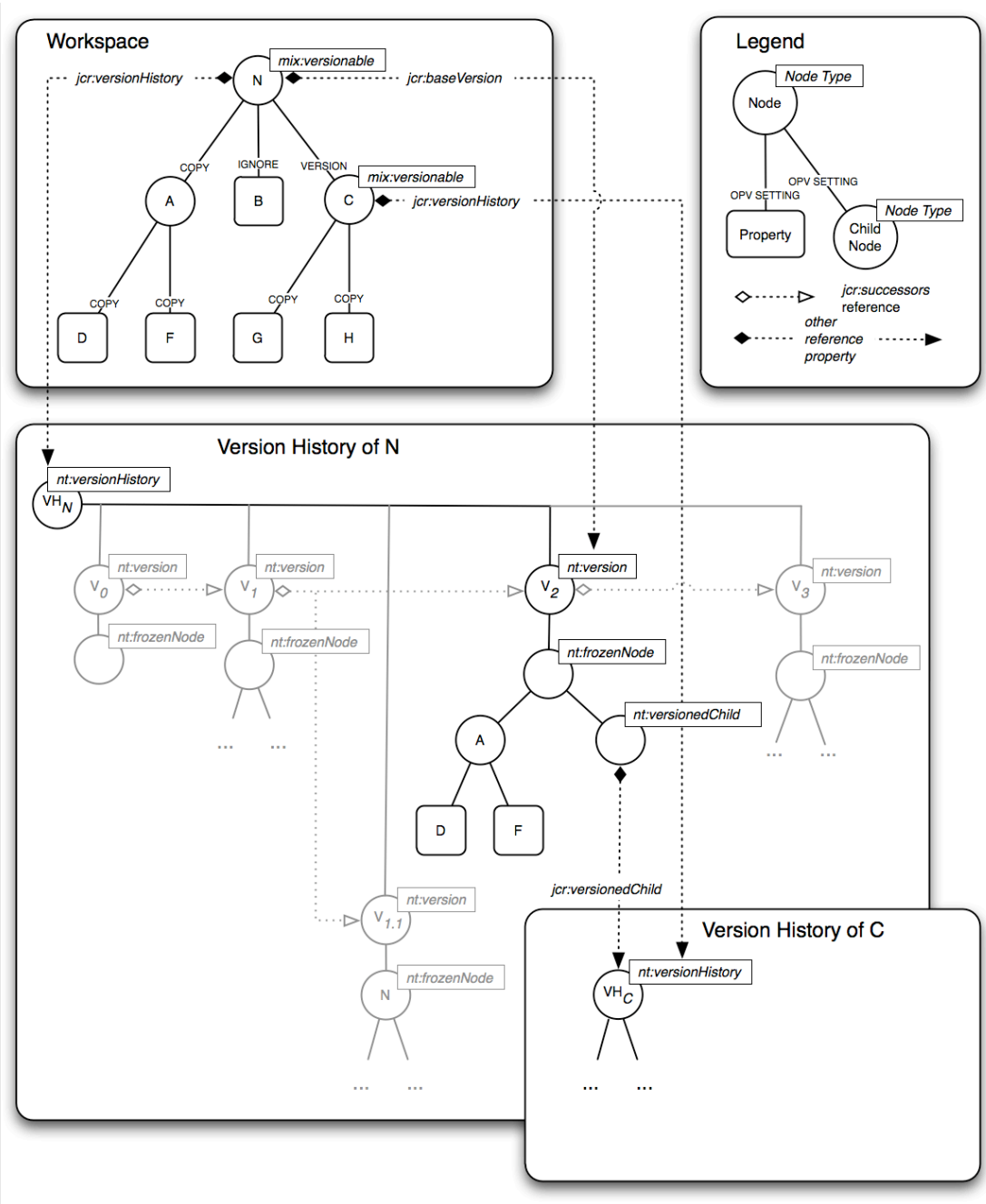
   Under full versioning, if C is mix:versionable, then a special nt:versionedChild node with a reference to the version history of C is substituted in place of C as a child of the frozen node. The

`nt:versionedChild` node has the same name as `C` and, in a repository that supports orderable child nodes, the relative ordering of any such child node `C` is preserved. The definition of `nt:versionedChild` is:

```
[nt:versionedChild]
  - jcr:childVersionHistory (REFERENCE) mandatory
      autocreated protected ABORT
      < 'nt:versionHistory'
```

The property `jcr:childVersionedHistory` points to the `nt:versionHistory` of `C`.

3.13.10 Full Versioning Diagram

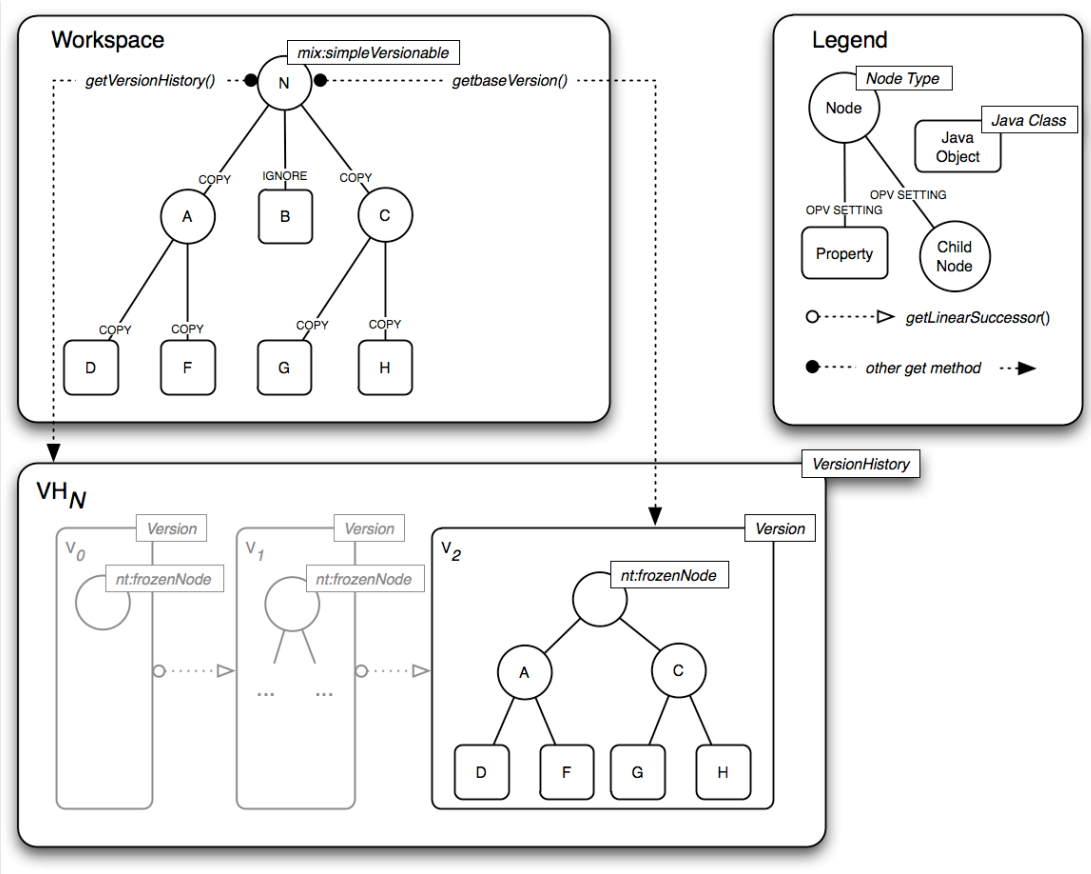


The above diagram depicts the main elements of the full versioning model. A workspace contains a versionable node N with child nodes A and C and property B. The on-parent-version settings of each child are shown. A has an OPV of COPY while B has an OPV of IGNORE. C is itself versionable and has an OPV of VERSION.

VH_N is the nt:versionHistory node holding the version history of N which, in the situation depicted, consists of the versions V_0, V_1, $V_{1.1}$, V_2 and V_3, where V_0 is the root version, V_1 is the successor of V_0, $V_{1.1}$ and V_2 are both successors of V_1 (constituting a *branch*) and V_3 is the successor of V_2.

V_2 is the current base version of N and is shown in detail. As defined by the OPV values of the children of N, V_2 contains a partial copy of N's subtree in its frozen node. This partial copy consists of the subtree rooted at A (since A in N has an OPV of COPY) but does not include the property B (since B in N has an OPV of IGNORE). Since C is itself versionable it has its own, separate, version history at VH_C and since it has an OPV of VERSION, C is represented in the frozen subtree of V_2 by an nt:versionedChild node that points to VH_C.

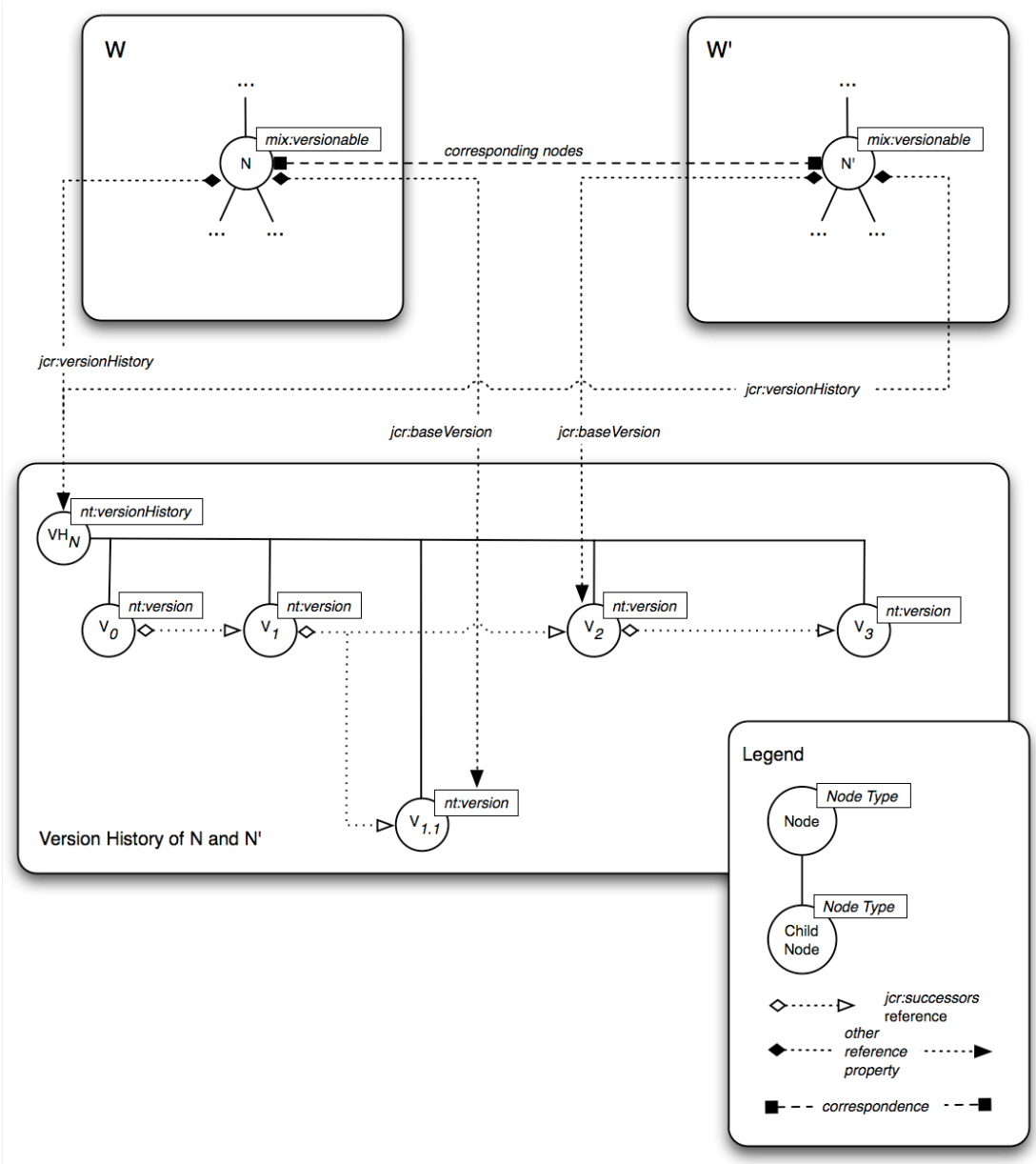
3.13.11 Simple Versioning Diagram



The above diagram depicts the main elements of the simple versioning model. Unlike under full versioning, the connections from the versionable node to the version history and the base version are not mediated by reference properties but

through API methods only. As well, the version history and its contained versions are represented only by Java object instances (of classes `VersionHistory` and `Version`, respectively) not by nodes. Finally, as the diagram indicates, under simple versioning the version history is always linear and the `VERSION` on-parent-version setting and associated structures are not supported.

3.13.12 Versioning and Corresponding Nodes Diagram



The above diagram depicts two workspaces, `W` and `W'` containing corresponding nodes `N` and `N'`, both versionable. Because the two versionable nodes correspond, they share the same version history, as shown by their respective `jcr:versionHistory` references both pointing to the same `nt:versionHistory` node. Despite sharing the same history, at any given time, corresponding nodes may have distinct base versions within that history. In this diagram the base version of `N` is $V_{1.1}$ while the base version of `N'` is V_2.

4 Connecting

4.1 Repository Object

To begin using a repository, an application must acquire a `Repository` object.

Access to a `Repository` object may be provided through a number of standard Java naming and discovery mechanisms, but *must* at the minimum be provided through an implementation of the `RepositoryFactory` interface.

Any implementation of `RepositoryFactory` must have a zero-argument public constructor. Repository factories may be installed in an instance of the Java platform as extensions, that is, jar files placed into any of the usual extension directories. Factories may also be made available by adding them to the applet or application class path or by some other platform-specific means.

A repository factory implementation should support the Java Standard Edition Service Provider mechanism[9], that is, an implementation should include the file `META-INF/services/javax.jcr.RepositoryFactory`. This file contains the fully qualified name of the class that implements `RepositoryFactory`.

Once the `RepositoryFactory` is acquired, the `Repository` object itself is acquired through

```
Repository RepositoryFactory.getRepository(Map parameters)
```

which attempts to retrieve a `Repository` object using the given parameters.

Parameters are passed in a `Map` of `String` key/value pairs. The keys are not specified by JCR and are implementation specific. However, vendors should use keys that are namespace qualified in the Java package style to distinguish their key names. Alternatively, a client may request a default repository instance by passing a `null`.

The implementation must return `null` if a default repository instance is requested and the factory is not able to identify such a repository or if parameters are passed and the factory does not understand them. See the associated Javadoc for example connection code.

4.1.1 Example Repository Acquisition

An application may explicitly specify the repository factory implementation. For example:

```
Map parameters = new HashMap();
```

[9] See `http://java.sun.com/j2se/1.4.2/docs/guide/jar/jar.html#Service%20Provider`.

```
parameters.put("com.vendor.address",
               "vendor://localhost:9999/repo");

RepositoryFactory factory = (RepositoryFactory)
  Class.forName("com.vendor.RepositoryFactoryImpl");

Repository repo = factory.getRepository(parameters);
```

Some implementations may allow acquisition of a `RepositoryFactory` through the `ServiceLoader` in Java SE 6. For example:

```
Map parameters = new HashMap();

parameters.put("com.vendor.address",
               "vendor://localhost:9999/repo");

Repository repo = null;

for (RepositoryFactory factory :
  ServiceLoader.load(RepositoryFactory.class)) {
    repo = factory.getRepository(parameters);
    if (repo != null) {
        // factory accepted parameters
        break;
    }
}
```

Note that in Java SE prior to version 6, one may use the class `javax.imageio.spi.ServiceRegistry` to look up the available `RepositoryFactory` implementations.

4.1.2 Thread Safety

A repository implementation must provide thread-safe implementations of all the methods of the `RepositoryFactory` and `Repository` interfaces. A repository implementation is not required to provide thread-safe implementations of any other interface. As a consequence, an application which concurrently or sequentially operates against objects having affinity to a particular `Session` through more than one thread must provide synchronization sufficient to ensure no more than one thread concurrently operates against that `Session` and changes made by one thread are visible to other threads.

4.2 Login

Interaction with the repository begins with the user acquiring a `Session` through a call to `Repository.login`. In the most general case, the client supplies a `Credentials` object and a workspace name:

```
Session Repository.login(Credentials credentials,
                         String workspaceName).
```

Other signatures of `login` are also provided (see §4.2.4 *External Authentication*).

4.2.1 Credentials

The `Credentials` interface is an empty marker for the object that carries the information necessary to authenticate and authorize the user. A repository may use the supplied `SimpleCredentials` implementation or its own implementation.

4.2.2 Guest Credentials

`GuestCredentials` is used to acquire an anonymous session.

4.2.3 Workspace Name

The `workspaceName` passed on `login` identifies one of the persistent workspaces of the repository. More than one `Session` can be simultaneously bound to the same persistent workspace.

4.2.4 External Authentication

By providing a signature of `Repository.login` that does not require `Credentials`, the content repository allows for authorization and authentication to be handled by JAAS (or another external mechanism) if the implementer so chooses.

To use such an external mechanism to create sessions with end-user identity, invocations of the `Repository.login` method that do not specify `Credentials` (i.e., either a `null Credentials` is passed or a signature without the `Credentials` parameter is used) should obtain the identity of the already-authenticated user through that external mechanism.

4.3 Impersonate

A client may also open a new `Session` from within an existing one using

```
Session Session.impersonate(Credentials credentials).
```

The returned `Session` is bound to the same workspace as the current `Session`, though it may (and typically will) have a different authorization. The implementation is free to take both the supplied `Credentials` and the authorization of the current `Session` into account in determining the authorization of the returned `Session`.

4.4 Session

The `Session` object is granted a set of permissions toward the specified persistent workspace. These are determined by the `Session`'s credentials combined with any access control restrictions, either JCR-defined or implementation-specific, which may apply (see §9.1 *Permissions*).

4.4.1 User

Each `Session` has a user ID, accessed through

```
String Session.getUserID().
```

How the user ID is set is up to the implementation. It may be passed in as part of the `Credentials` or it may be acquired in some other way. This method is free to return an "anonymous user ID" or `null`.

4.4.2 Attributes

A `Session` may have arbitrary, implementation-specific named attributes bound to its `Credentials`. The method

```
String[] Session.getAttributeNames()
```

returns the set of attribute names, and the method

```
Object Session.getAttribute(String name)
```

returns the value of a named attribute.

4.4.3 Session to Repository

The `Repository` object through which a `Session` was acquired is retrieved with

```
Session.getRepository().
```

4.4.4 Live Status

The method

```
boolean Session.isLive()
```

is used to check whether a `Session` object represents a live, logged-in session.

4.4.5 Logout

A `Session` is closed using

```
void Session.logout().
```

4.5 Workspace

Though more than one `Session` can be bound to the same *persistent workspace*, each `Session` object has a single distinct corresponding `Workspace` object that *represents* the actual persistent workspace to which the `Session` is bound. A `Workspace` object can be thought of as a *view* on to the persistent workspace as seen through the permissions granted to its corresponding `Session` (see §10 *Writing*).

4.5.1 Session to Workspace

```
Workspace Session.getWorkspace().
```

returns the Workspace object representing the actual persistent workspace to which a `Session` is bound.

Despite their one-to-one relationship, `Session` and `Workspace` are defined as distinct interfaces in order to separate two types of write behavior: *transient* vs. *immediately persistent*, though this distinction is only strictly relevant in writable repositories.

4.5.2 Workspace to Session

```
Session Workspace.getSession()
```

returns the `Session` object to which a `Workspace` object is bound.

4.5.3 Workspace Name

```
String Workspace.getName()
```

returns the name of the persistent workspace represented by a `Workspace` object.

4.5.4 Accessible Workspaces

```
String[] Workspace.getAccessibleWorkspaceNames()
```

returns an array holding the names of all persistent workspaces accessible from a `Workspace` object. Accessibility is determined by the permissions granted to the `Session` to which the `Workspace` object is bound. In order to access one of the listed workspaces, the user performs another `Repository.login`, specifying the name of the desired workspace, and receives a new `Session` object.

5 Reading

There are three types of read access which a session may have with respect to a particular item, depending on that session's permissions: direct access, traversal access and query access.

5.1 Direct Access

Direct access to an item means being able to retrieve it by absolute and relative path and, in the case of nodes, by identifier.

Let `p(x)` return the normalized absolute path of item `x`, `p(x, y)` return the normalized relative path from item `x` to item `y` and, `id(x)` return the identifier of node `x`.

For any session `S` and node `N`, the statements below must be either all true or all false. If they are all true then `S` has direct access to `N`, if they are all false then `S` does not have direct access to `N`:

- `S.getItem(p(N))` returns `N`.
- `S.itemExists(p(N))` returns `true`.
- `S.getNode(p(N))` returns `N`.
- `S.nodeExists(p(N))` returns `true`.
- `S.getNodeByIdentifier(id(N))` returns `N`.
- If `N` is the primary item of a node `M` then `M.getPrimaryItem()` returns `N`.
- If `N` is the root node of the workspace then `S.getRootNode()` returns `N`.
- For all nodes `M` to which `S` has direct access, `M.getNode(p(M,N))` returns `N`.
- For all nodes `M` to which `S` has direct access, `M.hasNode(p(M,N))` returns `true`.

For any session `S` and property `P`, the statements below must be either all true or all false. If they are all true then `S` has direct access to `P`, if they are all false then `S` does not have direct access to `P`:

- If there is no node at the path `p(P)` to which S has read access then
  - `S.getItem(p(P))` returns `P` and
  - `S.itemExists(p(P))` returns `true`.
- `S.getProperty(p(P))` returns `P`.
- `S.propertyExists(p(P))` returns `true`.
- `S` has read access to the value of `P` (see §9.1 *Permissions*).
- If `P` is the primary item of a node `N` then `N.getPrimaryItem()` returns `P`.

- For all nodes `M` to which `S` has direct access, `M.getProperty(p(M,P))` returns `P`.
- For all nodes `M` to which `S` has direct access, `M.hasProperty(p(M,P))` returns `true`.

5.1.1 Getting the Root Node

The root node of the workspace can be acquired with

```
Node Session.getRootNode().
```

5.1.2 Testing for Existence by Absolute Path

The existence of a node or property at a particular absolute path can be tested for with

```
boolean Session.itemExists(String absPath),
```

```
boolean Session.nodeExists(String absPath)
```
and

```
boolean Session.propertyExists(String absPath).
```

5.1.3 Access by Absolute Path

Nodes and properties can be acquired by absolute path with

```
Item Session.getItem(String absPath),
```

```
Node Session.getNode(String absPath)
```
and

```
Property Session.getProperty(String absPath).
```

5.1.4 Getting a Node by Identifier

A node can be retrieved by its identifier with

```
Node Session.getNodeByIdentifier(String identifier).
```

Using an identifier-based absolute path a node can also be retrieved by identifier with a path-base `get` method. For example,

```
S.getNode("[" + id + "]")
```

where `S` is the session and `id` is the identifier.

5.1.5 Testing for Existence by Relative Path

Existence of nodes and properties can be tested by path relative to a given node with

```
boolean Node.hasNode(String relPath)
```
and

```
boolean Node.hasProperty(String relPath).
```

5.1.6 Access by Relative Path

Nodes and properties can be acquired via relative path with

```
Node Node.getNode(String relPath)
```
and

```
Property Node.getProperty(String relPath)
```

5.1.7 Primary Item Access

If a primary child item is specified by the node type of a node, this item can be retrieved directly from the node with

```
Item Node.getPrimaryItem().
```

See §3.7.1.7 *Primary Item*.

5.1.8 Node and Property with Same Name

In some repositories a node and property with the same parent may have the same name. The methods `Node.getNode`, `Session.getNode`, `Node.getProperty` and `Session.getProperty` specify whether the desired item is a node or a property. The method `Session.getItem` will return the item at the specified path if there is only one such item, if there is both a node and a property at the specified path, `getItem` will return the node.

Whether an implementation supports this feature can be determined by querying the repository descriptor table with

```
Repository.OPTION_NODE_AND_PROPERTY_WITH_SAME_NAME_SUPPORTED.
```

A return value of `true` indicates support (see §24.2 *Repository Descriptors*).

5.2 Traversal Access

Traversal access to an item `I` means that it is returned when iterating over the children of a node.

For any given session `S` and item `I`, the statements below must be either both true or both false. If they are both true then `S` has traversal access to `I`, if they are both false then `S` does not have traversal access to `I`:

- `S` has access to `N` where `N` is the parent of `I` and `I` appears among the items in the iterator returned by either `N.getNodes` or `N.getProperties`.
- `S` has access to `N`, `I` is a descendant of `N` and I appears in the serialized output of an export of the subgraph rooted at `N`.

5.2.1 Testing Existence

A client can test whether a retrieved iterator will be empty using the following:

```
boolean Node.hasNodes()

boolean Node.hasProperties()
```

5.2.2 Iterating Over Child Items

Iterators over child nodes and properties can be acquired using the following methods:

```
NodeIterator Node.getNodes()

PropertyIterator Node.getProperties()
```

These methods return all the child nodes or properties (as the case may be) of the node that are visible to the current session.

```
NodeIterator Node.getNodes(String namePattern)

NodeIterator Node.getNodes(String[] nameGlobs)

PropertyIterator Node.getProperties(String namePattern)

PropertyIterator Node.getProperties(String[] nameGlobs)
```

These methods return all the child nodes or properties (as the case may be) of the node that are both visible to the current session and that match the passed `namePattern` or `nameGlobs` array.

5.2.2.1 Name Patterns

The `namePattern` passed in `Node.getNodes` and `Node.getProperties` is a string matched against the qualified names (not the paths) of the immediate child items of this node. We call the `namePattern` parameter the *pattern* and the qualified names against which it is tested the *target strings*.

- A *pattern* consists of one or more *globs*. In cases of two or more globs, they are delimited by the pipe character (`|`, U+0076).
- A pattern matches a target string if and only if at least one of its globs matches that target string.
- A *glob* matches a target string if and only if it matches character for character, except for any asterisk characters (*, U+002A) in the glob, which match any substring (including the empty string) in the target string.

The characters "`|`" and "`*`" are excluded from qualified JCR names (see §3.2.5.2 *Qualified Form*), so their use as metacharacters in the pattern will not lead to a conflict.

For backwards compatibility with JCR 1.0, leading and trailing whitespace around a glob is ignored but whitespace within a glob forms part of the pattern to be matched.

5.2.2.2 Name Globs

The alternate signatures

```
NodeIterator Node.getNodes(String[] nameGlobs)

PropertyIterator Node.getProperties(String[] nameGlobs)
```

Behave identically to those that take `namePattern` except that the parameter passed is an array of globs, as defined above, which are "ORed" together, removing the need for the "`|`" metacharacter to indicate disjunction. The items

returned, therefore, are those that match *at least one* of the globs in the array. Unlike the `namePattern` case, leading and trailing whitespace in globs *is not* ignored by these methods.

5.2.2.3 Child Node Order Preservation

Depending on the implementation, the order of child nodes within the returned iterator may be more or less stable across different retrievals. A repository that supports *preservation of child node ordering* will maintain a constant total order across separate retrievals. A repository that supports orderable child nodes necessarily also supports order preservation (§23 *Orderable Child Nodes*).

5.2.3 Export

Exporting a subgraph within a workspace can be done with

`Session.exportSystemView` or

`Session.exportDocumentView`.

See §7 *Export*.

5.3 Query Access

A session `S` has query access to `I` if and only if for at least one `Query` object `Q`, where `Q` is created through the `QueryManager` of the `Workspace` object bound to `S`, `I` is returned in the `QueryResult` for `Q` (see §6 *Query*).

5.4 Relationship among Access Modes

For any given session `S` and item `I`:

- If `S` has traversal access to `I` then `S` must have direct access to `I`.

If `S` has query access `I` then `S` has direct access to `I`.

However, note that,

- If S has direct access to `I` then `S` may or may not have traversal access to `I`.
- If S has direct access to `I` then `S` may or may not have query access to `I`.
- If S has direct access to `I` then `S` may or may not have direct access to any parent of `I`.

5.5 Effect of Access Denial on Read

If a repository restricts the read access of a session, then the nodes and properties to which that session does not have read access must appear not to exist. For example, the iterator returned on `N.getNodes` will not include subnodes of `N` to which the session in question does not have read access. In other words, lack of read access to an item blocks access to both information about the content of that item and information about the existence of that item.

In repositories that support *same-name siblings*, denial of access to a subset of nodes within a same-name sibling series *may* result in gaps in the index

numbering of that series, thus revealing information about the existence of the inaccessible nodes.

5.6 Item Information

The `Item` interface includes a number of methods that provide information about an item.

5.6.1 Item to Session

This method provides access to the current `Session`.

```
Session Item.getSession()
```

5.6.2 Item in Hierarchy

These methods provide information about the location of an `Item` within the workspace hierarchy:

```
String Item.getName()
```

returns the name of the `Item`.

```
String Item.getPath()
```

returns the absolute path of the `Item`.

```
Node Item.getAncestor(int depth)
```

returns the ancestor of the `Item` that is at the specified depth below the root node.

```
Node Item.getParent()
```

returns the parent of the `Item`.

```
int Item.getDepth()
```

returns the depth below the root node of the `Item`.

5.6.3 Item Subclass

```
boolean Item.isNode()
```

returns `true` if the `Item` is a `Node` and `false` if it is a `Property`.

5.6.4 Item Comparison

This method is used to determine the repository-level semantic identity of two `Item` objects.

```
boolean Item.isSame(Item otherItem)
```

returns true if this `Item` object represents the same actual repository item as the object `otherItem`. This method does not compare the states of the two items. For example, if two `Item` objects representing the same actual repository item have been retrieved through two different sessions and one has been modified, then

this method will still return `true` for these two objects. Note that if two `Item` objects representing the same repository item are retrieved through the same `Session` they will always reflect the same state so comparing state is not an issue (see section §10.11.7 *Reflecting Item State*).

5.6.5 Item Visitor

This method implements the *visitor design pattern*.

```
void Item.accept(ItemVisitor visitor)
```

The `ItemVisitor` interface defines the methods

```
void ItemVisitor.visit(Node node)
```

and

```
void ItemVisitor.visit(Property property)
```

which the user can implement.

5.7 Node Identifier

The method

```
String Node.getIdentifier()
```

returns the identifier of a node.

5.8 Node Index

The method

```
int Node.getIndex()
```

returns the index of a node among its same-name siblings (see §22 *Same-Name Siblings*). Same-name sibling indexes begin with `[1]`, so this method will return `1` for a node without any same-name siblings.

5.9 Iterators

Methods that return a set of `Node` or `Property` objects do so using a `NodeIterator` or `PropertyIterator`, subclasses of `RangeIterator`.

JCR also specifies the following subclasses of `RangeIterator`: `RowIterator`, `NodeTypeIterator`, `VersionIterator`, `EventListenerIterator`, `AccessControlPolicyIterator`, `EventIterator` and `EventJournal`.

5.9.1 Iterator Lifespan

The lifespan of an instance of `RangeIterator` or any of its subclasses is implementation-specific. For example, in some implementations a `Session.refresh` (see §10.11.1 *Refresh*) might invalidate a previously acquired `NodeIterator` while in others it might not.

5.10 Reading Properties

If a session has read access to a single–value property then it can read the value of that property. If a session has read access to a multi-value property then it can read *all* the values of that property.

5.10.1 Getting a Value

The generic value getter for single value properties is

```
Value Property.getValue().
```

For multi-value properties it is

```
Value[] Property.getValues().
```

Single and multi-value properties can be distinguished by calling

```
boolean Property.isMultiple().
```

5.10.2 Value Type

```
int Value.getType()
```

returns one of the constants of `PropertyType` (see §3.6.1 *Property Types*) indicating the property type of the `Value`.

5.10.3 Value Length

The length of a value in a single-value property, as defined in §3.6.7 *Length of a Value*, is returned by

```
long Property.getLength()
```

Similarly, the method

```
long[] Property.getLengths()
```

is used to get an array of the lengths of all the values of a multi-value property.

5.10.4 Standard Value Read Methods

Each property type has a standard `Value` read method. This is the method that returns the Java object or primitive type that corresponds naturally to the JCR property type. A `Value` may also be readable by a non-standard read method, depending on whether it is convertible to that method's return type according to the rules described in §3.6.4 *Property Type Conversion*. The following sections set out the standard read method for each type.

5.10.4.1 STRING

```
String Value.getString()
```

returns a JCR `STRING` as a `java.lang.String`.

5.10.4.2 BINARY

```
Binary Value.getBinary()
```

returns a JCR `BINARY` as a `javax.jcr.Binary` (see §5.10.5 *Binary Object*).

5.10.4.3 LONG

```
long Value.getLong()
```

returns a JCR `LONG` as a Java `long`.

5.10.4.4 DOUBLE

```
double Value.getDouble()
```

returns a JCR `DOUBLE` as a Java `double`.

5.10.4.5 DECIMAL

```
BigDecimal Value.getDecimal()
```

returns a JCR `DECIMAL` as a `java.math.BigDecimal`.

5.10.4.6 DATE

```
Calendar Value.getDate()
```

returns a JCR `DATE` as a `java.util.Calendar`.

5.10.4.7 BOOLEAN

```
boolean Value.getBoolean()
```

returns a JCR `BOOLEAN` as a Java `boolean`.

5.10.4.8 NAME

```
String Value.getString()
```

returns a JCR `NAME` as a `String`. The `String` returned must be the JCR name in *qualified form* (see §3.2.5.2 *Qualified Form*).

5.10.4.9 PATH

```
String Value.getString()
```

returns a JCR `PATH` as a `String`. The `String` returned must be the JCR path in *standard form* (see §3.4.3.1 *Standard Form*). However, if the original value was *non-normalized* it must be returned non-normalized, preserving the path structure as it was originally set, including any redundant path segments that may exist (see §3.4.5 *Normalized Paths*).

5.10.4.10 REFERENCE and WEAKREFERENCE

```
String Value.getString()
```

returns a JCR `REFERENCE` or `WEAKREFERENCE` as a `String`. The value of a `REFERENCE` or `WEAKREFERENCE` is a node referenceable identifier (see §3.8.3 *Referenceable Identifiers*). Since an identifier is simply a `String`, the returned value can be used directly to find the referenced node (see §5.1.4 *Getting a Node by Identifier*).

5.10.5 Binary Object

The `Binary` object returned by `Value.getBinary()` provides the following methods:

```
InputStream Binary.getStream(),
```

which returns an `InputStream` representation of the value. Each call to this method returns a new stream and the API consumer is responsible for calling `close()` on the returned stream.

```
int Binary.read(byte[] b, long position),
```

which reads successive bytes starting from the specified position in the value into the passed byte array until either the byte array is full or the end of the value is encountered.

```
long Binary.getSize(),
```

which returns the size of the value in bytes.

5.10.5.1 Disposing of a Binary Object

When an application is finished with a Binary object it should call

```
void Binary.dispose()
```

on that object. This will releases all resources associated with the object and inform the repository that these resources may now be reclaimed.

5.10.5.2 Deprecated Binary Behavior

The `Binary` interface and its related methods in `Property`, `Value` and `ValueFactory` replace the deprecated `Value.getStream()` and `Property.getStream()` methods from JCR 1.0. Though these methods have been deprecated, for reasons of backward compatibility their behavior must conform to the following rules:

- Once a `Value` object has been read once using `getStream()`, all subsequent calls to `getStream()` will return the same stream object. This may mean, for example, that the stream returned is fully or partially consumed. In order to get a fresh stream the `Value` object must be reacquired via `Property.getValue()` or `Property.getValues()`.
- Unlike in JCR 1.0, calling a `get` method other than getStream before calling `getStream` on the same `Value` object will never cause an `IllegalStateException`.

5.10.6 Dereferencing

PATH, WEAKREFERENCE and REFERENCE properties function as pointers to other items in the workspace. A PATH can point to a node or a property while a WEAKREFERENCE or REFERENCE can point only to a referenceable node. REFERENCE properties enforce referential integrity while WEAKREFERENCE properties and PATH properties do not. These properties can be dereferenced either manually or though convenience methods.

5.10.6.1 Manual Dereference

To manually dereference a pointer property it is first read as a string, for example with

```
Value.getString().
```

In the case of WEAKREFERENCE and REFERENCE properties the resulting string is passed to

```
Session.getNodeByIdentifier(String id).
```

In the case of PATH properties the string is passed to

```
Session.getNode(String absPath)
```

or

```
Session.getProperty(String absPath)
```

as appropriate to the target item. Whether the Item is a Node or Property can be determined with Session.nodeExists or Session.propertyExists (see §5.1.2 *Testing for Existence by Absolute Path*).

5.10.6.2 Dereferencing Convenience Methods

The Property interface provides convenience methods for dereferencing pointer properties:

```
Node Property.getNode()
```

returns the node pointed to by a single-value property. This method works with WEAKREFERENCE and REFERENCE properties and with PATH properties that point to nodes.

```
Property Property.getProperty()
```

returns the property pointed to by a single-value PATH property.

For multi-value pointer properties the array of values must be retrieved with Property.getValues and each individually manually dereferenced.

5.10.7 Backtracking References

Given a referenceable node,

```
Node.getReferences()
```

returns all accessible REFERENCE properties in the workspace that point to the node.

```
Node.getWeakReferences()
```

returns all accessible `WEAKREFERENCE` properties in the workspace that point to the node.

Note that access control and other implementation-specific limitations my mean that some references within the workspace are not accessible.

`PATH` properties are not automatically backtrackable.

5.10.8 Single-Value Property Read Methods

The property interface provides convenience methods for reading single-value properties which function identically to their `Value` counterparts.

5.10.9 Reading Multi-Value Properties

A multi-value property can be accessed with

```
Value[] Property.getValues().
```

5.10.10 PropertyType Class

The class `PropertyType` defines integer constants for the property types as well as string constants for their standardized type names (which are used in serialization) and two methods for converting back and forth between name and integer value (see Javadoc).

5.11 Namespace Mapping

The method

```
void Session.setNamespacePrefix(String prefix,
                                String uri)
```

is used to change the local namespace mappings of the current `Session`. When called, all local mappings that include either the specified `prefix` or the specified `uri` are removed and the new mapping is added. However, the method will throw an exception if

- the specified prefix begins with the characters "`xml`" (in any combination of case) or,
- the specified prefix is the empty string or,
- the specified namespace URI is the empty string.

The following methods are also related to the local namespace mapping:

```
String[] Session.getNamespacePrefixes()

String Session.getNamespaceURI(String prefix)

String Session.getNamespacePrefix(String uri)
```

6 Query

A repository may support *query*.

The structure and evaluation semantics of a query are defined by an *abstract query model* (AQM) for which two concrete language bindings are specified:

- *JCR-SQL2*, which expresses a query as a string with syntax similar to SQL, and
- *JCR-JQOM* (JCR Java Query Object Model), which expresses a query as a tree of Java objects.

The languages are both direct mappings of the AQM and are therefore equally expressive; any query expressed in one can be machine-transformed to the other.

Whether an implementation supports query can be determined by querying the repository descriptor table with the key

```
Repository.QUERY_LANGUAGES.
```

The returned array contains the constants representing the supported languages (see §24.2 *Repository Descriptors*). If a repository supports query it must return at least the constants for the two JCR-defined languages,

`javax.jcr.query.JCR-JQOM` and

`javax.jcr.query.JCR-SQL2`,

indicating support for those languages. In addition, a repository may support other query languages. These can be either additional language bindings to the AQM or completely independent of that model.

JCR 1.0 defines a dialect of SQL different from JCR-SQL2, as well as a dialect of XPath. Support for these languages is deprecated.

6.1 Optional Joins

Support for *joins* is optional beyond support for query itself. The extent of join support can be determined by querying the repository descriptor table with the key

```
Repository.QUERY_JOINS.
```

The value returned will be one of

- `QUERY_JOINS_NONE`: Joins are not supported and therefore queries are limited to a single selector.
- `QUERY_JOINS_INNER`: Inner joins are supported.
- `QUERY_JOINS_INNER_OUTER`: Inner and outer joins are supported.

6.2 Introduction to the Abstract Query Model

This section introduces how queries are specified and evaluated in the AQM.

6.2.1 Selectors

A query has one or more *selectors*. When the query is evaluated, each selector independently selects a subset of the nodes in the workspace based on node type.

In a repository that *does not* support *joins*, a query will have only one selector.

6.2.2 Joins

If the query has more than one selector, it also has one or more *joins* that transform the sets of nodes selected by each selector into a single set of *node-tuples*.

The membership of the set of node-tuples depends on the *join type* and *join condition* of each join. The join type can be *inner*, *left-outer*, or *right-outer*. The join condition can test the equality of properties' values or the hierarchical relationship between nodes.

If the query has *n* selectors, it has *n - 1* joins resulting in a set of *n*-tuples. For example, if the query has two selectors, it will have one join and produce a set of 2-tuples. If it has three selectors, it will have two joins and produce a set of 3-tuples. If it has only one selector, it will not have any joins and will produce a set of 1-tuples, that is, the nodes selected by its only selector.

Support for *joins* is optional. In a repository that *does not* support *joins*, the node-tuples produced are necessarily singletons. In other words, each node in the set produced by the (one and only) selector is converted directly into a node-tuple of size one. All further processing within the query evaluation operates on these tuples just as it would on tuples of size greater than one.

6.2.3 Constraints

A query can specify a *constraint* to filter the set of node-tuples by any combination of:

- Absolute or relative path, for example:
    - The node reached by path `/pictures/sunset.jpg`
    - Nodes that are children of `/pictures`
    - Nodes that are descendants of `/pictures`
- Name of the node, for example:
    - Nodes named `sunset.jpg`
- Value of a property, for example:
    - Nodes whose `jcr:created` property is after `2007-03-14T00:00:00.000Z`

- Length of a property, for example:
    - Nodes whose `jcr:data` property is longer than 100 KB
- Existence of a property, for example:
    - Nodes with a `jcr:language` property
- Full-text search, for example:
    - Nodes which have a property that contains the phrase "beautiful sunset"

6.2.4 Orderings

A query can specify *orderings* to sort the filtered node-tuples by property value.

6.2.5 Query Results

The filtered and sorted node-tuples form the *query results*. The query results are available in two formats:

- A list of node-tuples. For each node-tuple, you can retrieve the node for each selector. In a repository that does not support *joins* there will be only one selector and consequently only one node per tuple.
- A table whose rows are the node-tuples and whose columns are properties of the nodes in the node-tuples. This is referred to as the tabular view of the query results. A query can specify which properties appear as columns in the tabular view.

6.3 Equality and Comparison

When testing for equality or order of two property values of the same type, the query operators conform to the definitions in §3.6.5 *Comparison of Values*.

When testing for equality or order of two property values of differing type, the query operators perform standard property type conversion (see §3.6.4 *Property Type Conversion*) and conform to standard value comparison (see §3.6.5 *Comparison of Values*).

Support for equality and order comparison of `BINARY` values is not required.

6.4 Query Validity

To be successfully evaluated and produce query results, a query must be *valid*.

A query is *invalid* if:

- it cannot be expressed in the AQM, or
- it can be expressed in the AQM, but fails a validation constraint defined in §6.7 *Abstract Query Model and Language Bindings*.

An invalid query causes the repository to throw `InvalidQueryException`. Which method invocation throws this exception is implementation determined, but for

an invalid query, the exception must be thrown no later than completion of the `Query.execute()`.

6.5 Search Scope

A query *must* search the persistent workspace associated with the current session. It *may* take into account pending changes to the persistent workspace; that is, changes which are either unsaved or, within a transaction, saved but uncommitted.

6.6 Notations

Three notations are used in the following sections: the AQM type grammar, the JCR-SQL2 EBNF grammar and the JCR-JQOM Java API.

6.6.1 AQM Notation

The AQM is defined as a set of abstract types. The type grammar is written like this:

```
type Alpha ::=
      Foo foo,
      Bar? bar,
      Baz+ bazes,
      Quux* quuxes

type Beta extends Alpha ::=
      String name

enum Foo ::=
      Snap,
      Crackle,
      Pop
```

which means:

The type `Alpha` has 4 attributes:

`foo`: mandatory, of type `Foo`, which is an enumeration with possible values `Snap`, `Crackle` and `Pop`.

`bar`: optional, of type `Bar`

`bazes`: a list of one or more `Baz` items

`quuxes`: a list of zero or more `Quux` items

The type `Beta` is a subtype of `Alpha`. It inherits `Alpha`'s attributes, and adds:

`name`: mandatory, a string

6.6.2 JCR-SQL2 Notation

JCR–SQL2 is a mapping of the AQM to a string serialization based on the SQL language.

Each non-terminal in the JCR-SQL2 EBNF grammar corresponds to the type of the same name in the AQM grammar. The semantics of each JCR-SQL2 production is

described by reference to the semantics of the corresponding AQM production. The two grammars are, however, entirely distinct and self- contained. Care should be taken not to mix productions from one grammar with those of the other.

The JCR-SQL2 grammar is written like this:

```
Alpha ::= 'FOO' Foo ['BAR' Bar] 'BAZ' bazes
          ['QUUX' quuxes]

Foo ::= Snap | Crackle | Pop

Snap ::= 'SNAP'

Crackle ::= 'CRACKLE'

Pop ::= 'POP'

Bar ::= /* a Bar */

bazes ::= Baz {Baz}

Baz ::= /* a Baz */

quuxes ::= Quux {Quux}

Quux ::= /* a Quux */
```

6.6.2.1 String Literals in JCR-SQL2 Grammar

Throughout this section string literals that appear in the syntactic grammar defining JCR-SQL2 must be interpreted as specified in §1.3.1 *String Literals in Syntactic Grammars* except that each character in the string literal must be interpreted as representing both upper and lower case versions. In other words, implementations must be case-insensitive with regard to JCR-SQL2.

6.6.3 JCR-JQOM Notation

JCR-JQOM is a mapping of the AQM to a Java API.

Each method and parameter name of the JCR-JQOM Java API corresponds to the type of the same name in the AQM grammar. The semantics of each JCR-JQOM method is described by reference to the semantics of the corresponding AQM production.

A JCR-JQOM query is built by assembling objects created using the factory methods of `QueryObjectModelFactory`.

For each AQM type, the following are listed:

- If the AQM type is a *non-enum* and *non-abstract* (in the AQM sense, not the Java sense) then the factory method of `QueryObjectModelFactory` used to create an instance of that type is listed.
- If the AQM type is *non-enum* then the corresponding Java interface is listed.

- If the AQM type is an *enum* then the corresponding constants of `QueryObjectModelConstants` are listed.

Unless otherwise indicated, the Java interfaces listed in this section are in the package `javax.jcr.query.qom`.

6.7 Abstract Query Model and Language Bindings

The following section describes the AQM grammar and its mapping to JCR-SQL2 and JCR-JQOM. For each AQM production, a description of its semantics is provided, followed by the corresponding JCR-SQL2 production and the corresponding JCR-JQOM methods.

For queries with only one selector the JCR-SQL2 syntax permits the selector name to be omitted. In such cases the implementation must automatically generate a selector name for internal use. If the resulting query is later examined through the JCR-JQOM API, the automatically produced selector name will be seen.

6.7.1 Query

AQM

```
type Query ::=
      Source source,
      Constraint? constraint,
      Ordering* orderings,
      Column* columns
```

A `Query` consists of:

- A `Source`. When the query is evaluated, the `Source` evaluates its selectors and the joins between them to produce a (possibly empty) set of node-tuples. This is a set of 1-tuples if the query has one selector (and therefore no joins), a set of 2-tuples if the query has two selectors (and therefore one join), a set of 3-tuples if the query has three selectors (two joins), and so forth.
- An optional `Constraint`. When the query is evaluated, the constraint filters the set of node-tuples.
- A list of zero or more `Ordering`s. The orderings specify the order in which the node-tuples appear in the query results. The relative order of two node-tuples is determined by evaluating the specified orderings, in list order, until encountering an ordering for which one node-tuple precedes the other. If no orderings are specified, or if there is no ordering specified in which one node-tuple precedes the other, then the relative order of the node-tuples is implementation determined (and may be arbitrary).
- A list of zero or more `Column`s to include in the tabular view of the query results. If no columns are specified, the columns available in the tabular view are implementation determined, but minimally include, for each selector, a column for each single-valued non-residual property of the selector's node type.

JCR-SQL2

```
Query ::= 'SELECT' columns
          'FROM' Source
          ['WHERE' Constraint]
          ['ORDER BY' orderings]
```

JCR-JQOM

A query is represented by a `QueryObjectModel` object, created with:

```
QueryObjectModel QueryObjectModelFactory.
    createQuery(Source source,
                Constraint constraint,
                Ordering[] orderings,
                Column[] columns)
```

`QueryObjectModel` extends `javax.jcr.query.Query` and declares:

```
Source QueryObjectModel.getSource()

Constraint QueryObjectModel.getConstraint()

Ordering[] QueryObjectModel.getOrderings()

Column[] QueryObjectModel.getColumns()
```

6.7.2 Source

AQM

```
abstract type Source
```

Evaluates to a set of node-tuples.

JCR-SQL2

```
Source ::= Selector | Join
```

JCR-JQOM

`Source` is an empty interface with subclasses `Selector` and `Join`.

6.7.3 Selector

AQM

```
type Selector extends Source ::=
      Name nodeType,
      Name selectorName
```

Selects a subset of the nodes in the workspace based on node type.

The query is invalid if `nodeType` refers to a node type that has a *queryable node type* attribute of `false` (see §3.7.1.5 *Queryable Node Type*). Otherwise, if the *queryable node type* attribute is true, the following holds:

A selector selects every node in the workspace, subject to access control constraints, that satisfies at least one of the following conditions:

- the node's primary node type is `nodeType`, or
- the node's primary node type is a subtype of `nodeType`, or
- the node has a mixin node type that is `nodeType`, or
- the node has a mixin node type that is a subtype of `nodeType`.

A selector has a `selectorName` that can be used elsewhere in the query to identify the selector.

The query is *invalid* if `selectorName` is identical to the `selectorName` of another selector in the query.

The query is also *invalid* if `nodeType` is not a valid JCR name or is a valid JCR name but not the name of a node type available in the repository.

JCR-SQL2

```
Selector ::= nodeTypeName ['AS' selectorName]

nodeTypeName ::= Name
```

JCR-JQOM

A `Selector` is created with:

```
Selector QueryObjectModelFactory.
    selector(String nodeTypeName, String selectorName)
```

`Selector` extends `Source` and declares:

```
String Selector.getNodeTypeName()

String Selector.getSelectorName()
```

6.7.4 Name

AQM

```
type Name
```

A JCR name.

The query is *invalid* if the name does not satisfy either the `ExpandedName` production in §3.2.5.1 *Expanded Form* or the `QualifiedName` production in §3.2.5.2 *Qualified Form*.

JCR-SQL2

```
Name ::= '[' quotedName ']' |
         '[' simpleName ']' |
         simpleName

quotedName ::= /* A JCR Name */
```

```
simpleName ::= /* A JCR Name that is also a legal SQL identifier[10] */
```

JCR-JQOM

A JCR name in `String` form (either qualified or expanded).

6.7.5 Join

Support for *joins* is optional.

AQM

```
type Join extends Source ::=
      Source left,
      Source right,
      JoinType joinType,
      JoinCondition joinCondition
```

Performs a join between two node-tuple sources.

If `left` evaluates to **L**, a set of *m*-tuples, and `right` evaluates to **R**, a set of *n*-tuples, then the join evaluates to **J**, a set of (*m* + *n*)-tuples. The members of **J** depend on the `joinType` and `joinCondition`.

Let **L** x **R** be the Cartesian product of **L** and **R** as a set of (*m* + *n*)-tuples

$$\mathbf{L} \times \mathbf{R} = \{ \ell \cup r : \ell \in \mathbf{L}, r \in \mathbf{R} \}$$

and $\sigma_c(A)$ be the selection over **A** of its members satisfying `joinCondition` φ_c

$$\sigma_c(\mathbf{A}) = \{ a : a \in \mathbf{A}, \varphi_c(a) \}$$

Then if `joinType` is `Inner`:

$$\mathbf{J} = \sigma_c(\mathbf{L} \times \mathbf{R})$$

Otherwise, if `joinType` is `LeftOuter`:

$$\mathbf{J} = \sigma_c(\mathbf{L} \times \mathbf{R}) \cup (\mathbf{L} - \pi_L(\sigma_c(\mathbf{L} \times \mathbf{R})))$$

where $\pi_L(\sigma_c(\mathbf{L} \times \mathbf{R}))$ is the projection of the *m*-tuples contributed by **L** from the *(m + n)*-tuples of $\sigma_c(\mathbf{L} \times \mathbf{R})$.

Otherwise, if `joinType` is `RightOuter`:

$$\mathbf{J} = \sigma_c(\mathbf{L} \times \mathbf{R}) \cup (\mathbf{R} - \pi_R(\sigma_c(\mathbf{L} \times \mathbf{R})))$$

---

[10] See the SQL:92 rules for `<regular identifier>` (in ISO/IEC 9075:1992 §5.2 `<token>` and `<separator>`).

where $\pi_R(\sigma_c(\mathbf{L} \times \mathbf{R}))$ is the projection of the *n*-tuples contributed by **R** from the *(m + n)*-tuples of $\sigma_c(\mathbf{L} \times \mathbf{R})$.

The query is *invalid* if `left` is the same source as `right`.

JCR-SQL2

```
Join ::= left [JoinType] 'JOIN' right 'ON' JoinCondition
         // If JoinType is omitted INNER is assumed.

left ::= Source

right ::= Source
```

JCR-JQOM

A `Join` is created with:

```
Join QueryObjectModelFactory.
    join(Source left,
         Source right,
         String joinType,
         JoinCondition joinCondition)
```

`Join` extends `Source` and declares:

```
Source Join.getLeft()

Source Join.getRight()

String Join.getJoinType()

JoinCondition Join.getJoinCondition()
```

6.7.6 JoinType

Support for *joins* is optional.

AQM

```
enum JoinType ::=
      Inner,
      LeftOuter,
      RightOuter
```

JCR-SQL2

```
JoinType ::= Inner | LeftOuter | RightOuter

Inner ::= 'INNER'

LeftOuter ::= 'LEFT OUTER'

RightOuter ::= 'RIGHT OUTER'
```

JCR-JQOM

A join type is a `String` constant. One of:

`QueryObjectModelConstants.JCR_JOIN_TYPE_INNER`

`QueryObjectModelConstants.JCR_JOIN_TYPE_LEFT_OUTER`

`QueryObjectModelConstants.JCR_JOIN_TYPE_RIGHT_OUTER`

6.7.7 JoinCondition

Support for *joins* is optional.

AQM

```
abstract type JoinCondition
```

Filters the set of node-tuples formed from a join.

JCR-SQL2

```
JoinCondition ::= EquiJoinCondition |
                  SameNodeJoinCondition |
                  ChildNodeJoinCondition |
                  DescendantNodeJoinCondition
```

JCR-JQOM

`JoinCondition` is an empty interface with subclasses `EquiJoinCondition`, `SameNodeJoinCondition`, `ChildNodeJoinCondition` and `DescendantNodeJoinCondition`.

6.7.8 EquiJoinCondition

Support for *joins* is optional.

AQM

```
type EquiJoinCondition extends JoinCondition ::=
      Name selector1Name,
      Name property1Name,
      Name selector2Name,
      Name property2Name
```

Tests whether the value of a property in a first selector is equal to the value of a property in a second selector.

A node-tuple satisfies the constraint only if:

- the `selector1Name` node has a property named `property1Name`, and
- the `selector2Name` node has a property named `property2Name`, and
- the value of property `property1Name` *is equal to* the value of property `property2Name`, as defined in §3.6.5 *Comparison of Values*.

The query is *invalid* if

- either `selector1Name` or `selector2Name` is not the name of a selector in the query, or
- `selector1Name` is equal to `selector2Name`, or
- the `property1Name` is not the same property type as `property2Name`, or
- either `property1Name` or `property2Name` is a multi-valued property, or
- either `property1Name` or `property2Name` is a `BINARY` property and equality test for `BINARY` properties is not supported (see §3.6.6 *Value.equals Method*).

JCR-SQL2

```
EquiJoinCondition ::= selector1Name'.'property1Name '='
                      selector2Name'.'property2Name

selector1Name ::= selectorName

selector2Name ::= selectorName

property1Name ::= propertyName

property2Name ::= propertyName
```

JCR-JQOM

An `EquiJoinCondition` is created with:

```
EquiJoinCondition QueryObjectModelFactory.
    equiJoinCondition(String selector1Name,
                      String property1Name,
                      String selector2Name,
                      String property2Name)
```

`EquiJoinCondition` extends `JoinCondition` and declares:

```
String EquiJoinCondition getSelector1Name()

String EquiJoinCondition getProperty1Name()

String EquiJoinCondition getSelector2Name()

String EquiJoinCondition getProperty2Name()
```

6.7.9 SameNodeJoinCondition

Support for *joins* is optional.

AQM

```
type SameNodeJoinCondition extends JoinCondition ::=
      Name selector1Name,
      Name selector2Name,
      Path? selector2Path
```

Tests whether two nodes are "the same" according to the `Item.isSame` method.

If `selector2Path` is omitted:

- Tests whether the `selector1Name` node is the same as the `selector2Name` node. A node-tuple satisfies the constraint only if:

  ```
  selector1Node.isSame(selector2Node)
  ```

  would return `true`, where `selector1Node` is the node for the selector `selector1Name` and `selector2Node` is the node for the selector `selector2Name`.

Otherwise, if `selector2Path` is specified:

- Tests whether the `selector1Name` node is the same as a node identified by relative path `selector2Path` from the `selector2Name` node. A node-tuple satisfies the constraint only if:

  ```
  selector1Node.isSame(
      selector2Node.getNode(selector2Path))
  ```

  would return `true`, where `selector1Node` is the node for the selector `selector1Name` and `selector2Node` is the node for the selector `selector2Name`.

The query is *invalid* if:

- `selector1Name` is not the name of a selector in the query, or
- `selector2Name` is not the name of a selector in the query, or
- `selector1Name` is the same as `selector2Name`, or
- `selector2Path` is not a syntactically valid relative path, as defined in §3.4.3.3 *Lexical Path Grammar*. However, if `selector2Path` is syntactically valid but does not identify a node in the workspace visible to this session, the query is valid but the constraint is not satisfied.

JCR-SQL2

```
SameNodeJoinCondition ::=
        'ISSAMENODE(' selector1Name ','
                      selector2Name
                      [',' selector2Path] ')'

selector2Path ::= Path
```

JCR-JQOM

A `SameNodeJoinCondition` is created with:

```
SameNodeJoinCondition QueryObjectModelFactory.
    sameNodeJoinCondition(String selector1Name,
                          String selector2Name,
                          String selector2Path)
```

`SameNodeJoinCondition` extends `JoinCondition` and declares:

```
String SameNodeJoinCondition.getSelector1Name()

String SameNodeJoinCondition.getSelector2Name()

String SameNodeJoinCondition.getSelector2Path()
```

6.7.10 ChildNodeJoinCondition

Support for *joins* is optional.

AQM

```
type ChildNodeJoinCondition extends JoinCondition ::=
      Name childSelectorName,
      Name parentSelectorName
```

Tests whether the `childSelectorName` node is a child of the `parentSelectorName` node. A node-tuple satisfies the constraint only if:

```
childSelectorNode.getParent().isSame(parentSelectorNode)
```

would return `true`, where `childSelectorNode` is the node for the selector `childSelectorName` and `parentSelectorNode` is the node for the selector `parentSelectorName`.

The query is *invalid* if:

- `childSelectorName` is not the name of a selector in the query, or
- `parentSelectorName` is not the name of a selector in the query, or
- `childSelectorName` is the same as `parentSelectorName`.

JCR-SQL2

```
ChildNodeJoinCondition ::=
        'ISCHILDNODE(' childSelectorName ','
                       parentSelectorName ')'

childSelectorName ::= selectorName

parentSelectorName ::= selectorName
```

JCR-JQOM

A `ChildNodeJoinCondition` is created with:

```
ChildNodeJoinCondition QueryObjectModelFactory.
    childNodeJoinCondition(String childSelectorName,
                           String parentSelectorName)
```

`ChildNodeJoinCondition` extends `JoinCondition` and declares:

```
String ChildNodeJoinCondition.getChildSelectorName()

String ChildNodeJoinCondition.getParentSelectorName()
```

6.7.11 DescendantNodeJoinCondition

Support for *joins* is optional.

AQM

```
type DescendantNodeJoinCondition
            extends JoinCondition ::=
      Name descendantSelectorName,
      Name ancestorSelectorName
```

Tests whether the `descendantSelectorName` node is a descendant of the `ancestorSelectorName` node. A node-tuple satisfies the constraint only if:

```
descendantSelectorNode.getAncestor(n).
    isSame(ancestorSelectorNode) &&
  descendantSelectorNode.getDepth() > n
```

would return `true` for some non-negative integer `n`, where `descendantSelectorNode` is the node for the selector `descendantSelectorName` and `ancestorSelectorNode` is the node for the selector `ancestorSelectorName`.

The query is *invalid* if:

- `descendantSelectorName` is not the name of a selector in the query, or
- `ancestorSelectorName` is not the name of a selector in the query, or
- `descendantSelectorName` is the same as `ancestorSelectorName`.

JCR-SQL2

```
DescendantNodeJoinCondition ::=
        'ISDESCENDANTNODE(' descendantSelectorName ','
                            ancestorSelectorName ')'

descendantSelectorName ::= selectorName

ancestorSelectorName ::= selectorName
```

JCR-JQOM

A `DescendantNodeJoinCondition` is created with:

```
DescendantNodeJoinCondition QueryObjectModelFactory.
    descendantNodeJoinCondition(String descendantSelectorName,
                                String ancestorSelectorName)
```

`DescendantNodeJoinCondition` extends `JoinCondition` and declares:

```
String DescendantNodeJoinCondition.getDescendantSelectorName()

String DescendantNodeJoinCondition.getAncestorSelectorName()
```

6.7.12 Constraint

AQM

```
abstract type Constraint
```

Filters the set of node-tuples formed by evaluating the query's selectors and the joins between them.

To be included in the query results, a node-tuple must satisfy the constraint.

JCR-SQL2

```
Constraint ::= And | Or | Not | Comparison |
               PropertyExistence | FullTextSearch |
               SameNode | ChildNode | DescendantNode
```

In JCR-SQL2, the following precedence classes apply, in order of evaluation:

Class	Constraint Production	JCR-SQL2 Syntax
1		`()` *(grouping with parentheses)*

2	`Comparison`
	`PropertyExistence`
	`FullTextSearch`
	`SameNode`
	`ChildNode`
	`DescendantNode`
		`=` , `<>`, `<`, `<=`, `>`, `>=`, `LIKE`
		`IS NOT NULL`
		`CONTAINS()`
		`ISSAMENODE()`
		`ISCHILDNODE()`
		`ISDESCENDANTNODE()`

3	`Not`	`NOT`
4	`And`	`AND`
5	`Or`	`OR`

JCR-JQOM

`Constraint` is an empty interface with subclasses `And`, `Or`, `Not`, `Comparison`, `PropertyExistence`, `FullTextSearch`, `SameNode`, `ChildNode` and `DescendantNode`.

6.7.13 And

AQM

```
type And extends Constraint ::=
      Constraint constraint1,
      Constraint constraint2
```

Performs a logical conjunction of two other constraints.

To satisfy the `And` constraint, a node-tuple must satisfy both `constraint1` and `constraint2`.

JCR-SQL2

```
And ::= constraint1 'AND' constraint2

constraint1 ::= Constraint

constraint2 ::= Constraint
```

JCR-JQOM

An `And` is created with:

```
And QueryObjectModelFactory.
    and(Constraint constraint1, Constraint constraint2)
```

`And` extends `Constraint` and declares:

```
Constraint And.getConstraint1()

Constraint And.getConstraint2()
```

6.7.14 Or

AQM

```
type Or extends Constraint ::=
      Constraint constraint1,
      Constraint constraint2
```

Performs a logical disjunction of two other constraints.

To satisfy the `Or` constraint, the node-tuple must either:

- satisfy `constraint1` but not `constraint2`, or
- satisfy `constraint2` but not `constraint1`, or
- satisfy both `constraint1` and `constraint2`.

JCR-SQL2

```
Or ::= constraint1 'OR' constraint2
```

JCR-JQOM

An `Or` is created with:

```
Or QueryObjectModelFactory.
    or(Constraint constraint1, Constraint constraint2)
```

`Or` extends `Constraint` and declares:

```
Constraint Or.getConstraint1()

Constraint Or.getConstraint2()
```

6.7.15 Not

AQM

```
type Not extends Constraint ::=
      Constraint constraint
```

Performs a logical negation of another constraint.

To satisfy the `Not` constraint, the node-tuple must *not* satisfy `constraint`.

JCR-SQL2

```
Not ::= 'NOT' Constraint
```

JCR-JQOM

A `Not` is created with:

```
Not QueryObjectModelFactory.
    not(Constraint constraint)
```

`Not` extends `Constraint` and declares:

```
Constraint Not.getConstraint()
```

6.7.16 Comparison

AQM

```
type Comparison extends Constraint ::=
      DynamicOperand operand1,
      Operator operator,
      StaticOperand operand2
```

Filters node-tuples based on the outcome of a binary operation.

For any comparison, `operand2` always evaluates to a scalar value. In contrast, `operand1` may evaluate to an array of values (for example, the values of a multi-valued property), in which case the comparison is separately performed for each element of the array, and the `Comparison` constraint is satisfied as a whole if the comparison against *any* element of the array is satisfied.

If `operand1` and `operand2` evaluate to values of different property types, the value of `operand2` is converted to the property type of the value of `operand1` as described in §3.6.4 *Property Type Conversion*. If the type conversion fails, the query is *invalid*.

Given an operator `O` and a property instance `P` of property type `T`, `P` can be compared using `O` only if:

- The implementation supports comparison of properties of type `T` using `O`. For example, some implementations may permit `EqualTo` and `NotEqualTo` as comparison operators for `BINARY` properties while others may not.

- Assuming that comparison of properties of type `T` is supported in general, the property definition that applies to `P` (found in the node type of `P`'s parent node) must also list `O` among its *available query operators* (see §3.7.3.3 *Available Query Operators*).

If `operator` is not supported for the property type of `operand1`, the query is *invalid*.

If `operand1` evaluates to `null` (for example, if the operand evaluates the value of a property which does not exist), the constraint is not satisfied.

The `EqualTo` operator is satisfied *only if* the value of `operand1` *is equal to* the value of `operand2`, as described in §3.6.5 *Comparison of Values*.

The `NotEqualTo` operator is satisfied *unless* the value of `operand1` *is equal to* the value of `operand2`, as described in §3.6.5 *Comparison of Values*.

The `LessThan` operator is satisfied *only if* the value of `operand1` *is ordered before* the value of `operand2`, as described in §3.6.5 *Comparison of Values*.

The `LessThanOrEqualTo` operator is satisfied *unless* the value of `operand1` *is ordered after* the value of `operand2`, as described in §3.6.5 *Comparison of Values*.

The `GreaterThan` operator is satisfied *only if* the value of `operand1` *is ordered after* the value of `operand2`, as described in §3.6.5 *Comparison of Values*.

The `GreaterThanOrEqualTo` operator is satisfied *unless* the value of `operand1` *is ordered before* the value of `operand2`, as described in §3.6.5 *Comparison of Values*.

The `Like` operator is satisfied *only if* the value of `operand1` *matches* the pattern specified by the value of `operand2`, where in the pattern:

- the character "`%`" matches zero or more characters, and
- the character "`_`" (underscore) matches exactly one character, and
- the string "`\`*x*" matches the character "*x*", and
- all other characters match themselves.

JCR-SQL2

```
Comparison ::= DynamicOperand Operator StaticOperand
```

JCR-JQOM

A `Comparison` is created with:

```
Comparison QueryObjectModelFactory.
    comparison(DynamicOperand operand1,
               String operator,
               StaticOperand operand2)
```

`Comparison` extends `Constraint` and declares:

```
DynamicOperand Comparsion.getOperand1()

String Comparison.getOperator()

StaticOperand Comparison.getOperand2()
```

6.7.17 Operator

AQM

```
enum Operator ::=
      EqualTo,
      NotEqualTo,
      LessThan,
      LessThanOrEqualTo,
      GreaterThan,
      GreaterThanOrEqualTo,
      Like
```

JCR-SQL2

```
Operator ::= EqualTo | NotEqualTo | LessThan |
             LessThanOrEqualTo | GreaterThan |
             GreaterThanOrEqualTo | Like

EqualTo ::= '='

NotEqualTo ::= '<>'

LessThan ::= '<'

LessThanOrEqualTo ::= '<='

GreaterThan ::= '>'

GreaterThanOrEqualTo ::= '>='

Like ::= 'LIKE'
```

JCR-JQOM

An operator is a `String` constant. One of:

```
QueryObjectModelConstants.JCR_OPERATOR_EQUAL_TO

QueryObjectModelConstants.JCR_OPERATOR_GREATER_THAN

QueryObjectModelConstants.JCR_OPERATOR_GREATER_THAN_OR_EQUAL_TO

QueryObjectModelConstants.JCR_OPERATOR_LESS_THAN

QueryObjectModelConstants.JCR_OPERATOR_LESS_THAN_OR_EQUAL_TO

QueryObjectModelConstants.JCR_OPERATOR_LIKE

QueryObjectModelConstants.JCR_OPERATOR_NOT_EQUAL_TO
```

6.7.18 PropertyExistence

AQM

```
type PropertyExistence extends Constraint ::=
      Name selectorName,
      Name propertyName
```

Tests the existence of a property.

A node-tuple satisfies the constraint if the `selectorName` node has a property named `propertyName`.

The query is *invalid* if `selectorName` is not the name of a selector in the query.

JCR-SQL2

```
PropertyExistence ::=
        selectorName'.'propertyName 'IS NOT NULL' |
        propertyName 'IS NOT NULL' /* If only one
                                      selector exists in
                                      this query*/

        /* Note: The negation, 'NOT x IS NOT NULL'
           can be written 'x IS NULL' */
```

JCR-JQOM

A `PropertyExistence` is created with:

```
PropertyExistence QueryObjectModelFactory.
    propertyExistence(String selectorName, String propertyName)
```

`PropertyExistence` extends `Constraint` and declares:

```
String PropertyExistence.getSelectorName()

String PropertyExistence.getPropertyName()
```

6.7.19 FullTextSearch

AQM

```
type FullTextSearch extends Constraint ::=
      Name selectorName,
      Name? propertyName,
      StaticOperand fullTextSearchExpression
```

Performs a full-text search.

The full-text search expression is evaluated against the set of full-text indexed properties within the full-text search scope. If `propertyName` is specified, the full-text search scope is the property of that name on the `selectorName` node in the node-tuple; otherwise the full-text search scope is implementation determined.

Whether a particular property is full-text indexed can be determined by the *full-text searchable* attribute of its property definition (see §3.7.3.4 *Full-Text Searchable*).

It is implementation-determined whether `fullTextSearchExpression` is independently evaluated against each full-text indexed property in the full-text search scope, or collectively evaluated against the set of such properties using some implementation-determined mechanism.

Similarly, for multi-valued properties, it is implementation-determined whether `fullTextSearchExpression` is independently evaluated against each element in the array of values, or collectively evaluated against the array of values using some implementation-determined mechanism.

The `fullTextSearchExpression` is a `StaticOperand`, meaning that it may be either a literal JCR value or a bound variable (which evaluates to a JCR value). The value must be a `STRING` (or convertible to a `STRING`) that conforms to the following grammar:

```
FullTextSearchLiteral ::= Disjunct
                          {Space 'OR' Space Disjunct}

Disjunct ::= Term {Space Term}

Term ::= ['-'] SimpleTerm

SimpleTerm ::= Word | '"' Word {Space Word} '"'

Word ::= NonSpaceChar {NonSpaceChar}

Space ::= SpaceChar {SpaceChar}

NonSpaceChar ::= Char - SpaceChar
                 /* Any Char except SpaceChar */

SpaceChar ::= ' ' /* Unicode character U+0020 */

Char ::= /* Any character */

/* See §1.3.1 String Literals in Syntactic Grammars for details
   on the interpetation of string literals in this grammar */
```

A query satisfies a `FullTextSearch` constraint if the value (or values) of the full-text indexed properties within the full-text search scope satisfy the specified `fullTextSearchExpression`, evaluated as follows:

- A `term` *not* preceded with "-" (minus sign) is satisfied only if the value *contains* that term.
- A term preceded with "-" (minus sign) is satisfied only if the value *does not contain* that term.
- Terms separated by whitespace are implicitly "ANDed".
- Terms separated by "OR" are "ORed".
- "AND" has higher precedence than "OR".

- Within a term, each "`"`" (double quote), "`-`" (minus sign), and "`\`" (backslash) must be escaped by a preceding "`\`".

The query is *invalid* if:

- `selectorName` is not the name of a selector in the query, or
- `fullTextSearchExpression` does not conform to the above grammar (as augmented by the implementation).

The grammar and semantics described above defines the *minimal* requirement, meaning that any search string accepted as valid by an implementation must conform to this grammar. An implementation may, however, restrict acceptable search strings further by augmenting this grammar and expanding the semantics appropriately.

If `propertyName` is specified but, for a node-tuple, the `selectorName` node does not have a property named `propertyName`, the query is *valid* but the constraint is not satisfied.

JCR-SQL2

```
FullTextSearch ::=
    'CONTAINS(' ([selectorName'.']propertyName |
                 selectorName'.*') ','
                 FullTextSearchExpression ')'
                  /* If only one selector exists in this query,
                     explicit specification of the selectorName
                     preceding the propertyName is optional */

FullTextSearchExpression ::= BindVariable |
                             ''' FullTextSearchLiteral '''
                             /* see above */
```

JCR-JQOM

A `FullTextSearch` is created with:

```
FullTextSearch QueryObjectModelFactory.
    fullTextSearch(String selectorName,
                   String propertyName,
                   StaticOperand fullTextSearchExpression)
```

`FullTextSearch` extends `Constraint` and declares:

```
String FullTextSearch.getSelectorName()

String FullTextSearch.getPropertyName()

StaticOperand FullTextSearch.getFullTextSearchExpression()
```

6.7.20 SameNode

AQM

```
type SameNode extends Constraint ::=
     Name selectorName,
```

```
      Path path
```

Tests whether the `selectorName` node is reachable by the absolute path specified. A node-tuple satisfies the constraint only if:

```
selectorNode.isSame(session.getNode(path))
```

would return `true`, where `selectorNode` is the node for the specified selector.

The query is *invalid* if:

- `selectorName` is not the name of a selector in the query, or
- `path` is not a syntactically valid absolute path (see §3.3.4 *Lexical Path Grammar*). Note, however, that if `path` is syntactically valid but does not identify a node in the workspace (or the node is not visible to this session, because of access control constraints), the query is *valid* but the constraint is not satisfied.

JCR-SQL2

```
SameNode ::= 'ISSAMENODE(' [selectorName ','] Path ')'
             /* If only one selector exists in this query, explicit
                specification of the selectorName is optional */
```

JCR-JQOM

A `SameNode` is created with:

```
SameNode QueryObjectModelFactory.
    sameNode(String selectorName, String path)
```

`SameNode` extends `Constraint` and declares:

```
String SameNode.getSelectorName()

String SameNode.getPath()
```

6.7.21 ChildNode

AQM

```
type ChildNode extends Constraint ::=
      Name selectorName,
      Path path
```

Tests whether the `selectorName` node is a child of a node reachable by the absolute path specified. A node-tuple satisfies the constraint only if:

```
selectorNode.getParent().isSame(session.getNode(path))
```

would return `true`, where `selectorNode` is the node for the specified selector.

The query is *invalid* if:

- `selectorName` is not the name of a selector in the query, or

- `path` is not a syntactically valid absolute path (see §3.3.4 *Lexical Path Grammar*). Note, however, that if `path` is syntactically valid but does not identify a node in the workspace (or the node is not visible to this session, because of access control constraints), the query is *valid* but the constraint is not satisfied.

JCR-SQL2

```
ChildNode ::= 'ISCHILDNODE(' [selectorName ','] Path ')'
            /* If only one selector exists in this query, explicit
               specification of the selectorName is optional */
```

JCR-JQOM

A `ChildNode` is created with:

```
ChildNode QueryObjectModelFactory.
    childNode(String selectorName, String path)
```

`ChildNode` extends `Constraint` and declares:

```
String ChildNode.getSelectorName()

String ChildNode.getParentPath()
```

6.7.22 DescendantNode

AQM

```
type DescendantNode extends Constraint ::=
      Name selectorName,
      Path path
```

Tests whether the `selectorName` node is a descendant of a node reachable by the absolute path specified. A node-tuple satisfies the constraint only if:

```
selectorNode.getAncestor(n).isSame(session.getNode(path))
    &&  selectorNode.getDepth() > n
```

would return `true` for some non-negative integer `n`, where `selectorNode` is the node for the specified selector.

The query is *invalid* if:

- `selectorName` is not the name of a selector in the query, or
- `path` is not a syntactically valid absolute path (see §3.3.4 *Lexical Path Grammar*). Note, however, that if `path` is syntactically valid but does not identify a node in the workspace (or the node is not visible to this session, because of access control constraints), the query is *valid* but the constraint is not satisfied.

JCR-SQL2

```
DescendantNode ::=
      'ISDESCENDANTNODE(' [selectorName ','] Path ')'
```

```
      /* If only one selector exists in this query, explicit
         specification of the selectorName is optional */
```

JCR-JQOM

A `DescendantNode` is created with:

```
DescendantNode QueryObjectModelFactory.
    descendantNode(String selectorName, String path)
```

`DescendantNode` extends `Constraint` and declares:

```
String DescendantNode.getSelectorName()

String DescendantNode.getAncestorPath()
```

6.7.23 Path

AQM

```
type Path
```

A JCR path.

JCR-SQL2

```
Path ::= '[' quotedPath ']' |
         '[' simplePath ']' |
         simplePath

quotedPath ::= /* A JCR Path that contains non-SQL-legal
                  characters */

simplePath ::= /* A JCR Name that contains only SQL-legal
                  characters[11] */
```

JCR-JQOM

A JCR path in string form (standard, non-standard, normalized or non-normalized, see §3.3.5 *Standard and Non-Standard Form* and §3.3.6.3 *Normalized Paths*).

6.7.24 Operand

AQM

```
abstract type Operand
```

[11] See the SQL:92 rules for `<regular identifier>` (in ISO/IEC 9075:1992 §5.2 `<token>` and `<separator>`).

JCR-SQL2

```
Operand ::= StaticOperand | DynamicOperand
            /* 'Operand' not referenced in JCR-SQL2
               grammar. For possible future use. */
```

JCR-JQOM

`Operand` is an empty interface with subclasses `StaticOperand` and `DynamicOperand`.

6.7.25 StaticOperand

AQM

```
abstract type StaticOperand extends Operand
```

An operand whose value can be determined from static analysis of the query, prior to its evaluation.

JCR-SQL2

```
StaticOperand ::= Literal | BindVariableValue
```

JCR-JQOM

`StaticOperand` is an empty interface with subclasses `Literal` and `BindVariableValue`.

6.7.26 DynamicOperand

AQM

```
abstract type DynamicOperand extends Operand
```

An operand whose value can only be determined in evaluating the query.

JCR-SQL2

```
DynamicOperand ::= PropertyValue | Length | NodeName |
                   NodeLocalName | FullTextSearchScore |
                   LowerCase | UpperCase
```

JCR-JQOM

`DynamicOperand` is an empty interface with subclasses `PropertyValue`, `Length`, `NodeName`, `NodeLocalName`, `FullTextSearchScore`, `LowerCase` and `UpperCase`.

6.7.27 PropertyValue

AQM

```
type PropertyValue extends DynamicOperand ::=
      Name selectorName,
      Name propertyName
```

Evaluates to the value (or values, if multi-valued) of a property.

If, for a node-tuple, the `selectorName` node does not have a property named `propertyName`, the operand evaluates to `null`.

The query is *invalid* if `selectorName` is not the name of a selector in the query.

JCR-SQL2

```
PropertyValue ::= [selectorName'.'] propertyName
                  /* If only one selector exists in this query,
                     explicit specification of the selectorName is
                     optional */
```

JCR-JQOM

A `PropertyValue` is created with:

```
PropertyValue QueryObjectModelFactory.
    propertyValue(String selectorName, String propertyName)
```

`PropertyValue` extends `DynamicOperand` and declares:

```
String PropertyValue.getSelectorName()

String PropertyValue.getPropertyName()
```

6.7.28 Length

AQM

```
type Length extends DynamicOperand ::=
      PropertyValue propertyValue
```

Evaluates to the length (or lengths, if multi-valued) of a property. In evaluating this operand, a repository *should* use the semantics defined in §3.6.7 *Length of a Value*.

If `propertyValue` evaluates to `null`, the `Length` operand also evaluates to `null`.

JCR-SQL2

```
Length ::= 'LENGTH(' PropertyValue ')'
```

JCR-JQOM

A `Length` is created with:

```
Length QueryObjectModelFactory.
    length(PropertyValue propertyValue)
```

`Length` extends `DynamicOperand` and declares:

```
PropertyValue Length.getPropertyValue()
```

6.7.29 NodeName

AQM

```
type NodeName extends DynamicOperand ::=
      Name selectorName
```

Evaluates to a `NAME` value equal to the *JCR name* of a node.

The query is *invalid* if `selectorName` is not the name of a selector in the query.

JCR-SQL2

```
NodeName ::= 'NAME(' [selectorName] ')'
             /* If only one selector exists in this query, explicit
                specification of the selectorName is optional */
```

JCR-JQOM

A `NodeName` is created with:

```
NodeName QueryObjectModelFactory.
    nodeName(String selectorName)
```

`NodeName` extends `DynamicOperand` and declares:

```
String NodeName.getSelectorName()
```

6.7.30 NodeLocalName

AQM

```
type NodeLocalName extends DynamicOperand ::=
      Name selectorName
```

Evaluates to a `STRING` value equal to the *JCR local name* of a node.

The query is *invalid* if `selectorName` is not the name of a selector in the query.

JCR-SQL2

```
NodeLocalName ::= 'LOCALNAME(' [selectorName] ')'
                  /* If only one selector exists in this query,
                     explicit specification of the selectorName is
                     optional */
```

JCR-JQOM

A `NodeLocalName` is created with:

```
NodeLocalName QueryObjectModelFactory.
    nodeLocalName(String selectorName)
```

`NodeLocalName` extends `DynamicOperand` and declares:

```
String NodeLocalName.getSelector()
```

6.7.31 FullTextSearchScore

AQM

```
type FullTextSearchScore extends DynamicOperand ::=
      Name selectorName
```

Evaluates to a `DOUBLE` value equal to the full-text search score of a node.

Full-text search score ranks a selector's nodes by their relevance to the `fullTextSearchExpression` specified in a `FullTextSearch`. The values to which `FullTextSearchScore` evaluates and the interpretation of those values are implementation specific. `FullTextSearchScore` may evaluate to a constant value in a repository that does not support full-text search scoring or has no full-text indexed properties.

The query is *invalid* if `selector` is not the name of a selector in the query.

JCR-SQL2

```
FullTextSearchScore ::= 'SCORE(' [selectorName] ')'
                        /* If only one selector exists in this query,
                           explicit specification of the selectorName
                           is optional */
```

JCR-JQOM

A `FullTextSearchScore` is created with:

```
FullTextSearchScore QueryObjectModelFactory.
    fullTextSearchScore(String selectorName)
```

`FullTextSearchScore` extends `DynamicOperand` and declares:

```
String FullTextSearchScore.getSelector()
```

6.7.32 LowerCase

AQM

```
type LowerCase extends DynamicOperand ::=
      DynamicOperand operand
```

Evaluates to the lower-case string value (or values, if multi-valued) of `operand`.

If `operand` does not evaluate to a string value, its value is first converted to a string as described in §3.6.4 *Property Type Conversion*. The lower-case string value is computed as though the `toLowerCase()` method of `java.lang.String` were called.

If `operand` evaluates to `null`, the `LowerCase` operand also evaluates to `null`.

JCR-SQL2

```
LowerCase ::= 'LOWER(' DynamicOperand ')'
```

JCR-JQOM

A `LowerCase` is created with:

```
LowerCase QueryObjectModelFactory.
    lowerCase(DynamicOperand operand)
```

`LowerCase` extends `DynamicOperand` and declares:

```
DynamicOperand LowerCase.getOperand()
```

6.7.33 UpperCase

AQM

```
type UpperCase extends DynamicOperand ::=
      DynamicOperand operand
```

Evaluates to the upper-case string value (or values, if multi-valued) of `operand`.

If `operand` does not evaluate to a string value, its value is first converted to a string as described in §3.6.4 *Property Type Conversion*. The upper-case string value is computed as though the `toUpperCase()` method of `java.lang.String` were called.

If `operand` evaluates to `null`, the `UpperCase` operand also evaluates to `null`.

JCR-SQL2

```
UpperCase ::= 'UPPER(' DynamicOperand ')'
```

JCR-JQOM

An `UpperCase` is created with:

```
UpperCase QueryObjectModelFactory.
    upperCase(DynamicOperand operand)
```

`UpperCase` extends `DynamicOperand` and declares:

```
DynamicOperand UpperCase.getOperand()
```

6.7.34 Literal

AQM

```
type Literal extends StaticOperand ::=
      javax.jcr.Value Value
```

A JCR value.

JCR-SQL2

```
Literal ::= CastLiteral | UncastLiteral

CastLiteral ::= 'CAST(' UncastLiteral ' AS ' PropertyType ')'
```

```
PropertyType ::= 'STRING' | 'BINARY' | 'DATE' | 'LONG' | 'DOUBLE' |
                 'DECIMAL' | 'BOOLEAN' | 'NAME' | 'PATH' |
                 'REFERENCE' | 'WEAKREFERENCE' | 'URI'

UncastLiteral ::= UnquotedLiteral | ''' UnquotedLiteral ''' |
                  '"' UnquotedLiteral '"'

UnquotedLiteral ::= /* String form of a JCR Value, as defined in
                       §3.5.4 Conversion of Values */
```

An `UncastLiteral` may be interpreted as a `Value` of property type `STRING` or some other type inferred from static analysis. A `CastLiteral`, on the other hand, is interpreted as the string form of a `Value` of the `PropertyType` indicated.

JCR-JQOM

A JCR `Value`. A `Value` object can be created using `ValueFactory` (see §6.10 *Literal Values*). Note that unlike in the case of JCR-SQL2, property type information is intrinsic to the `Value` object, so no equivalent of the `CAST` function is needed in JCR-JQOM.

6.7.35 BindVariable

AQM

```
type BindVariableValue extends StaticOperand ::=
      Prefix bindVariableName
```

Evaluates to the value of a bind variable.

The query is *invalid* if no value is bound to `bindVariableName`.

JCR-SQL2

```
BindVariableValue ::= '$'bindVariableName

bindVariableName ::= Prefix
```

JCR-JQOM

A `BindVariableValue` is created with:

```
BindVariableValue QueryObjectModelFactory.
    bindVariableValue(String bindVariableName)
```

`BindVariableValue` extends `StaticOperand` and declares:

```
StaticOperand BindVariableValue.getBindVariableName()
```

6.7.36 Prefix

AQM

```
type Prefix
```

A JCR prefix.

The query is *invalid* if the prefix does not satisfy the `prefix` production in §3.2.5.2 *Qualified Form*.

JCR-SQL2

```
Prefix ::= /* A String that conforms to the JCR Name
              prefix syntax. Not required to be an actual
              prefix in use in the repository. The prefix
              syntax is used simply to characterize the
              range of possible variables. */
```

JCR-JQOM

A string that conforms to the JCR Name prefix syntax. This is not required to be an actual prefix in use in the repository. The prefix syntax is used simply to characterize the range of possible variables.

6.7.37 Ordering

AQM

```
type Ordering ::=
      DynamicOperand operand,
      Order order
```

Determines the relative order of two node-tuples by evaluating `operand` for each.

For a first node-tuple, `nt1`, for which `operand` evaluates to `v1`, and a second node-tuple, `nt2`, for which `operand` evaluates to `v2`:

If operand is a `PropertyValue` (see §6.7.27 *PropertyValue*) of a property `P` and the *query-orderable* attribute of the property definition of `P` is `false` (see §3.7.3.5 *Query-Orderable*) then the relative order of `nt1` and `nt2` is implementation determined, otherwise, if the *query-orderable* attribute is `true`, then:

If `order` is `Ascending`, then:

- if either `v1` is `null`, `v2` is `null`, or both `v1` and `v2` are `null`, the relative order of `nt1` and `nt2` is implementation determined, otherwise
- if `v1` is a different property type than `v2`, the relative order of `nt1` and `nt2` is implementation determined, otherwise
- if `v1` *is ordered before* `v2`, as described in §3.6.5 *Comparison of Values*, then `nt1` precedes `nt2`, otherwise
- if `v1` *is ordered after* `v2`, as described in §3.6.5 *Comparison of Values*, then `nt2` precedes `nt1`, otherwise
- the relative order of `nt1` and `nt2` is implementation determined and may be arbitrary.

Otherwise, if `order` is `Descending`, then:

- if either `v1` is `null`, `v2` is `null`, or both `v1` and `v2` are `null`, the relative order of `nt1` and `nt2` is implementation determined, otherwise
- if `v1` is a different property type than `v2`, the relative order of `nt1` and `nt2` is implementation determined, otherwise
- if `v1` *is ordered before* `v2`, as described in §3.6.5 *Comparison of Values*, then `nt2` precedes `nt1`, otherwise
- if `v1` *is ordered after* `v2`, as described in §3.6.5 *Comparison of Values*, then `nt1` precedes `nt2`, otherwise
- the relative order of `nt1` and `nt2` is implementation determined and may be arbitrary.

The query is *invalid* if `operand` does not evaluate to a scalar value.

JCR-SQL2

```
orderings ::= Ordering {',' Ordering}

Ordering ::= DynamicOperand [Order]
```

If `Order` is omitted in the JCR-SQL2 statement the default is `ASC` (see §6.7.38 *Order*).

JCR-JQOM

An ascending `Ordering` is created with:

```
Ordering QueryObjectModelFactory.
    ascending(DynamicOperand operand)
```

A descending `Ordering` is created with:

```
Ordering QueryObjectModelFactory.
    descending(DynamicOperand operand)
```

`Ordering` declares:

```
DynamicOperand Ordering.getOperand()

String Ordering.getOrder()
```

6.7.38 Order

AQM

```
enum Order ::=
      Ascending,
      Descending
```

`Order` is either `Ascending` or `Descending`.

JCR-SQL2

```
Order ::= Ascending | Descending
```

```
Ascending ::= 'ASC'

Descending ::= 'DESC'
```

JCR-JQOM

An order is a `String` constant. One of:

```
QueryObjectModelConstants.JCR_ORDER_ASCENDING

QueryObjectModelConstants.JCR_ORDER_DESCENDING
```

6.7.39 Column

AQM

```
type Column ::=
      Name selectorName,
      Name? propertyName,
      Name? columnName
```

Defines a column to include in the tabular view of query results.

If `propertyName` is not specified, a column is included for each single-valued non-residual property of the node type specified by the `nodeType` attribute of the selector `selectorName`.

If `propertyName` is specified, `columnName` is required and used to name the column in the tabular results. If `propertyName` is not specified, `columnName` must not be specified, and the included columns will be named "*`selectorName.propertyName`*".

The query is *invalid* if:

- `selectorName` is not the name of a selector in the query, or
- `propertyName` is specified but does not evaluate to a scalar value, or
- `propertyName` is specified but `columnName` is omitted, or
- `propertyName` is omitted but `columnName` is specified, or
- the columns in the tabular view are not uniquely named, whether those column names are specified by `columnName` (if `propertyName` is specified) or generated as described above (if `propertyName` is omitted).

If `propertyName` is specified but, for a node-tuple, the `selectorName` node does not have a property named `propertyName`, the query is *valid* and the column has `null` value.

JCR-SQL2

```
columns ::= (Column ',' {Column}) | '*'

Column ::= ([selectorName'.']propertyName
           ['AS' columnName]) |
```

```
            (selectorName'.*')
            /* If only one selector exists in this query, explicit
               specification of the selectorName preceding the
               propertyName is optional */

selectorName ::= Name

propertyName ::= Name

columnName ::= Name
```

JCR-JQOM

A `Column` is created with:

```
Column QueryObjectModelFactory.
    column(String selectorName,
           String propertyName,
           String columnName)
```

`Column` declares:

```
String Column.getSelectorName()

String Column.getPropertyName()

String Column.getColumnName()
```

6.8 QueryManager

The query function is accessed through the `QueryManager` object, acquired through

```
QueryManager Workspace.getQueryManager().
```

6.8.1 Supported Languages

```
String[] QueryManager.getSupportedQueryLanguages()
```

returns an array of strings representing the supported query languages. In all repositories that support query, the array will contain at least the string constants

`Query.JCR_SQL2` and

`Query.JCR_JQOM`.

Any additional languages also supported will also be listed in the returned array.

6.9 Query Object

A new `Query` object can be created with

```
Query QueryManager.
    createQuery(String statement, String language).
```

The `language` parameter is a string representing one of the supported languages. The `statement` parameter is the query statement itself. This method is used for languages that are string-based (i.e., most languages, such as JCR-SQL2) as well

as for the *string serializations* of non-string-based languages (such as JCR-JQOM). For example, the call

```
QM.createQuery(S, Query.JCR_SQL2),
```

where `QM` is the `QueryManager` and `S` is a JCR-SQL2 statement, returns a `Query` object encapsulating `S`.

However, the call

```
QM.createQuery(S, Query.JCR_JQOM)
```

also works. It returns a `QueryObjectModel` (a subclass of `Query`) holding the JCR-JQOM object tree equivalent to `S`.

In either case the returned `Query` object encapsulates the resulting query. In some repositories the first method call (with JCR-SQL2 specified) may also result in a `QueryObjectModel`, though this is not required.

6.9.1 QueryObjectModelFactory

To programmatically build a query tree using JCR-JQOM the user acquires a `QueryObjectModelFactory` using

```
QueryObjectModelFactory QueryManager.getQOMFactory().
```

The user then builds the query tree using the factory methods of `QueryObjectModelFactory`, ultimately resulting in a `QueryObjectModel` object (a subclass of `Query)` representing the query.

6.9.1.1 Serialized Query Object Model

The JCR-SQL2 language, in addition to being a query language in its own right is also the standard serialization of a valid JCR-JQOM object tree. Since the two languages are formally equivalent they can always be roundtripped.

6.9.2 Getting the Statement

```
String Query.getStatement()
```

returns the statement set for the query. If the `Query` was created with an explicitly supplied `statement` string parameter using `QueryManager.createQuery` then this method returns that statement. The statement returned must be semantically identical to the original statement but need not be an identical string (for example, it may be normalized).

If the `Query` is actually a `QueryObjectModel` created with `QueryObjectModelFactory.createQuery` then `Query.getStatement` must return the serialized form of the query, in JCR-SQL2 syntax.

6.9.3 Getting the Language

```
String Query.getLanguage()
```

returns the language in which the query is specified. If the `Query` was created with an explicitly supplied `language` string parameter using `QueryManager.createQuery` then this method returns that string.

If the `Query` is actually a `QueryObjectModel` created with `QueryObjectModelFactory.createQuery` then `Query.getLanguage` will return the string constant `Query.JCR_SQL2`.

6.9.4 Query Limit

```
Query.setLimit(long limit)
```

Sets the maximum size of the result set, expressed in terms of the number of `Row`s, as found in the table-view of the `QueryResult` (see §6.11 *QueryResult*).

6.9.5 Query Offset

```
Query.setOffset(long offset)
```

Sets the offset within the full result set at which the returned result set should start, expressed in terms of the number of `Row`s to skip, as found in the table-view of the `QueryResult` (see §6.11 *QueryResult*).

6.9.6 Bind Variables

A query may contain variables.

```
void Query.bindValue(String varName, Value value)
```

binds `value` to the variable `varName`.

In JCR-SQL2 a bind variable is indicated by a leading dollar-sign. In JCR-JQOM it is a QOM object created with the `QueryObjectModelFactory` (see §6.7.35 *BindVariable*).

The method

```
String[] Query.getBindVariableNames()
```

returns the names of the bind variables in the query. If the query does not contains any bind variables then an empty array is returned.

6.9.7 Stored Query

When a new `Query` object is first created it is a *transient query*. If the repository supports the node type `nt:query`, then a transient query can be stored in content by calling

```
Node Query.storeAsNode(String absPath).
```

This creates an `nt:query` node at the specified path. A `save` is required to persist the node.

6.9.7.1 nt:query

The `nt:query` node type is defined as follows:

```
[nt:query]
  - jcr:statement (STRING)
  - jcr:language (STRING)
```

`jcr:statement` holds the string returned by `Query.getStatement()`.

`jcr:language` holds the string returned by `Query.getLanguage()`.

If the language of this query is JCR-JQOM, `jcr:statement` will hold the JCR-SQL2 serialization of the JCR-JQOM object tree and `Query.getStatement()` will return that string. Also, since the original query was constructed using JCR-JQOM, `jcr:language` records the language as "`JCR-JQOM`" and `Query.getLanguage()` returns "`JCR-JQOM`".

6.9.7.2 Stored Query Path

```
String Query.getStoredQueryPath()
```

returns the absolute path of a `Query` that has been stored as a node.

6.9.7.3 Retrieving a Stored Query

```
Query QueryManager.getQuery(Node node)
```

retrieves a previously persisted query and instantiates it as a `Query` object.

6.9.7.4 Namespace Fragility

Note that the query statement stored within a stored query (the value of the property `jcr:statement`) is stored as a simple string. Therefore, if it contains qualified JCR names it will be *namespace-fragile*. If the stored query is run in a context where a prefix used maps to a different namespace than it did upon creation then the query will not reproduce the original result. To mitigate this, users should either,

- always use expanded form names within queries, or
- always ensure that appropriate namespace mappings are in place when a stored query is executed.

6.10 Literal Values

When creating a `Comparison` object (see 6.7.16 *Comparison*) a user may wish to pass a literal property value (see 6.7.34 *Literal*) in the form of a `Value` object. `Value` objects are created using the `ValueFactory` acquired through

```
ValueFactory Session.getValueFactory().
```

(see §10.4.3 *Creating Value Objects*).

6.11 QueryResult

Once a query has been defined, it can be executed. The method

```
QueryResult Query.execute()
```

returns the a `QueryResult` object. The `QueryResult` is returned in two formats: as a table and as a list of nodes.

6.11.1 Table View

The table view of a result is accessed with

```
RowIterator QueryResult.getRows()
```

The returned `RowIterator` holds a series of `Row` objects. A `Row` object represents a single row of the query result table which corresponds to a node-tuple returned by the query.

6.11.1.1 Row

Upon retrieving an individual `Row`, the set of `Value`s making up that row can be retrieved with

```
Value[] Row.getValues()
```

The values are returned in that same order as their corresponding column names are returned by `QueryResult.getColumns`.

```
Value Row.getValue(String columnName)
```

returns the `Value` of the indicated column of the `Row`. The names of the columns can be retrieved with

```
String[] QueryResult.getColumnNames().
```

In queries with only one selector included among the specified columns, each `Row` corresponds to a single `Node`. In such cases

```
Node Row.getNode()
```

returns that `Node`.

In queries with more than one selector included among the specified columns, a particular selector must be indicated in order to retrieve its corresponding `Node` . This is done using

```
Node Row.getNode(String selectorName).
```

The available selector names can be retrieved with

```
String[] QueryResult.getSelectorNames().
```

If the `Row` is from a result involving outer joins, it may have no `Node` corresponding to the specified selector, in which case this method returns `null`.

The methods

```
String Row.getPath() and

String Row.getPath(String selectorName)
```

are equivalent to `Row.getNode().getPath()` and `Row.getNode(String selectorName).getPath()`, respectively. However, some implementations may be able gain efficiency by not resolving the actual `Node`.

The method

```
double Row.getScore(String selectorName)
```

returns the full text search score for this row that is associated with the specified selector. This is equivalent to the score of the `Node` associated with that this `Row` and that selector.

If no `FullTextSearchScore` AQM object (see §6.7.31 *FullTextSearchScore*) is associated with the specified selector this method will still return a value but that value may not be meaningful or may simply reflect the minimum possible relevance level (for example, in some systems this might be a score of 0).

If this `Row` is from a result involving outer joins, it may have no `Node` corresponding to the specified selector, in which case this method returns an implementation selected value, as it would if there were no `FullTextSearchScore` associated with the selector.

The method

```
double Row.getScore()
```

works identically to `Row.getScore(String selectorName)`, but only in cases where there is exactly one selector and therefore its name need not be explicitly specified.

6.11.2 Node View

For queries with only one selector

```
QueryResult.getNodes()
```

returns an iterator over all matching nodes in the order specified by the query. For queries with more than one selector the order in which nodes are returned is implementation-specific.

6.12 Query Scope

Each `Query` is bound to a `Session` object via the `QueryManager` through which it was created and the `Workspace` object through which that `QueryManager` was acquired. Through its associated `Workspace` and `Session` objects a query is therefore bound to a single persistent workspace and a single transient store.

6.12.1 Access Restrictions

A query result always respects the access restrictions of its bound `Session`. This includes all restrictions, as reflected in the *capabilities* of the `Session`, which encompasses *privileges*, *permissions* and *other restrictions* (see §9 *Permissions and Capabilities*).

In general, if the bound `Session` does not have read access to a particular item, then that item will not be included in the result set even if it would otherwise constitute a match.

6.12.2 Queryable Content

A query runs against *either*

- the content of its bound persistent workspace, *without regard to any pending changes* in its bound transient store, or
- the content of its bound persistent workspace *as modified by the pending changes* in its bound transient store.

The choice of which scope to use is an implementation-variant.

6.12.3 Query Result Items

Regardless of which scope is used, when an item is accessed from within a `QueryResult` object, the state of the item returned will obey the same semantics as if it were retrieved using a normal `Node.getNode` or `Node.getProperty`: the item state will reflect any pending changes in transient store of the `Session`. As a result, it is possible that an item returned as a match will not reflect the state that caused it to *be* a match (i.e., its persistent state). Applications can clear the `Session` (either through `save` or `refresh(false)`) before running a query in order to avoid such discrepancies.

7 Export

A JCR repository must support export of content to two XML formats: *system view* and *document view*.

7.1 Exporting a Subgraph

Export operates on a subgraph of a workspace. Given a repository *R* with workspace *W* and a node *N* within *W* the following sections describe the algorithm for producing the system view and document view serializations of the subgraph rooted at *N*.

In a repository that supports *shareable nodes* the set of nodes below *N* may not be a tree, it may, more generally, be a subgraph with unique source *N* (see §3.9 *Shareable Nodes Model*).

7.2 System View

The exported system view XML document is constructed as follows:

1. For every namespace used within the subgraph rooted at *N*, the corresponding JCR namespace mapping in the current session *is included* as an XML namespace declaration such that any use of a namespace prefix is within the scope of the appropriate namespace declaration.

2. Other JCR namespace mappings in the current session *may be included* as XML namespace declarations in the exported document.

3. The JCR namespace mapping of the prefix `xml` *may be excluded* from the namespace declarations in the exported document.

4. A namespace declaration for the URI `http://www.jcp.org/jcr/sv/1.0`, is included such that any use of the corresponding namespace prefix is within the scope of the declaration. In this section the prefix `sv` is assumed, making the declaration `xmlns:sv="http://www.jcp.org/jcr/sv/1.0"`.

5. Each JCR node becomes an XML element `<sv:node>`.

6. Each JCR property becomes an XML element `<sv:property>`.

7. The name of each JCR node or property becomes the value of the `sv:name` attribute of the corresponding `<sv:node>` or `<sv:property>` element.

8. If the root node of a workspace is included in the serialized subgraph, it receives the name `jcr:root`.

9. The property type of each content repository property is recorded in the `sv:type` attribute of the corresponding `<sv:property>` element, using the standard string forms for property type names as returned by the method `PropertyType.nameFromValue`.

10. The value of each `BINARY` JCR property is Base64[12] encoded and the resulting string is included as XML text within an `<sv:value>` element within the `<sv:property>` element.

11. The value of each non-`BINARY` JCR property is converted to string form according to the standard conversion (see §3.6.4 *Property Type Conversion*) and the resulting string is included as XML text within an `<sv:value>` element within the `<sv:property>` element.

    a. Entity references are used to escape characters which cannot be included as literals within XML text (see §7.5 *Escaping of Values*).

    b. If, after conversion to string and entity escaping is performed, the string form of a value still contains characters which cannot appear in an XML document (neither as literals nor as character references[13]) then:

        i. The string form is further encoded using Base64 encoding.

        ii. The attribute `xsi:type="xsd:base64Binary"` is added to the `<sv:value>` element.

        iii. The namespace mappings for `xsi` and `xsd` are added to the exported XML document so that the `xsi:type` attribute is within their scope. The namespace declarations required are `xmlns:xsd="http://www.w3.org/2001/XMLSchema"` and `xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"`. Note that the prefixes representing these two namespaces need not be *literally* "`xsd`" and "`xsi`". Any two prefixes are permitted as long as the corresponding namespace declarations are changed accordingly.

12. A multi-value property is converted to an `<sv:property>` element containing multiple `<sv:value>` elements. The order of the `<sv:value>` elements reflects the order of the value array returned by `Property.getValues`. If a property is multi-valued but happens to have only one value, then the attribute `sv:multiple="true"` *must* be added to the corresponding `<sv:property>` element. If the property is multi-valued and has more than one value then the `sv:multiple="true"` attribute *may* be added.

---

[12] See `http://tools.ietf.org/html/rfc4648` §4.

[13] See `http://www.w3.org/TR/REC-xml/#charsets`, `http://www.w3.org/TR/REC-xml/#NT-CharRef`, and `http://www.w3.org/TR/REC-xml/#wf-Legalchar`.

13. The hierarchy of the content repository nodes and properties is reflected in the hierarchy of the corresponding XML elements.

14. Within an `<sv:node>` element all `<sv:property>` subelements must occur before the first `<sv:node>` subelement.

15. The first `<sv:property>` element in an `<sv:node>` element must be `jcr:primaryType` (see §3.7.10 *Base Primary Node Type*).

16. If a node has a `jcr:mixinTypes` property, then the second `<sv:property>` element in the `<sv:node>` element must be `jcr:mixinTypes` (see §3.7.10 *Base Primary Node Type*).

17. In the case of referenceable nodes, the third `<sv:property>` element in the `<sv:node>` element must be `jcr:uuid` (see §3.8.1.1 *mix:referenceable*).

18. The order of the `<sv:node>` subelements of a parent `<sv:node>` must reflect the order in which the corresponding child nodes are returned by `Node.getNodes()`.

19. Shared nodes are exported as described in §14.7 *Export*.

A writable repository may support import using system view (see §11 *Import*).

7.3 Document View

The document view provides a more human-readable serialization than system view. Unlike system view, document view is lossy. It does not preserve all information in the subgraph.

1. For every namespace used within the subgraph rooted at *N*, the corresponding JCR namespace mapping in the current session *is included* in the exported document such that any use of the namespace prefix in the exported document is within the scope of the appropriate namespace declaration.

2. Other JCR namespace mappings in the current session *may be included* as XML namespace declarations in the exported document.

3. The JCR namespace mapping of the prefix `xml` *may be excluded* from the namespace declarations in the exported document.

4. Each JCR node `N` becomes an XML element of the same name, `N`.

5. If the root node of a workspace is included in the serialized subgraph, it becomes an XML elements with the name `jcr:root`.

6. Each child node `C` of `N` becomes a subelement `C` of XML element `N`.

7. The order of the subelements of element `N` must reflect the order in which the corresponding child nodes are returned by `Node.getNodes`.

8. Each property `P` of node `N` becomes an XML attribute `P` of XML element `N`.

9. If P is a BINARY property its value is Base64 encoded. The resulting string becomes the value of the XML attribute P.

10. If P is a non-BINARY property its value is converted to string form according to the standard conversion (see §3.6.4 *Property Type Conversion*). Entity references are used to escape characters which cannot be included as literals within attribute values (see §7.5 *Escaping of Values*).

A writable repository may support document view import (see §11.1 *Importing Arbitrary XML*).

The following sections describe the exceptions to the above general rules.

7.3.1 XML Text

In a repository that supports it, on document view import XML text is converted to the special node/property structure jcr:xmltext/jcr:xmlcharacters (see §11.1 *Importing Document View*). When this structure is exported back to XML the process is reversed.

If a child node of N called jcr:xmltext is encountered and that jcr:xmltext node has one and only one child item and that item is a single-valued property called jcr:xmlcharacters, then the jcr:xmltext node is not converted into an XML element. Instead, the value of the jcr:xmlcharacters property becomes text within the body of the XML element N. Entity references are used to escape characters which cannot be included as literals within XML text (see §7.5 *Escaping of Values*) however, escaping of whitespace is not performed (see §7.3.3 *Multi-Value Properties*). Two or more jcr:xmltext nodes adjacent within the ordering of a child node set will have the values of their respective jcr:xmlcharacters concatenated into a single resulting XML text node.

7.3.2 Invalid Item Names

If the name of a content repository item I is not a valid XML element or attribute name (as the case may be) then on export the repository may either ignore the item in question or employ the escaping scheme described in §7.4 *Escaping of Names*. Which approach is taken is implementation-dependent.

7.3.3 Multi-Value Properties

If a multi-value property P is encountered on export, the repository may either ignore the multi-value property or serialize it as an attribute whose value is an XML Schema list type[14] (i.e., a whitespace-delimited list of strings). If the latter approach is taken then:

[14] See http://www.w3.org/TR/xmlschema-0/#ListDt for more information about the XML Schema list type.

- Each value in the property is converted to a string according to standard conversion, see §3.6.4 *Property Type Conversion*. If the multi-value property contains no values, then it is serialized as an empty string.
- All literal whitespace within each string is escaped, as well as any characters that should not be included as literals in any case, see §7.5 *Escaping of Values*.
- The final attribute value is constructed by concatenating the resulting strings, with the addition of the space delimiter, into a single string. The order of concatenation must be the same as the order in which the values appear in the `Value` array returned by `Property.getValues`.
- Furthermore, if multi-value property serialization is supported, then a mechanism must be adopted whereby upon re-import the distinction between multi- and single- value properties is not lost, see §7.5 *Escaping of Values*.
- Note that this escaping of space literals does not apply to the value of `jcr:xmltext/jcr:xmlcharacters` when it is converted to XML text. In that case only the standard XML entity escaping is required, regardless of whether multi-value property serialization is supported (see §7.3.1 *XML Text* and §7.5 *Escaping of Values*).

7.3.4 Invalid Characters in Values

If the string form of the value of property `P` contains characters which cannot appear in XML documents at all (neither as literals nor as character references[15]) then the attribute `P` is simply excluded from the document view serialization and does not appear at all.

7.4 Escaping of Names

Though a JCR prefix is always a valid XML prefix, the JCR local name may not be a valid XML name. Consequently, for document view serialization, each JCR name is converted to a valid XML name (as defined by XML 1.0) by translating invalid characters into escaped numeric entity encodings[16].

The escape character is the underscore ("_"). Any invalid character is escaped as `_x`*`HHHH`*`_`, where *`HHHH`* is the four-digit hexadecimal UTF-16 code for the character.

---

[15] See `http://www.w3.org/TR/REC-xml/#charsets`, `http://www.w3.org/TR/REC-xml/#NT-CharRef`, and `http://www.w3.org/TR/REC-xml/#wf-Legalchar`.

[16] This escaping scheme is based on the scheme described in ISO/IEC 9075-14:2003 for converting arbitrary strings into valid XML element and attribute names.

When producing escape sequences the implementation should use lowercase letters for the hex digits `a-f`. When unescaping, however, both upper and lowercase alphabetic hexadecimal characters must be recognized.

Escaping and unescaping is done by parsing the name from left to right.

The underscore character ("_"), when appearing as literal, is itself escaped if it is followed by *`xHHHH`* where *`H`* is one of the following characters: `0123456789abcdefABCDEF`.

For example,

"`My Documents`" is encoded as "`My_x0020_Documents`".

"`My_Documents`" is not encoded.

"`My_x0020Documents`" is encoded as "`My_x005f_x0020Documents`".

"`My_x0020_Documents`" is encoded as "`My_x005f_x0020_Documents`".

"`My_x0020 Documents`" is encoded as "`My_x005f_x0020_x0020_Documents`".

7.5 Escaping of Values

When a non-`BINARY` value is serialized during either system view or document view export, it is first converted to string form using standard value conversion, see §3.6.4 *Property Type Conversion.* `BINARY` values are encoded using Base64. The resulting string then undergoes any further changes required by the standard XML escaping rules[17].

In document view serialization, if the property being serialized is multi-valued (or if the implementation chooses to encode spaces in single-value properties as well, see below) then the value or values must be further encoded by escaping any occurrence of one of the four whitespace characters: space, tab, carriage return and line feed. The scheme used to encode these characters is the same as that described in §7.4 *Escaping of Names*. Note that in this restricted context, applying those escaping rules amounts to the following: a space becomes `_x0020_`, a tab becomes `_x0009_`, a carriage return becomes `_x000D_`, a line feed becomes `_x000A_` and any underscore (_) that occurs as the first character of a sequence that could be misinterpreted as an escape sequence becomes `_x005f_`.

Finally, in document view export, the value of the attribute representing a multi-value property is constructed by concatenating the results of the above escaping into a space-delimited list.

In document view export (though not in system view), if multi-value property serialization is supported (see §7.3.3 *Multi-Value Properties*) then a mechanism must be adopted whereby upon re-import the distinction between multi- and

[17] See `http://www.w3.org/TR/xml/#syntax`.

single- value properties is not lost. One option is that escaping of space literals must be applied to the value of all single-value properties as well. Another option is that when an XML document is imported in document view, each attribute is assumed to be a single-value property unless out-of-band information defines it to be multi-valued (for example, if the applicable node type defines the property as multi-valued or the XML document is associated with a schema definition that indicates that the attribute is a list value). The approach taken is implementation-specific.

Note that the value of a `jcr:xmlcharacters` property used to represent XML text (see §7.3.1 *XML Text*) is not space-escaped, regardless of the prevailing multi-value property serialization policy.

7.6 Export API

Exported XML can be output either as a stream or as a series of SAX events. The export methods are found in the `Session` object.

7.6.1 System View Export

The methods

```
void Session.exportSystemView(String absPath,
                              ContentHandler contentHandler,
                              boolean skipBinary,
                              boolean noRecurse)
```

and

```
void Session.exportSystemView(String absPath,
                              OutputStream out,
                              boolean skipBinary,
                              boolean noRecurse)
```

serialize the item subgraph starting at `absPath`.

The first method serializes content to XML as a series of SAX events triggered by the repository calling the methods of the supplied `org.xml.sax.ContentHandler`.

The second method serializes content to an XML stream and outputs it to the supplied `java.io.OutputStream`.

The resulting XML is in the system view form.

If `skipBinary` is `true` then any properties of type `BINARY` will be serialized with empty `sv:value` elements. In the case of multi-value `BINARY` properties, the number of values in the property will be reflected in the serialized output, though they will all be empty.

If `skipBinary` is `false` then the actual values of each `BINARY` property are serialized.

If `noRecurse` is `true` then only the node at `absPath` and its properties, but not its child nodes, are serialized. If `noRecurse` is `false` then the entire subgraph is serialized.

7.6.2 Document View Export

The methods

```
void Session.exportDocumentView(String absPath,
                                ContentHandler contentHandler,
                                boolean skipBinary,
                                boolean noRecurse)
```

and

```
void Session.exportDocumentView(String absPath,
                                OutputStream out,
                                boolean skipBinary,
                                boolean noRecurse)
```

work identically to their respective system view variants, except that the resulting XML is in the document view form.

7.7 Export Scope

Export obeys the access restrictions of the bound `Session`. If the `Session` lacks read access to some subsection of the specified content, that section is not exported.

The exported output reflects the state of the bound persistent workspace as modified by the transient store of the bound `Session`. This means that pending changes and all namespace mappings in the namespace registry, as modified by the current session-mappings, are reflected in the output.

7.8 Encoding

XML streams produced by export must be encoded in UTF-8 or UTF-16 as per the XML specification[18].

[18] See `http://www.w3.org/TR/xml/#charsets`.

8 Node Type Discovery

All repositories are required to support methods for the discovery of the following node type-related information:

- Which node types are supported in the repository.
- The definition of a supported node type.
- The node type of a node.
- The definition of an item in the node type of its parent.

8.1 NodeTypeManager Object

A repository has a single, global node type registry that holds all node types available in the repository. The registry is represented by a `NodeTypeManager` object acquired through

```
NodeTypeManager Workspace.getNodeTypeManager().
```

The method

```
NodeType NodeTypeManager.getNodeType(String nodeTypeName)
```

returns the `NodeType` object representing the specified registered node type. `NodeTypeManager` also provides the following related methods for accessing registered node types:

```
boolean NodeTypeManager.hasNodeType(String nodeTypeName)

NodeTypeIterator NodeTypeManager.getPrimaryNodeTypes()

NodeTypeIterator NodeTypeManager.getMixinNodeTypes()

NodeTypeIterator NodeTypeManager.getAllNodeTypes()
```

8.2 NodeType Object

The `NodeType` interface is a subclass of `NodeTypeDefinition`, which provides access methods to the static definitional characteristics of a node type.

`NodeType` adds methods relevant to a "live" node type that is registered in a repository.

Repositories that support *node type management* must implement `NodeTypeTemplate`, which is another subclass of `NodeTypeDefinition` (see §19 *Node Type Management*).

The `NodeType` interface provides methods to access the attributes of a node type:

8.2.1 Name

```
String NodeTypeDefinition.getName()
```

returns the name of the node type (see §3.7.1.1 *Node Type Name*).

8.2.2 Supertypes and Subtypes

```
String[] NodeTypeDefinition.getDeclaredSupertypeNames()
```

returns the list of the names of declared supertypes in this definition (see §3.7.1.2 *Supertypes*).

In a repository that supports *node type management* `NodeTypeDefinition` objects not bound to a live node type may be encountered (for example, in the form of a `NodeTypeTemplate`). In such cases this method may return `null`.

`NodeType` additionally provides the following methods for accessing supertype and subtype information

```
NodeType[] NodeType.getDeclaredSupertypes()

NodeType[] NodeType.getSuperTypes()

boolean NodeType.isNodeType(String nodeTypeName)

NodeTypeIterator NodeType.getDeclaredSubtypes()

NodeTypeIterator NodeType.getSubtypes()
```

8.2.3 Abstract

```
boolean NodeTypeDefinition.isAbstract()
```

returns `true` if the node type is abstract and `false` otherwise (see §3.7.1.3 *Abstract*).

8.2.4 Mixin

```
boolean NodeTypeDefinition.isMixin()
```

returns `true` if the node type is a mixin and `false` if it is a primary type (see §3.7.1.4 *Mixin*).

8.2.5 Queryable Node Type

```
boolean NodeTypeDefinition.isQueryable()
```

returns `true` if the node type is queryable and `false` otherwise (see §3.7.1.5 *Queryable Node Type*).

8.2.6 Orderable Child Nodes

```
boolean NodeTypeDefinition.hasOrderableChildNodes()
```

returns `true` if the node type supports orderable child nodes and `false` otherwise (see §3.7.1.6 *Orderable Child Nodes*). Support for *orderable child nodes* is optional (see §23 *Orderable Child Nodes*).

8.2.7 Primary Item

```
String NodeTypeDefinition.getPrimaryItemName()
```

returns the primary item of the node type, if any (see §3.7.1.7 *Primary Item*).

8.2.8 Property Definitions

The set of property definitions is represented by an array of `PropertyDefinition` objects, accessed through the following methods:

```
PropertyDefinition[]
    NodeTypeDefinition.getDeclaredPropertyDefinitions()

PropertyDefinition[] NodeType.getPropertyDefinitions()
```

(see §3.7.1.8 *Property Definitions*)

8.2.9 Child Node Definitions

The set of child node definitions is represented by an array of `NodeDefinition` objects, accessed through the following methods:

```
NodeDefinition[]
    NodeTypeDefinition.getDeclaredChildNodeDefinitions()

NodeDefinition[] NodeType.getChildNodeDefinitions()
```

(see §3.7.1.8 *Property Definitions*)

8.3 ItemDefinition Object

The attributes common to both property and child node definitions are accessed through the `ItemDefinition` interface. Attributes specific to property definitions or child node definitions are accessed through the `PropertyDefinition` and `NodeDefinition` interfaces, respectively. These interfaces are both subclasses of `ItemDefinition`. The `ItemDefinition` interface provides methods to access the following attributes:

8.3.1 Name

```
String ItemDefinition.getName()
```

returns the JCR Name (in qualified form) of the item to which the definition applies or "*", indicating that the definition is residual (see §3.7.2.1 *Item Definition Name*).

8.3.2 Protected

```
boolean ItemDefinition.isProtected()
```

returns `true` if the item is protected and `false` otherwise (see §3.7.2.2 *Protected*).

8.3.3 Auto-Created

```
boolean ItemDefinition.isAutoCreated()
```

returns `true` if the item is auto-created and `false` otherwise (see §3.7.2.3 *Auto-Created*).

8.3.4 Mandatory

```
boolean ItemDefinition.isMandatory()
```

returns `true` if the item is mandatory and `false` otherwise (see §3.7.2.4 *Mandatory*).

8.3.5 On-Parent-Version

```
int ItemDefinition.getOnParentVersion()
```

returns the on-parent-version setting of the definition; one of the constants of `OnParentVersionAction` (see §3.7.2.5 *On-Parent-Version*).

8.3.6 Declaring Node Type

```
NodeType ItemDefinition.getDeclaringNodeType()
```

returns the `NodeType` object that contains this definition (see §8.2 *NodeType Object*).

8.4 PropertyDefinition Object

The attributes specific to property definitions are accessed through the `PropertyDefinition` interface, which is a subclass of `ItemDefinition`:

8.4.1 Required Type

```
int PropertyDefinition.getRequiredType()
```

returns the property type setting of the definition, which must be one of the constants of the `PropertyType` interface (see §3.7.3.1 *Property Type*).

8.4.2 Default Values

```
Value[] PropertyDefinition.getDefaultValues()
```

returns the default values of the definition (see §3.7.3.2 *Default Values*).

8.4.3 Available Query Operators

```
String[] PropertyDefinition.getAvailableQueryOperators()
```

returns an array of `String` constants indicating which query operators are supported for this property (see §3.7.3.3 *Available Query Operators*). The constants are defined in the class `QueryObjectModelConstants` and represent the operators defined in §6.7.16 *Comparison*.

8.4.4 Full-Text Searchable

```
boolean NodeTypeDefinition.isFullTextSearchable()
```

returns `true` if the property is full-text searchable and `false` otherwise (see §3.7.3.4 *Full-Text Searchable*).

8.4.5 Query-Orderable

```
boolean NodeTypeDefinition.isQueryOrderable()
```

returns `true` if the property is query-orderable and `false` otherwise (see §3.7.3.5 *Query-Orderable*).

8.4.6 Value Constraints

```
String[] PropertyDefinition.getValueConstraints()
```

returns the value constraints of the definition (see §3.7.3.6 *Value Constraints*),

8.4.7 Multi-value

```
boolean PropertyDefinition.isMultiple()
```

returns `true` if the definition defines a multi-value property and `false` if it defines a single value property (see §3.7.3.7 *Multi-Value*).

8.5 NodeDefinition Object

The attributes specific to child node definitions are accessed through the `NodeDefinition` interface, which is a subclass of `ItemDefinition`:

8.5.1 Required Primary Node Types

The methods

```
NodeType[] NodeDefinition.getRequiredPrimaryTypes()
```

and

```
String[] NodeDefinition.getRequiredPrimaryTypeNames()
```

return information about the required primary node types of the definition (§3.7.4.1 *Required Primary Node Types*). The latter method returns the names of the node types while the former method returns the live `NodeType` objects representing the types. The former only functions if the `NodeDefinition` is part of a live registered `NodeType`.

8.5.2 Default Primary Node Type

The methods

```
NodeType NodeDefinition.getDefaultPrimaryType()
```

and

```
String NodeDefinition.getDefaultPrimaryTypeName()
```

return information about the default primary node type of the definition (§3.7.4.2 *Default Primary Node Type*). The latter method returns the name of the node type while the former method returns the live `NodeType` object representing the type. The former only functions if the `NodeDefinition` is part of a live registered `NodeType`.

8.5.3 Same-Name Siblings

```
boolean NodeDefinition.allowsSameNameSiblings()
```

returns `true` if the definition allows same-name sibling nodes and `false` otherwise (see §3.7.4.3 *Same-Name Siblings*).

8.6 Node Type Information for Existing Nodes

Given an existing `Node`, the methods

```
NodeType Node.getPrimaryNodeType() and

NodeType[] Node.getMixinNodeTypes()
```

return, respectively, the primary and mixin node types of the node. The method

```
boolean Node.isNodeType(String nodeTypeName)
```

returns `true` if the `Node` is of the specified node type, according to the *is-of-type* relation (see §3.7.6.3 *Is-of-Type Relation*), and `false` otherwise.

8.6.1.1 Discovery of Item Definitions

The `Node` and `Property` interfaces offer methods that allow direct access to the `NodeDefinition` or `PropertyDefinition` within the node type of a parent node that is applicable to a particular child item:

```
NodeDefinition Node.getDefinition()

PropertyDefinition Property.getDefinition()
```

The definition that applies to an item is determined upon creation of that item (see §3.7.7 *Applicable Item Definition*).

8.6.1.2 Root Node Definition

The method `getDefinition` called on the root node must return a valid, non-`null`, `NodeDefinition` object. The values returned by the methods of this object must be as follows:

- `getName()`: "", the empty string.
- `getDeclaringNodeType()`: A valid `NodeType` object (see §8.6.1.3 *Root Declaring Node Type*).
- `isMandatory()`: `true`
- `isAutoCreated()`: `true`
- `isProtected()`: `false`
- `allowsSameNameSiblings()`: `false`
- `getOnParentVersion()`: `VERSION`, if versioning is supported and the root node is capable of being made versionable, `IGNORE` otherwise.
- `getDefaultPrimaryType()`: A valid non-`null` `NodeType` object (see §3.7.8 *Root Node Type*).

- `getRequiredPrimaryTypes()`: An array containing a single `NodeType` object identical with that returned by `getDefaultPrimaryType`.

8.6.1.3 Root Declaring Node Type

Calling `getDeclaringNodeType()` on the `NodeDefinition` of the root node must return a valid `NodeType` object. The values returned by the methods of this object must be as follows:

- `getName()` returns the name of a node type `N`, where `N` is implementation-determined.
- `isNodeType(String nodeTypeName)` returns `true` if an only if `nodeTypeName` is `N` or a supertype of `N`.
- `getChildNodeDefinitions()` and `getDeclaredChildNodeDefinitions()` both return an array containing the child node definition of the root node.

All other methods either return `false` (if they return a `boolean`) or an empty array (if they return an array).

9 Permissions and Capabilities

9.1 Permissions

Permissions encompass the restrictions imposed by any access control restrictions that may be in effect upon the content of a repository, either implementation specific or JCR-defined (see §16 *Access Control Management*).

In repositories that support *Access Control* this will include the restrictions governed by privileges but may also include any additional policy-internal refinements with effects too fine-grained to be exposed through privilege discovery (see §16.2 *Privilege Discovery*).

Permissions are reported through

```
boolean Session.hasPermission(String absPath, String actions)
```

which returns `true` if this `Session` has permission to perform all of the specified `actions` at the specified `absPath` and returns `false` otherwise. Similarly,

```
void Session.checkPermission(String absPath, String actions)
```

throws an `AccessDeniedException` if the this `Session` does not have permission to perform the specified `actions` and returns quietly if it does.

The `actions` parameter is a comma separated list of action strings, of which there are four, defined as follows:

`add_node`: The permission to add a node at `absPath`.

`set_property`: The permission to set (add or change) a property at `absPath`.

`remove`: The permission to remove an item at `absPath`.

`read`: The permission to retrieve (and read the value of, in the case of a property) an item at absPath.

The permission actions `add_node`, `set_property` and `remove` will only be relevant in a *writable repository*. In a read-only repository they will always return `false`.

The information returned through these methods only reflects access control-related restrictions, not other kinds of restrictions such as node type constraints. For example, even though `hasPermission` may indicate that a particular `Session` may add a property at `/A/B/C`, the node type of the node at `/A/B` may prevent the addition of a property called `C`.

Methods for testing restrictions more broadly are provided by the *capabilities* feature (see §9.2 *Capabilities*). For information on the relationships among *permissions*, *privileges* and *capabilities*, see §16.6 *Privileges Permissions and Capabilities*.

9.2 Capabilities

Capabilities encompass the restrictions imposed by permissions, but also include any further restrictions unrelated to access control. The method

```
boolean Session.hasCapability(String methodName,
                              Object target,
                              Object[] arguments)
```

checks whether an operation can be performed given as much context as can be determined by the repository, including:

- Permissions granted to the current user, including access control privileges.
- Current state of the target object (reflecting locks, checked-out status, retention and hold status etc.).
- Repository capabilities.
- Node type-enforced restrictions.
- Repository configuration-specific restrictions.

The implementation of this method is best effort: returning `false` guarantees that the operation cannot be performed, but returning `true` does not guarantee the opposite.

The `methodName` parameter identifies the method in question by its name as defined in the Javadoc.

The `target` parameter identifies the object on which the specified method is called.

The `arguments` parameter contains an array of type `Object` consisting of the arguments to be passed to the method in question. In cases where a parameter is a Java primitive type it must be converted to its corresponding Java object form.

For example, given a `Session S` and `Node N` then

```
boolean b = S.hasCapability("addNode", N, new Object[]{"foo"});
```

will result in `b == false` if a child node called `foo` cannot be added to the node `N` within the session `S`.

10 Writing

A repository may be *writable*.

Whether an implementation supports writing can be determined by querying the repository descriptor table with

```
Repository.WRITE_SUPPORTED.
```

A return value of `true` indicates support (see §24.2 *Repository Descriptors*).

10.1 Types of Write Methods

A JCR write method is either a *session-write* or a *workspace-write*.

10.1.1 Session-Write

Changes made through a session-write are buffered in a transient store associated with that method's current session (see §3.1.8.2 *Current Session and Workspace*). The transient store permits a series of changes to be made without validation at every step, thus allowing item structures to be temporarily invalid while they are being constructed. Once completed, the change set can be *saved*.

- Before save, a change in transient store is *pending*.
- Upon save, all changes in transient store are *dispatched*.

10.1.2 Workspace-Write

- A change made through a workspace-write is immediately *dispatched*.

10.1.3 Transactions

- In the absence of a transaction, every *dispatched* change is immediately *persisted*.
- Within a transaction, dispatched changes are persisted upon *commit*.

(see §21 *Transactions*).

10.1.4 Visibility of Changes

A change that is *pending* or *dispatched* (but not *persisted*) is visible only to the session through which that change was made. A change that is *persisted* is visible to all other sessions bound to the same persistent workspace that have sufficient read permission.

10.1.5 Write Methods

The write API is divided into the two types as follows.

10.1.5.1 Session-Write

The session-write methods are

- `Node.addNode`, `setProperty` and `orderBefore`.
- `Property.setValue`.
- `Item.remove`.
- `Node.removeShare`, and `removeSharedSet`.
- `Session.move`, `removeItem` and `importXML`.
- `Query.storeAsNode`.
- `Node.setPrimaryType`, `addMixin` and `removeMixin`.
- `RetentionManager.addHold`, `removeHold`, `setRetentionPolicy` and `removeRetentionPolicy`.
- `AccessControlManager.setPolicy` and `removePolicy`.

10.1.5.2 Workspace-Write Methods

The workspace-write methods are:

- `Workspace.move`, `copy`, `clone`, `restore` and `importXML`.
- `VersionManager.checkin`, `checkout`, `checkpoint`, `restore`, `restoreByLabel`, `merge`, `cancelMerge`, `doneMerge`, `createActivity` , `removeActivity` and `createConfiguration`.
- `Node.update` and `followLifecycleTransition`.
- `LockManager.lock` and `unlock`.
- `VersionHistory.addVersionLabel`, `removeVersionLabel` and `removeVersion`.
- `Session.save`.
- Methods of `org.xml.sax.ContentHandler` acquired through `Workspace.getContentHandler`.
- `Workspace.createWorkspace` and `deleteWorkspace` (these create or delete another workspace, though they do not affect *this* workspace).

10.1.5.3 Optional In-Content Side-Effects

Some repositories may choose to expose internal state data as virtual content within a workspace. For example, the set of registered node types may be so exposed.

In such cases, methods which directly affect the exposed internal state and, as a side effect, change virtual content must do so in a workspace-write manner. For example, `NodeTypeManager.registerNodeType` and `unregisterNodeType` should immediately dispatch changes to the in-content node type representation.

10.2 Core Write Methods

The *core write methods* of JCR are those write methods of the API whose write effect is *not* incidental to the support of another feature, such as versioning, import, locking, and so forth. Both session-write and workspace-write methods are among the core write methods. The core write methods are:

- `Node.addNode`, `setProperty` and `orderBefore`.
- `Property.setValue`.
- `Item.remove`.
- `Node.removeShare`, and `removeSharedSet`.
- `Session.move`, and `removeItem`.
- `Workspace.move`, `copy`, `clone`.
- `Session.save`.

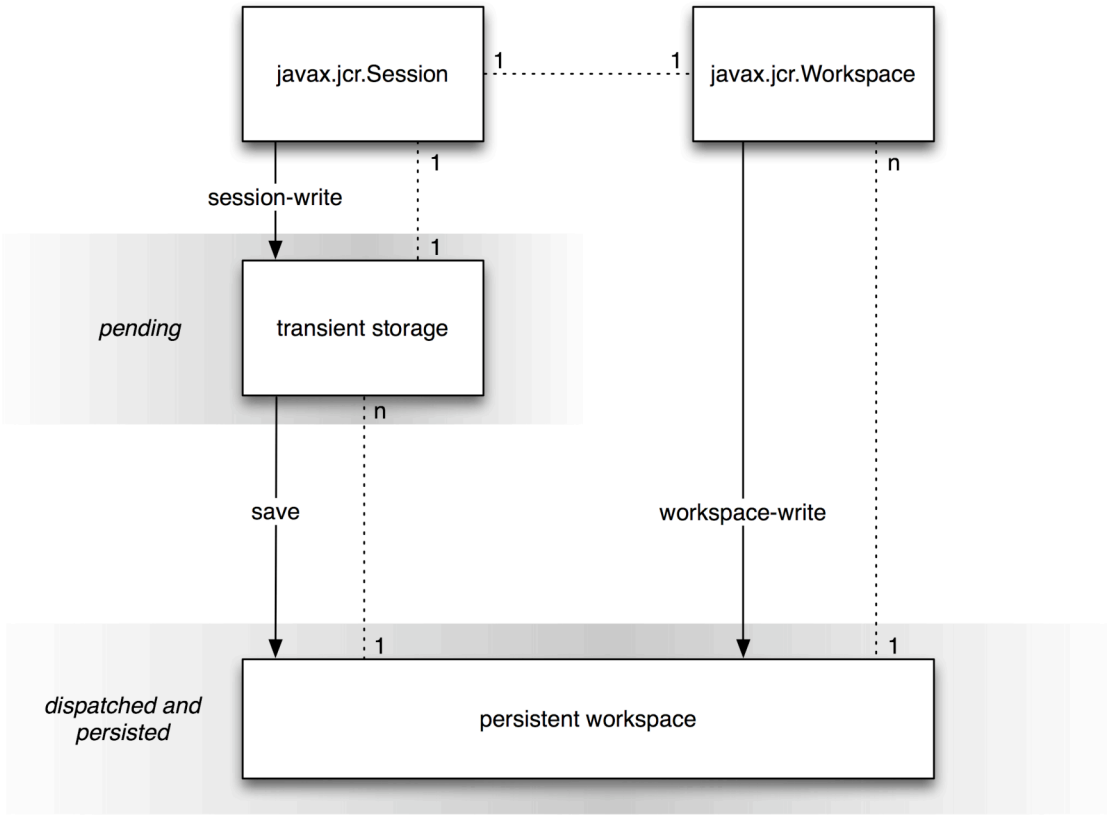
10.3 Session and Workspace Objects

Given the set $S_0..S_n$ of `Session` objects bound to a persistent workspace W^P, for each S_i, there exists a distinct `Workspace` object W_i, bound one-to-one to S_i, that represents the *view* of W^P through the access permissions of S_i.

Despite their one-to-one correspondence, `Session` and `Workspace` are defined as separate objects in order to differentiate the behavior of session-write methods from the behavior of workspace-write methods.

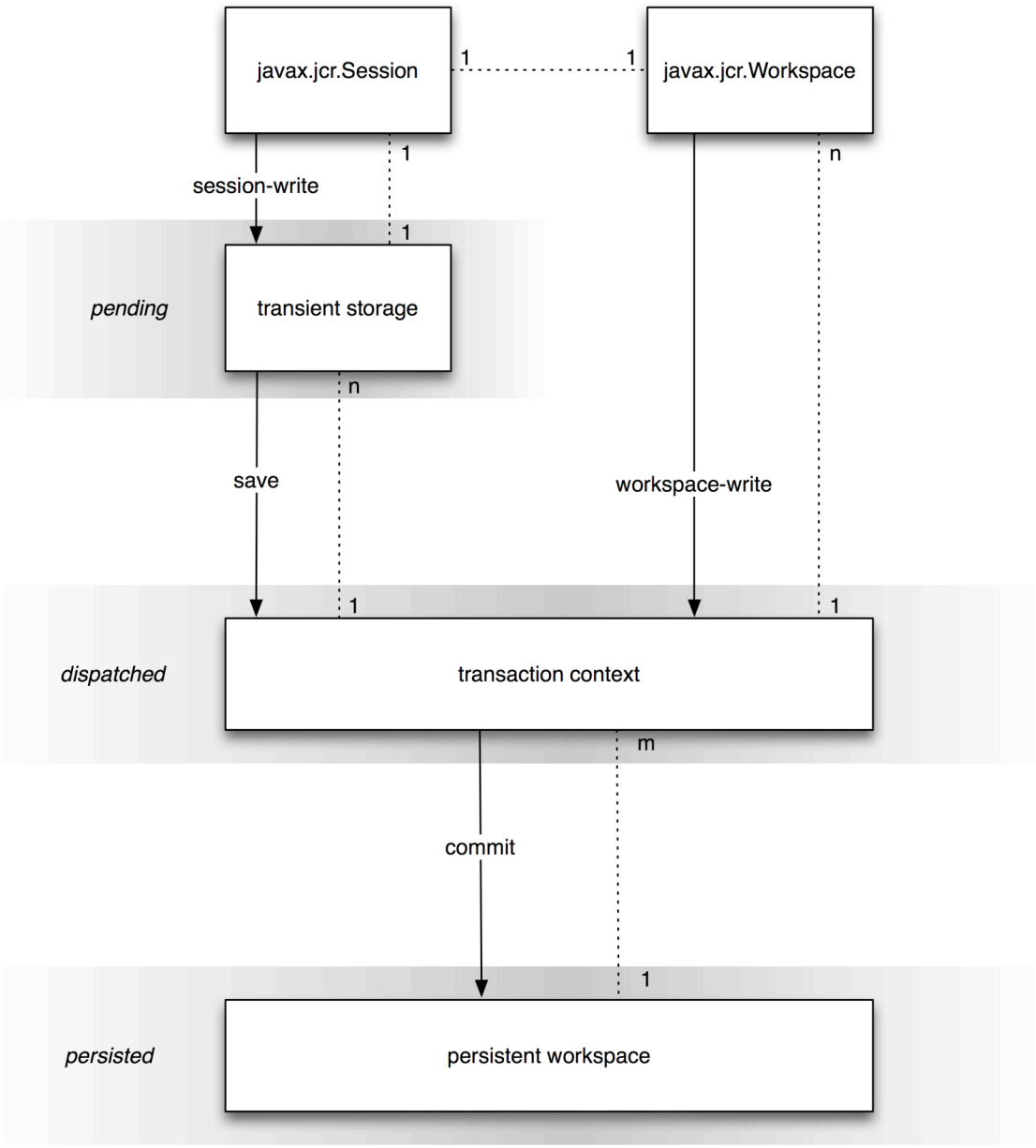
10.3.1 Writing Without a Transaction

The following diagram depicts the behavior of write methods *without a transaction*.


javax.jcr.Session
1
1
javax.jcr.Workspace
1
n
session-write
1
pending
transient storage
n
save
workspace-write
1
1
dispatched and persisted
persistent workspace

10.3.2 Writing Within a Transaction

The following diagrams depicts the behavior of write methods *within a transaction*.



10.4 Adding Nodes and Setting Properties

This section covers the JCR API methods for adding new nodes and properties and changing the values of existing properties.

10.4.1 Adding a Node

The methods

```
Node Node.addNode(String relPath, String primaryNodeTypeName)
```

and

```
Node Node.addNode(String relPath)
```

add a node at the specified location relative to this `Node`. The former specifies the intended primary node type of the node, while the latter assumes that the repository can determine the intended type from the node type of the parent.

`Node.addNode` is a *session-write* method and therefore requires a `Session.save` to dispatch the change (see §10.11 *Saving*).

10.4.2 Setting a Property

The generic method for setting a property is

```
Property Node.setProperty(String name, Value value, int type).
```

This method sets the property of this `Node` with the specified `name` to the specified value and the specified type, converting the given value to that type if necessary. If the property already exists its value is changed. If it does not exist, it is added.

`Node.setProperty` is a *session-write* method and therefore requires a `Session.save` to dispatch the change (see §10.11 *Saving*).

10.4.2.1 Changing Existing Properties

An existing property can also be changed with

```
void Property.setValue(Value value).
```

`Property.setValue` is a *session-write* method and therefore requires a `Session.save` to dispatch the change (see §10.11 *Saving*).

10.4.2.2 Type-Specific Signatures

Signatures of `Node.setProperty` and `Property.setValue` are also provided in which the intended JCR type is implied by the Java type passed in. For example,

```
Node.setProperty(String name, Calendar value)
```

sets a JCR `DATE` property called `name` to the specified `value`. See the Javadoc for the full set of signatures.

10.4.2.3 Setting a DECIMAL Property

When setting a property of type `DECIMAL` using

```
Node.setProperty(String name, BigDecimal value)
```

or

```
Property.setValue(BigDecimal value)
```

the `java.math.BigDecimal` object passed must be an instance of the actual class `BigDecimal`, not an instance of a subclass.

10.4.2.4 No Null Values

Every property must have a value. The range of property states does not include having a "null value", or "no value". Setting a property to "null" is equivalent to removing that property (see §10.9 *Removing Nodes and Properties*).

10.4.2.5 Multi-value Properties and Null

As with single-value properties, there is no such thing as a null value. If a value within a multi-value property is set to null, this is equivalent to removing that value from the value array. In such a case the array is automatically compacted, shifting the indices of those values with an index greater than that of the removed value by -1. However, while no value within a multi-value property can be null, a multi-value property can exist with no values (i.e., it can be an empty array).

10.4.2.6 Setting Multi-value vs. Single-value Properties

Multi-value and single-value properties are set using different signatures of `Node.setProperty` and `Property.setValue`. Multi-value properties must be set using the signatures that take either a `Value[]` or `String[]`. Single-value properties must be set using the signatures that take non-array value arguments. An attempt to set a multi-value property with a non-array value argument, or a single-value property with an array value argument, will throw a `ValueFormatException`.

10.4.3 Creating Value Objects

In many cases a property must be set using a `Value` object. `Value` objects are created using a `ValueFactory`, acquired through

```
ValueFactory Session.getValueFactory().
```

The generic `Value` creation method is

```
Value ValueFactory.createValue(String value, int type)
```

which takes the string-form of the specified type and returns a `Value` of that type using standard property type conversion (see §3.6.4 *Property Type Conversion*).

10.4.3.1 Type-Specific Methods

`ValueFactory` also provides methods for creating values of each property type from the corresponding Java type. See the Javadoc for the full set of signatures.

10.4.3.2 Creating a BINARY Value

To create a `BINARY` value a `javax.jcr.Binary` object is first created from a stream using

```
Binary ValueFactory.createBinary(InputStream stream)
```

and then passed to

```
Value ValueFactory.createValue(Binary value).
```

10.5 Selecting the Applicable Item Definition

An `addNode` or `setProperty` method must determine which, if any, item definitions of the parent node apply to the new child item, based on the name of the new item and, if provided, its type.

If more than one item definition still applies even after taking the name and type constraints into consideration, the repository may either fail the add attempt or automatically select one of the item definitions based on implementation-specific criteria.

For example, if the parent node `P` has two residual child node definitions that differ only by their OPV value (see §3.7.2.5 *On-Parent-Version*), then even if both a name and a primary type are supplied in the call to `addNode` this will not be sufficient information to unambiguously determine which residual definition the new node should fall under. In such a case, an implementation might automatically select one of the definitions based on the implementation-specific rule that a node with the name *X* will always have an OPV of *V* while other nodes will have an OPV of *W*.

When `Node.setProperty` is used to change the value of an existing property, cases where the intended property is ambiguous are handled in the same way as when the method is used to create a new property.

10.6 Moving Nodes

The method

```
void Session.move(String srcAbsPath, String destAbsPath)
```

moves the subgraph at `srcAbsPath` to a new location at `destAbsPath`. This is a session-write operation (see §10.1.1 *Session-Write*). The method

```
void Workspace.move(String srcAbsPath, String destAbsPath)
```

does the same, but is a workspace-write operation (see §10.1.1 *Workspace-Write*).

10.6.1 Referenceable vs Non-Referenceable Nodes

A referenceable node is guaranteed to maintain the same identifier across a `move` operation.

Non-referenceable nodes, on the other hand, *may* be tied either partially or entirely (as in the case where the identifier equals the path) to their position in the hierarchy and therefore may change identifier upon `move`.

Though nothing prevents an implementation from making non-referenceable node identifiers as stable as referenceable node identifiers, a user cannot rely upon this across repository vendors. For an overview of how identifiers behave with different methods see §25.1 *Treatment of Identifiers.*

10.7 Copying Nodes

Nodes can be copied from one path location to another within a workspace and, in repositories with more than one workspace, across workspaces (see §3.10 *Multiple Workspaces and Corresponding Nodes*). A copy operation on a node copies the node and its subgraph. Properties cannot be copied individually.

10.7.1 Copying Within a Workspace

The method

```
void Workspace.copy(String srcAbsPath, String destAbsPath)
```

copies the node at `srcAbsPath` and its subgraph to a new location at `destAbsPath`. This is a workspace-write operation (see §10.1.1 *Workspace-Write*).

10.7.2 Copying Across Workspaces

In a repository with more than one workspace, the method

```
void Workspace.copy(String srcWorkspace,
                    String srcAbsPath,
                    String destAbsPath)
```

copies the node at `srcAbsPath` in `srcWorkspace` and its subgraph to a new location at `destAbsPath` in the current workspace. This is a workspace-write operation (see §10.1.2 *Workspace-Write*).

10.7.3 Copying to an Empty Location

When a node `N` is copied to a path location where no node currently exists, a new node `N'` is created at that location. The subgraph rooted at and including `N'` (call it `S'`) is created and is identical to the subgraph rooted at and including `N` (call it `S`) with the following exceptions:

- If the copy is within the same workspace, every node in `S'` is given a new and distinct identifier. If the copy is to another workspace, every referenceable node in `S'` is given a new and distinct identifier while every non-referenceable node in `S'` *may* be given a new and distinct identifier (see §3.8 *Referenceable Nodes*).
- The repository *may* automatically drop any *mixin node type* `T` present on any node `M` in `S`. Dropping a mixin node type in this context means that while `M` remains unchanged, its copy `M'` will lack the mixin `T` and any child nodes and properties defined by `T` that are present on `M`. For example, a node `M` that is `mix:versionable` may be copied such that the resulting node `M'` will be a copy of `N` except that `M'` will not be `mix:versionable` and will not have any of the properties defined by `mix:versionable`. In order for a mixin node type to be dropped it must be listed by name in the `jcr:mixinTypes` property of `M`. The resulting `jcr:mixinTypes` property of `M'` will reflect any change.

- If a node `M` in `S` is referenceable and its `mix:referenceable` mixin is *not* dropped on copy, then the resulting `jcr:uuid` property of `M'` will reflect the new identifier assigned to `M'`.
- Each `REFERENCE` or `WEAKEREFERENCE` property `R` in `S` is copied to its new location `R'` in `S'`. If `R` references a node `M` *within* `S` then the value of `R'` will be the identifier of `M'`, the new copy of `M`, thus preserving the reference within the subgraph (see §3.8 *Referenceable Nodes*).

10.7.4 Copying to an Existing Node

When a node `N` is copied to a location where a node `N'` already exists the repository may either immediately throw an `ItemExistsException` *or* attempt to update the node `N'` by selectively replacing part of its subgraph with a copy of the relevant part of the subgraph of `N`. If the node types of `N` and `N'` are compatible, the implementation supports update-on-copy for these node types and no other errors occur, then the copy will succeed. Otherwise an `ItemExistsException` is thrown.

Which node types can be updated on copy and the details of any such updates are implementation-dependent. For example, some implementations may support update-on-copy for mix:versionable nodes. In such a case the versioning-related properties of the target node would remain unchanged (`jcr:uuid`, `jcr:versionHistory`, etc.) while the substantive content part of the subgraph would be replaced with that of the source node.

10.8 Cloning and Updating Nodes

A node can be *cloned* to another workspaces to create a new corresponding node (see §3.10 *Corresponding Nodes*)

10.8.1 Cloning Nodes Across Workspaces

Corresponding nodes can be created by *cloning* a node from one workspace to another using:

```
void Workspace.clone(String srcWorkspace,
                     String srcAbsPath,
                     String destAbsPath,
                     boolean removeExisting)
```

This method clones the subgraph at `srcAbsPath` in `srcWorkspace` to `destAbsPath` in `this` workspace. The `clone` method clones both referenceable and non-referenceable nodes and preserves the identifier of every node in the source subgraph.

If there already exists anywhere in this workspace a node with the same identifier as an incoming node from `srcWorkspace`, and `removeExisting` is `false`, then `clone` will throw an `ItemExistsException`.

If `removeExisting` is `true` then the existing node is removed from its current location and the cloned node with the same identifier from `srcWorkspace` is copied to this workspace as part of the copied subgraph (that is, not into the

former location of the old node). The subgraph of the cloned node will reflect the state of the clone in `srcWorkspace`, in other words the existing node will be moved *and* changed. If the existing node cannot be moved and changed because of node type constraints, access control constraints or because its parent is checked-in (or its parent is non-versionable but its nearest versionable ancestor is checked-in), then the appropriate exception is thrown (`ConstraintViolationException`, `AccessControlException` or `VersionException`, respectively).

In the case of shareable nodes, it is possible to clone a node into its own workspace (see §14.1 *Creation of Shared Nodes*).

10.8.2 Getting a Corresponding Node

Finding the path of a node's corresponding node in another workspace is done with

```
String Node.getCorrespondingNodePath(String workspaceName).
```

This method returns the absolute path of the node in the specified workspace that corresponds to this node.

10.8.3 Updating Nodes Across Workspaces

Node correspondence governs the behavior of the `update` method:

```
void Node.update(String srcWorkspace)
```

causes `this` node to be updated to reflect the state of its corresponding node in `srcWorkspace`.

If this node does have a corresponding node in the workspace `srcWorkspace`, then this replaces this node and its subgraph with a clone of the corresponding node and its subgraph.

If this node does not have a corresponding node in `srcWorkspace`, then the method has no effect.

If the `update` succeeds, the changes made to this node and its subgraph are applied to the workspace immediately, there is no need to call `save`.

The `update` method does not respect the checked-in status of nodes. An `update` may change a node even if it is currently checked-in.

`Node.update` works for both versionable and non-versionable nodes (see §3.13 *Versioning Model*)

10.9 Removing Nodes and Properties

Removing a node or property can be done with

```
void Item.remove()
```

On the item to be removed itself, or

```
void Session.removeItem(String absPath)
```

Where `absPath` specifies the item to be removed.

These methods are session-write and therefore require a `Session.save` to dispatch the change.

10.9.1 Setting a Property to Null

A property can also be removed by setting its value to `null`. When this is done, the `null` parameter must be cast to a non-array type for single-value properties and an array type for multi-value properties.

Note that setting a multi-value property to an array containing `null` values is different from setting it to a `null` cast to an array type. In the former case, all `null` values within the array are removed and the array is compacted to include only non-null values even if this results in a multi-value property being set to an empty array. In the latter case the entire property is removed. For example,

```
p.setValue((String[])null)
```

would remove property `p`, whereas

```
p.setValue(new String[]{“a”, null, “b”})
```

would set `p` to `[“a”,”b”]` and

```
p.setValue(new String[]{null})
```

would set `p` to the empty array, `[]` (see §10.4.2.4 *No Null Values*).

10.9.1.1 Removing a REFERENCE Target

To remove a node that is the target of a `REFERENCE` property, one must first remove that `REFERENCE` property (with the exception of `REFERENCE` properties within the frozen node of a version, see §3.13.4.6 *References in a Frozen Node*).

The check for referential integrity is done *on persist* of the removal. If the subgraph to be removed contains a node that is the target of a `REFERENCE` property outside that subgraph, a `ReferentialIntegrityException` is thrown.

10.10 Node Type Assignment

Most writable repository implementations will support assignment of primary and mixin node types on node creation. Some implementations may also support assignment of new primary or mixin node types to existing nodes.

10.10.1 Node Type Assignment Behavior

On `Node.addNode` the primary node type of the new node is assigned. In cases where a `Node.addNode` does not explicitly specify a primary node type, it is determined by the applicable child node definition (see §3.7.7 *Applicable Item Definition*). Otherwise, it is determined by the node type name passed. The `jcr:primaryType` property is created immediately and set to the name of the primary node type. This property is defined as `mandatory` in the node type `nt:base` (see §3.7.10 *Base Primary Node Type*) and will therefore appear on every node.

The constraints enforced by the assigned node type may take effect immediately, or on persist. Whichever is chosen, this *node type assignment behavior* must be consistent across all methods that assign node types (`Node.setPrimaryType`, `Node.addMixin` and `Node.removeMixin`, see §10.10.2 *Updating a Node's Primary Type* and §10.10.3 *Assigning Mixin Node Types*).

If *immediate effect* is implemented then conflicts with other mixins or with the primary type are detected immediately and an exception thrown. If *on-persist effect* is implemented, such conflicts are detected and the appropriate exception thrown on persist. This validation can also be performed pre-emptively with

```
boolean Node.canAddMixin(String mixinName).
```

10.10.2 Updating a Node's Primary Type

A repository *may* permit the primary type of a node to be changed during its lifetime. Repositories are free to limit the scope of permitted changes both in terms of which nodes may be changed and which changes are allowed.

The method for changing the primary type of a particular node is

```
void Node.setPrimaryType(String nodeTypeName).
```

This method changes the primary node type of the node to `nodeTypeName`, and immediately changes the `jcr:primaryType` property of the node appropriately.

Semantically, the new node type takes effect in accordance with the *node type assignment behavior* of the repository (see §10.10.1 *Node Type Assignment Behavior*).

10.10.3 Assigning Mixin Node Types

In addition to its single primary node type, a node may have one or more mixin node types assigned to it (see §3.7.5 *Mixin Node Types*). Assignment of mixin types is done through

```
void Node.addMixin(String mixinName).
```

A repository that supports the assignment of mixin types may permit mixin addition only before the first save of a node (in effect, only on node creation) or it may permit mixin addition and removal during the lifetime of a node. Removal of mixin node types is done with

```
void Node.removeMixin(String mixinName).
```

10.10.3.1 jcr:mixinTypes

When a new mixin type is assigned using `Node.addMixin`, the name of the mixin is added immediately to the multi-valued `jcr:mixinTypes` property. If the property does not exist, it is created. This property is defined as non-mandatory in the node type `nt:base` and therefore may appear on any node. When a mixin is removed with `Node.removeMixin` the name of the mixin type is immediately removed from the property.

Semantically, any changes to mixin node types take effect in accordance with the *node type assignment behavior* of the repository (see §10.10.1 *Node Type Assignment Behavior*).

10.10.3.2 Pre-emptive Node Type Validation

A `NodeType` object can be queried to pre-emptively determine whether a particular child item's addition or removal is allowed by that node type. The methods are:

```
boolean NodeType.canSetProperty(String propertyName,
                                Value value)

boolean NodeType.canSetProperty(String propertyName,
                                Value[] values)

boolean NodeType.canAddChildNode(String childNodeName)

boolean NodeType.canAddChildNode(String childNodeName,
                                 String nodeTypeName)

boolean NodeType.canRemoveNode(String nodeName)

boolean NodeType.canRemoveProperty(String propertyName)
```

10.10.3.3 Automatic Addition and Removal of Mixins

A repository may automatically assign a mixin type to a node upon creation. For example if, as a matter of configuration, all `nt:file` nodes in a repository are to be versionable, then the repository may automatically assign the mixin type `mix:versionable` to each such node as it is created.

Similarly, a repository may automatically strip incoming imported nodes of any mixin node types that the repository does not support (see §11.3 *Respecting Property Semantics*).

Note that this behavior is distinct from that of adding a mixin type as a supertype of some primary types in the node type inheritance hierarchy (see §3.7.16.1.2 *Additions to the Hierarchy*). Though the two features may both be employed in the same repository, they differ in that one affects the actual hierarchy of the supported node types, while the other works on a node-by-node basis.

10.11 Saving

When a change is made to an item through a session-write method bound to a session *S*, that change is immediately visible through all subsequent read method calls through *S*. When

```
void Session.save()
```

is performed on *S*, all pending changes recorded in *S* are dispatched. Without transactions this causes the changes to be persisted. Within a transaction the changes must first be committed in order to be persisted. When a change is persisted it becomes visible to other sessions bound to the same persistent workspace.

From the point of view of a session *S*, the apparent state of an `Item` bound to *S* does not change upon a `save` of *S* (apart from the values returned by `isNew` or `isModified`, see §10.11.3 *Item Status*) since that state will have been visible *to S* since the session-write method call that caused it.

If one or more of the pending changes cause an exception to be thrown on `save`, then *no* pending changes are dispatched and the set of pending changes recorded on the session is left unaffected.

10.11.1 Refresh

The method

```
void Session.refresh(boolean keepChanges)
```

refreshes the state of the transient session store.

If `keepChanges` is `false`, all pending changes in the session are discarded and all items bound to that session revert to their current dispatched state. Without transactions, this is the current persisted state. Within a transaction, this state will reflect persistent storage as modified by changes that have been saved but not yet committed.

If `keepChanges` is `true` then pending changes are not discarded but items that do not have changes pending have their state refreshed to reflect the current persisted state, thus revealing changes made by other sessions.

If an exception occurs on `refresh`, the set of all pending changes recorded on the session is left unaffected and the state of all bound `Item`s is also unaffected.

10.11.2 Session Status

The method

```
boolean Session.hasPendingChanges()
```

is used to determine if a session holds pending changes.

10.11.3 Item Status

Whether an `Item` has pending changes can be determined with

```
boolean Item.isModified().
```

Whether an `Item` constitutes part of the pending changes of its parent can be determined with

```
boolean Item.isNew().
```

10.11.4 Persisting by Identifier

When a change to an item is persisted, the item in the persistent workspace to which that pending change is written is determined as follows:

- If the changed `Item` is a `Node` with identifier *I*, then the changes are written to the persistent node with identifier *I*.

- If the changed `Item` is a `Property` named *P* of a `Node` with identifier *I*, then the change is written to the persistent property *P* of the persistent node with identifier *I*.

These principles apply to both referenceable and non-referenceable nodes (see §3.8 *Referenceable Nodes*). For an overview of how identifiers behave with different methods see §25.1 *Treatment of Identifiers*.

10.11.5 Timing of Validation

For session-write methods, implementers have flexibility in deciding whether a particular validation is to be performed immediately on invocation of the write method or later on persist. For example, in the case `Node.addNode`, an implementer might immediately check that the path given is valid while postponing validation of node type constraints until persist-time.

- It is *suggested* that an implementation perform each validation as soon as possible, given the underlying design of the repository.
- It is *required* that an implementation prohibit the emergence of a persistent state in violation of the validation rules defined by this specification.

10.11.6 Invalid States

If an item has been modified in the `Session` but not yet persisted, and its corresponding item in the persistent workspace is altered through a workspace-write method, this has no effect on the transient state of the `Session`. The altered item in the `Session` remains and may be persisted later. However, the change made to the workspace *may* render the attempt to persist the session-change invalid (for example, if the workspace-change removed the parent of the session-change item). Note that this is precisely the same situation as would arise if a change were made to a workspace through *another* `Session`. In both cases the persist of the change may throw an `InvalidItemStateException`.

10.11.7 Reflecting Item State

When changes are made to an `Item` object, those changes are recorded in its bound `Session` and immediately reflected in the `Item` object itself. A subsequent re-retrieval of the same item entity through a method bound to the same `Session`, will return an `Item` object reflecting the recent change. Note that this includes acquisition of nodes and properties through standard getter methods such as `getNode` and also retrieval through other means, such as a query (see §6 *Query*).

Whether the second `Item` object is the same actual Java object instance as the first is an implementation issue. However, the state reflected by the object must at all times be consistent with any other `Item` object bound to the same `Session` that represents the same actual item entity. The criteria of item identity in this context are those described in §10.11.4 *Persisting by Identifier*.

10.11.8 Invalid Items

An `Item` object may become invalid for a number of reasons.

If `Item.remove` has been called on the item any subsequent calls to any read or write methods or invocations of `save` or `refresh` on that `Item`, from within the same `Session`, will throw an `InvalidItemStateException`. Before the removal is saved it may be cancelled by a `Session.refresh(false)`. At this point the invalid `Item` object may become valid again, or the repository may require a new `Item` object to be acquired. Which approach is taken is a matter of implementation.

An `InvalidItemStateException` *may* be thrown immediately on a write method of an `Item` if the change being made would, upon persist, conflict with a change made and persisted through another `Session`. If detection of the conflict is only possible at persist-time, then an `InvalidItemStateException` will be thrown at that point. Whether a conflict is detected when the change is made or on persist depends on the implementation.

Apart from these specific cases, the validity of an `Item` must be as stable as the *identifiers* used in the repository (see §3.3 *Identifiers*).

10.11.9 Seeing Changes Made by Other Sessions

Transient storage of pending changes in a `Session` may be implemented a number of ways. A repository is free to use any approach as long as it guarantees that two `Item` objects bound to the same `Session` will never reflect conflicting state information.

10.12 Namespace Registration

A repository has a single namespace registry (see §3.5.1 *Namespace Registry*) represented by the `NamespaceRegistry` object, acquired through

```
NamespaceRegistry Workspace.getNamespaceRegistry().
```

`NamespaceRegistry` allows for persistent changes to namespaces through the following methods.

10.12.1 Registering a Namespace

```
void NamespaceRegistry.
    registerNamespace(String prefix, String uri)
```

sets a one-to-one mapping between `prefix` and `uri` in the global namespace registry of this repository.

Assigning a new prefix to a URI that already exists in the namespace registry erases the old prefix. Apart from the XML restriction (see §10.9.3 *Namespace Restrictions*) this can almost always be done, though an implementation is free to prevent particular remappings by throwing a `NamespaceException`. Re-assigning an already registered prefix to a new URI in effect unregisters its former URI.

10.12.2 Unregistering a Namespace

The method

```
void NamespaceRegistry.unregisterNamespace(String prefix)
```

removes a namespace mapping from the registry.

10.12.3 Namespace Restrictions

The following restrictions apply to registering, re-registering and unregistering namespaces:

- To avoid conflicts with XML , attempting to register a prefix that begins with the characters "xml" (in any combination of case) will throw a `NamespaceException`.
- Attempting to re-assign or unregister a built-in prefix (`jcr`, `nt`, `mix`, `sv`, `xml`, or the empty prefix) will throw a `NamespaceException`.
- An attempt to unregister a namespace that is not currently registered will throw a `NamespaceException`.
- An implementation may prevent the re-assignment or unregistration of any prefixes for implementation-specific reasons by throwing a `NamespaceException`.

10.12.4 Namespace Information

The following methods provide information about the state of the registry:

```
String[] NamespaceRegistry.getPrefixes()
```

returns all currently registered prefixes.

```
String[] NamespaceRegistry.getURIs()
```

returns all currently registered URIs.

```
String NamespaceRegistry.getURI(String prefix)
```

returns the URI currently mapped to the given `prefix`.

```
String NamespaceRegistry.getPrefix(String uri)
```

returns the prefix currently mapped to the given `uri`.

10.12.4.1 Relationship to Session Namespace Mapping

The repository namespace registry serves as the default mapping and is copied to a session's internal mapping table on session creation. The mappings can then be changed independently of the registry within the scope of that session. The methods shown here affect and report only the state of the central registry. Existing local namespace mappings will not be affected by changes to the persistent namespace registry.

11 Import

A repository may support the bulk *import* of content from XML. Two XML mappings are defined: *document view* and *system view*. The former is used primarily for the import of arbitrary XML into the repository while the latter is a full serialization of repository content (see §7 *Export*). A repository that supports import must support both formats.

Whether an implementation supports import can be determined by querying the repository descriptor table with

```
Repository.OPTION_XML_IMPORT_SUPPORTED.
```

A return value of `true` indicates support (see §24.2 *Repository Descriptors*).

11.1 Importing Document View

The *document view* XML mapping (see §7.3 *Document View*) allows the import of arbitrary XML into the repository. On import, the repository first checks if the incoming XML appears to be a system view document. If it does not, then it is assumed to be in document view form, and the following occurs:

1. For each XML namespace declaration with prefix `P` and URI `U`:
    a. If the namespace registry already contains a mapping of some prefix `P'` to `U` (where `P'` may or may not be equal to `P`) then the namespace registry is left unchanged.
    b. If the namespace registry does not contain a mapping to `U` then such a mapping is added to the registry. The prefix assigned may be *P*, if *P* is not already used in the registry, otherwise the repository must generate and assign a new, previously unused, prefix.
2. Each XML element `E` becomes a JCR node of the same name, `E`.
3. The node type of the JCR node `E` is determined by the implementation in accordance with its policy on respecting property semantics (see §11.3 *Respecting Property Semantics* and §11.4 *Determining Node Types*).
4. Each child XML element `C` of XML element `E` becomes a JCR child node `C` of node `E`.
5. Each XML attribute `A` within an XML element `E` becomes a property `A` of JCR node `E`. The value of each XML attribute `A` becomes the value of the corresponding property `A`.
6. The type of each imported property is determined by the implementation in accordance with its policy on respecting property semantics (see §11.3 *Respecting Property Semantics* and §11.4 *Determining Node Types*).
7. Escape sequences representing non-XML-valid characters in element names and whitespace in attribute values may be encountered (for example, if the incoming XML stream is the product of an earlier

document view export). In such cases, whether the escape sequences are decoded is left up to the implementation. Note that the predefined entity references `&amp;`, `&lt;`, `&gt;`, `&apos;` and `&quot;`, as well as all other entity and character references, must be decoded in any case, in accordance with the XML specification.

8. An implementation that respects node type information may be able to determine whether a particular attribute is intended to be a single or multi-value property, and treat any spaces embedded in the value accordingly (either as delimiters or as literal spaces). Implementations are also free to rely on information external to this specification (such as any schema associated by the incoming XML) to help determine the intended interpretation of whitespace within a particular incoming attribute value.

9. Text within an XML element `E` becomes a `STRING` property called `jcr:xmlcharacters` of a JCR node called `jcr:xmltext`, which itself becomes a JCR child node of the node `E`. The value of `E/jcr:xmltext/jcr:xmlcharacters` will be the character data passed to `ContentHandler.characters`.

10. If import is done through the `ContentHandler` returned by `getImportContentHandler`, data passed to `ContentHandler.ignorableWhitespace` is ignored.

11. If import is done through `importXML`, pure whitespace between elements (that is, a string containing no non-whitespace characters) is ignored. However, whitespace leading, trailing and between non-whitespace characters is included in the text that is stored in `E/jcr:xmltext/jcr:xmlcharacters`.

11.1.1 Roundtripping

Not all information within the infoset of an XML document is maintained on import to document view. Information lost will include processing instructions, the distinction between text and CDATA and namespace scoping at the sub-document level. As a result, perfect roundtripping of a full XML infoset is not possible through document view.

On document view import, the repository will automatically add repository metadata in the form of JCR properties (at least `jcr:primaryType`, for example), if these are not already present in the incoming XML. When re-exported using document view, the resulting XML will contain these properties in the form of XML attributes. As a result, the application must take care of stripping out unwanted repository metadata.

11.2 Import System View

Given a system view XML document the subgraph constructed upon import is determined by reversing the mapping discussed in §7.2 *System View*. Though the mapping is largely straightforward some special considerations are discussed in §11.3 *Respecting Property Semantics* and §11.9 *Importing jcr:root*.

11.3 Respecting Property Semantics

During either system or document view import, XML elements (in system view) or XML attributes (in document view) may be encountered that correspond to JCR properties with repository-level semantics such as the `jcr`-prefixed properties of such node types as `nt:base`, `mix:referenceable` or `mix:versionable`, among others.

When an element or attribute representing such a property is encountered, an implementation may either *skip* it or *respect* it.

- A repository that respects a particular element or attribute must import it and alter the internal state of the repository in accordance with the semantics of the property given the configuration of that repository instance. For example, a repository that respects `jcr:primaryType` will attempt to create a node of the indicated primary node type. If that node type is not supported, the repository will throw an exception.
- A repository that skips an element or attribute must not import it all. *It must not import it but then ignore the semantics of the resulting property*.

The implementation-specific policy regarding what to skip and what to respect must be internally consistent. For example, it makes no sense to skip `jcr:mixinTypes` (thus missing the presence of `mix:lockable`, for example) and yet respect `jcr:lockOwner` and `jcr:lockIsDeep`.

If an implementation chooses to skip `jcr:primaryType`, the node type of the imported node is determined by the implementation (see §11.5 *Determining Node Types*).

11.4 Determining Node Types

In cases of XML import where primary node type information is unavailable, either because it is skipped or because it is not available (as is the case on document view import of arbitrary XML), the implementation must determine an appropriate node type to assign to each newly created node. How this is done is implementation-dependent.

11.5 Determining Property Types

On import of arbitrary XML using document view, the implementation must determine a suitable property type for each incoming property. Determination of the property type must be done as follows:

- If the property type is determinable from the node type assigned to its node (regardless of how this node type is itself determined; see §11.5 *Determining Node Types*) then that property type is used.
- If the property type is not determinable from the node type assigned to its node then the determination of the property is left up to the implementation. For example, an implementation may use `STRING` properties exclusively, or attempt to "guess" the type according to an analysis of the content.

11.6 Event-Based Import Methods

The `Workspace` and `Session` interfaces provide the following event-based import methods:

```
org.xml.sax.ContentHandler
  Workspace.getImportContentHandler(String parentAbsPath,
                                    int uuidBehavior)
```

and

```
org.xml.sax.ContentHandler
  Session.getImportContentHandler(String parentAbsPath,
                                  int uuidBehavior)
```

These methods return an `org.xml.sax.ContentHandler` without altering either the `Workspace` or `Session`. The actual changes to the repository are made through the methods of the `ContentHandler`[19]. Invalid XML data will cause the `ContentHandler` to throw a `SAXException`.

If the incoming XML is a *system view* XML document then it is interpreted as such, otherwise it is imported as *document view*.

The incoming XML is imported into a subgraph of items immediately below the node at `parentAbsPath`.

11.6.1 Workspace Event-Based Import

A `ContentHandler` acquired through the `Workspace` method dispatches changes immediately. Node type constraints are enforced by the `ContentHandler` by throwing a `SAXException` during deserialization. However, which node type constraints are enforced depends upon whether node type information in the imported data is respected, and this is an implementation-specific issue (see §11.3 *Respecting Property Semantics*).

11.6.2 Session Event-Based Import

A `ContentHandler` acquired through the `Session` will build the graph of new items in the transient session store. The changes are then dispatched on `save`.

Different node type constraints may be enforced at different times. Those that would be immediately enforced on a core write method (see §10.2 *Core Write Methods*) of that particular implementation will cause the returned `ContentHandler` to throw an immediate `SAXException`. All other node type constraints are enforced as they would be if made through the core write methods. However, which node type constraints are enforced also depends upon

---

[19] See `http://java.sun.com/j2se/1.4.2/docs/api/org/xml/sax/ContentHandler.html`.

whether node type information in the imported data is respected, which is an implementation-specific issue (see §11.3 *Respecting Property Semantics*).

11.7 Stream-Based Import Methods

The `Workspace` and `Session` interfaces provide the following stream-based import methods:

```
void Workspace.importXML(String parentAbsPath,
                         InputStream in,
                         int uuidBehavior)
```

and

```
void Session.importXML(String parentAbsPath,
                       InputStream in,
                       int uuidBehavior)
```

These methods import the XML document in the input stream and add the resulting item subgraph as a child of the node at `parentAbsPath`. If the incoming XML is a *system view* XML document then it is interpreted as such, otherwise it is imported as *document view*.

11.7.1 Workspace Stream-Based Import

On `Workspace.importXML` changes are dispatched immediately. Node type constraints are enforced by throwing a `ConstraintViolationException`. However, which node type constraints are enforced depends upon whether node type information in the imported data is respected, which is an implementation-specific issue (see §11.3 *Respecting Property Semantics*).

11.7.2 Session Stream-Based Import

On `Session.importXML` changes remain pending until dispatched on `save`. Node type constraints that would be immediately enforced on a core write method (see §10.2 *Core Write Methods*) of that particular implementation will cause an immediate `ConstraintViolationException` during import. All other node type constraints are enforced as they would be if made through the core write methods. However, which node type constraints are enforced depends upon whether node type information in the imported data is respected, and this is an implementation-specific issue (see §11.3 *Respecting Property Semantics*).

11.8 Identifier Handling

The `uuidBehavior` flag governs how the identifiers of imported nodes are handled. There are four options, defined as constants in the interface `javax.jcr.ImportUUIDBehavior`:

11.8.1 Create New Identifiers

`IMPORT_UUID_CREATE_NEW`: Incoming nodes are assigned newly created identifiers upon addition to the workspace. As a result, identifier collisions never occur.

11.8.2 Remove Existing Node

`IMPORT_UUID_COLLISION_REMOVE_EXISTING`: If an incoming non-shareable node has the same identifier as a node already existing in the workspace then the already existing node (and its subgraph) is removed from wherever it may be in the workspace before the incoming node is added. Note that this can result in nodes "disappearing" from locations in the workspace that are remote from the location to which the incoming subgraph is being written. In the case of shareable node, however, the behavior differs (see §14.1.2 *Shared Node Creation on Import*).

11.8.3 Replace Existing Node

`IMPORT_UUID_COLLISION_REPLACE_EXISTING`: If an incoming non-shareable node has the same identifier as a node already existing in the workspace, then the already existing node is replaced by the incoming node in the same position as the existing node. Note that this may result in the incoming subgraph being disaggregated and "spread around" to different locations in the workspace. In the most extreme case this behavior may result in no node at all being added as child of `parentAbsPath`. This will occur if the topmost element of the incoming XML has the same identifier as an existing node elsewhere in the workspace. In the case of shareable node, however, the behavior differs (see §14.1.2 *Shared Node Creation on Import*).

11.8.4 Throw on Identifier Collision

`IMPORT_UUID_COLLISION_THROW`: If an incoming non-shareable node has the same identifier as a node already existing in the workspace, then either a `SAXException` is thrown by the `ContentHandler` (in the case of event-based import) or an `ItemExistsException` is thrown by the `importXML` method (in the case of stream-based import). In the case of shareable nodes, the behavior differs (see §14.1.2 *Shared Node Creation on Import*).

11.8.5 Usage of Term UUID

The term "UUID" occurs in the names of certain properties, classes and methods in JCR 1.0. This usage is maintained in JCR 2.0 to preserve compatibility with JCR 1.0. However, in the context of JCR 2.0 these names should be understood to apply to identifiers *in general* and not just identifiers that use of the UUID syntax, or that possess global uniqueness.

11.9 Importing *jcr:root*

If the root node of a workspace is exported it will be rendered in XML (in either view) under the name `jcr:root`. In addition, if the root node is referenceable this will be recorded in the serialization of the `jcr:uuid` property.

If this XML document is imported back into the workspace a number of different results may occur, depending on the methods and settings used to perform the import. The following summarizes the possible results of using various `uuidBehavior` values (in either using either `Workspace.getImportContentHandler` or `Workspace.importXML`) when a node

with the same identifier as the existing root node is encountered on import (the constants below are defined in the interface `javax.jcr.ImportUUIDBehavior`).

> `IMPORT_UUID_CREATE_NEW`: The XML element representing `jcr:root` is rendered as a normal node at the position specified (with the name `jcr:root`). It gets a new identifier, so there is no effect on the existing root node of the workspace.
>
> `IMPORT_UUID_COLLISION_REMOVE_EXISTING`: If deserialization is done through a `ContentHandler` (acquired by `getImportContentHandler`) a `SAXException` will be thrown. Similarly, if deserialization is done through `importXML` a `ConstraintViolationException` will be thrown. Note that this is simply a special case of the general rule that under this `uuidBehavior` setting, an exception will be thrown on any attempt to import a node with the same identifier as the node at `parentAbsPath` *or any of its ancestors* (which, of course, includes the root node).
>
> `IMPORT_UUID_COLLISION_REPLACE_EXISTING`: This setting is equivalent to importing into the `Session` and then calling `save` since `save` always operates according to identifier. In both cases the result is that the root node of the workspace will be replaced along with its subgraph (i.e., the whole workspace), just as if the root node had been altered through the normal `getNode`-*make change*-`save` cycle.
>
> `IMPORT_UUID_COLLISION_THROW`: Under this setting a `ContentHandler` will throw a `SAXException` and the `importXML` method will throw `ItemExistsException`.

Note that an implementation is always free to prevent the replacement of a root node (or indeed any node) either through access control restrictions or other implementation-specific restrictions.

12 Observation

A repository may support *observation*, which enables an application to receive notification of persistent changes to a workspace. JCR defines a general event model and specific APIs for asynchronous and journaled observation. A repository may support asynchronous observation, journaled observation or both.

Whether an implementation supports asynchronous or journaled observation can be determined by querying the repository descriptor table with the keys

`Repository.OPTION_OBSERVATION_SUPPORTED` or

`Repository.OPTION_JOURNALED_OBSERVATION_SUPPORTED`.

A return value of `true` indicates support (see §24.2 *Repository Descriptors*).

12.1 Event Model

A persisted change to a workspace is represented by a set of one or more *events*. Each event reports a single simple change to the structure of the persistent workspace in terms of an item added, changed, moved or removed. The six standard event types are:

`NODE_ADDED`,
`NODE_MOVED`,
`NODE_REMOVED`,
`PROPERTY_ADDED`,
`PROPERTY_REMOVED` and
`PROPERTY_CHANGED`.

A seventh event type,

`PERSIST`,

may also appear in certain circumstances (see §12.7.3 *Event Bundling in Journaled Observation*).

12.2 Scope of Event Reporting

The scope of event reporting is implementation-dependent. An implementation should make a *best-effort* attempt to report all events, but may exclude events if reporting them would be impractical given implementation or resource limitations. For example, on an import, move or remove of a subgraph containing a large number of items, an implementation may choose to report only events associated with the root node of the affected graph and not those for every subitem in the structure.

12.2.1 Externally Caused Events

Some implementations may expose capabilities through the JCR API while also being writable through a mechanism external to JCR. Whether events are generated for changes made through such external means is left up to the implementation.

12.3 The Event Object

Each event generated by the repository is represented by an `Event` object.

12.3.1 Event Types

The type of an `Event` is retrieved through

```
int Event.getType()
```

which returns one of the `int` constants found in the `Event` interface: `NODE_ADDED`, `NODE_MOVED`, `NODE_REMOVED`, `PROPERTY_ADDED`, `PROPERTY_REMOVED`, `PROPERTY_CHANGED` or `PERSIST`.

12.3.2 Event Information

Each `Event` is associated with a path, an identifier and an information map, the interpretation of which depend upon the event type.

The event path is retrieved through

```
String Event.getPath(),
```

the identifier through

```
String Event.getIdentifier()
```

and the information map through

```
java.util.Map Event.getInfo()
```

If the event is a `NODE_ADDED` or `NODE_REMOVED` then,

- `Event.getPath()` returns the absolute path of the node that was added or removed.
- `Event.getIdentifier()` returns the identifier of the node that was added or removed.
- `Event.getInfo()` returns an empty `Map` object.

If the event is `NODE_MOVED` then,

- `Event.getPath()` returns the absolute path of the *destination* of the move.
- `Event.getIdentifier()` returns the identifier of the node that was moved.
- `Event.getInfo()` returns a `Map` containing parameter information from the method that caused the event (see §12.4.3 *Event Information on Move and Order*).

If the event is a `PROPERTY_ADDED`, `PROPERTY_CHANGED` or `PROPERTY_REMOVED` then,

- `Event.getPath()` returns the absolute path of the property that was added, changed or removed.

- `Event.getIdentifier()` returns the identifier of the parent node of the property that was added, changed or removed.
- `Event.getInfo()` returns an empty `Map` object.

If the event is a `PERSIST` (see §12.6.3 *Event Bundling in Journaled Observation*) then `Event.getPath()` and `Event.getIdentifier()` return `null` and `Event.getInfo()` returns an empty `Map`.

12.3.3 Event Information on Move and Order

On a `NODE_MOVED` event, the `Map` object returned by `Event.getInfo()` contains parameter information from the method that caused the event. There are three JCR methods that cause this event type: `Session.move`, `Workspace.move` and `Node.orderBefore`.

If the method that caused the `NODE_MOVE` event was a `Session.move` or `Workspace.move` then the returned `Map` has keys `srcAbsPath` and `destAbsPath` with values corresponding to the parameters passed to the `move` method, as specified in the Javadoc.

If the method that caused the `NODE_MOVE` event was a `Node.orderBefore` then the returned `Map` has keys `srcChildRelPath` and `destChildRelPath` with values corresponding to the parameters passed to the `orderBefore` method, as specified in the Javadoc.

12.3.3.1 Externally Caused NODE_MOVED Event

In a repository that reports events caused by mechanisms external to JCR (see §12.2.1 *Externally Caused Events*), the keys and values found in the information map returned on a `NODE_MOVED` are implementation-dependent.

12.3.4 User ID

An `Event` also records the identity of the `Session` that caused it.

```
String Event.getUserID()
```

returns the user ID of the `Session`, which is the same value that is returned by `Session.getUserID()` (see §4.4.1 *User*).

12.3.5 User Data

An `Event` may also contain arbitrary string data specific to the session that caused the event. A session may set its current user data using

```
void ObservationManager.setUserData(Sting userData).
```

Typically a session will set this value in order to provide information about its current state or activity. Any events produced by the session while its user data is set to particular value will carry that value with them. A process responding to these events will then be able to access this information through

```
String Event.getUserData()
```

and use the retrieved data to provide additional context for the event, beyond that provided by the identify of the causing session alone.

12.3.6 Event Date

An event also records the time of the change that caused it. This acquired through

```
long Event.getDate()
```

The date is represented as a millisecond value that is an offset from the epoch January 1, 1970 00:00:00.000 GMT (Gregorian). The granularity of the returned value is implementation-dependent.

12.4 Event Bundling

A repository that supports observation *may* support event bundling under asynchronous observation, journaled observation, or both.

In such a repository, events are produced in bundles where each corresponds to a single atomic change to a persistent workspace and contains only events caused by that change (see §10.1 *Types of Write Methods*).

For example, given a session with a set of pending node and property additions, on persist, a `NODE_ADDED` or `PROPERTY_ADDED` is produced, as appropriate, for each new item. This set of events is the event bundle associated with that particular persist operation. By grouping events together in this manner, additional contextual information is provided, simplifying the interpretation of the event stream.

12.4.1 Event Ordering

In both asynchronous and journaled observation the order of events within a bundle and the order of event bundles is not guaranteed to correspond to the order of the operations that produced them.

12.5 Asynchronous Observation

Asynchronous observation enables an application to respond to changes made in a workspace as they occur.

An application connects with the asynchronous observation mechanism by registering an event listener with the workspace. Listeners apply *per workspace*, not repository-wide; they only receive events for the workspace in which they are registered. An event listener is an application-specific class implementing the `EventListener` interface that responds to the stream of events to which it has been subscribed.

This observation mechanism is *asynchronous* in that the operation that causes an event to be dispatched does not wait for a response to the event from the listener; execution continues normally on the thread that performed the operation.

12.5.1 Observation Manager

Registration of event listeners is done through the `ObservationManager` object acquired from the `Workspace` through

```
ObservationManager Workspace.getObservationManager().
```

12.5.2 Adding an Event Listener

An event listener is added to a workspace with

```
void ObservationManager.
    addEventListener(EventListener listener,
                     int eventTypes,
                     String absPath,
                     boolean isDeep,
                     String[] uuid,
                     String[] nodeTypeName,
                     boolean noLocal)
```

The `EventListener` object passed is provided by the application. As defined by the `EventListener` interface, this class must provide an implementation of the `onEvent` method:

```
void EventListener.onEvent(EventIterator events)
```

When an event occurs that falls within the scope of the listener (see 12.6.3 *Event Filtering*), the repository calls the `onEvent` method invoking the application-specific logic that processes the event.

12.5.3 Event Filtering

Which events a listener receives are determined as follows.

12.5.3.1 Access Privileges

An event listener will only receive events for which its `Session` (the `Session` associated with the `ObservationManager` through which the listener was added) has sufficient access privileges.

12.5.3.2 Event Types

An event listener will only receive events of the types specified by the `eventTypes` parameter of the `addEventListener` method. The `eventTypes` parameter is an `int` composed of the bitwise `AND` of the desired event type constants.

12.5.3.3 Local and Nonlocal

If the `noLocal` parameter is `true`, then events generated by the `Session` through which the listener was registered are ignored.

12.5.3.4 Node Characteristics

Node characteristic restrictions on an event are stated in terms of the *associated parent node* of the event. The associated parent node of an event is the *parent* node of the item at (or formerly at) the path returned by `Event.getPath()`.

12.5.3.4.1 Location

If `isDeep` is `false`, only events whose associated parent node is at `absPath` will be received.

If `isDeep` is `true`, only events whose associated parent node is at or below `absPath` will be received.

It is permissible to register a listener for a path where no node currently exists.

12.5.3.4.2 Identifier

Only events whose associated parent node has one of the identifiers in the `uuid String` array will be received. If this parameter is `null` then no identifier-related restriction is placed on events received. Note that specifying an empty array instead of `null` results in no nodes being listened to. The `uuid` is used for backwards compatibility with JCR 1.0.

12.5.3.4.3 Node Type

Only events whose associated parent node is of one of the node types in the `nodeTypeNames String` array will be received. If this parameter is `null` then no node type-related restriction is placed on events received. Note that specifying an empty array instead of `null` results in no nodes being listened to.

12.5.4 Re-registration of Event Listeners

The filters of an already-registered `EventListener` can be changed at runtime by re-registering the same `EventListener` Java object with a new set of filter arguments. The implementation must ensure that no events are lost during the changeover.

12.5.5 Implementation-Specific Restrictions

In addition to the filters placed on a listener though the `addEventListener` method, the scope of observation support, in terms of which subgraphs are observable, may also be subject to implementation-specific restrictions. For example, in some repositories observation of changes in the `jcr:system` subgraph may not be supported (see 3.11 *System Node*).

12.5.6 Event Iterator

In asynchronous observation the `EventIterator` holds an event bundle or a single event, if bundles are not supported. `EventIterator` inherits the methods of `RangeIterator` and adds an `Event`-specific `next` method:

```
Event EventIterator.nextEvent()
```

(see §5.9 *Iterators*)

12.5.7 Listing Event Listeners

```
EventListenerIterator ObservationManager.
    getRegisteredEventListeners()
```

12.5.7.1 EventListenerIterator

Methods that return a set of `EventListener` objects (such as `ObservationManager.getRegisteredEventListeners`) do so using an `EventListenerIterator`. The `EventListenerIterator` class inherits the methods of `RangeIterator` and adds an `EventListener`-specific `next` method:

```
EventListener EventListenerIterator.nextEventListener()
```

(see §5.9 *Iterators*)

12.5.8 Removing Event Listeners

```
void ObservationManager.
    removeEventListener(EventListener listener)
```

12.5.9 User Data

```
void ObservationManager.setUserData(String userData)
```

12.6 Journaled Observation

Journaled observation allows an application to periodically connect to the repository and receive a report of changes that have occurred since some specified point in the past (for example, since the last connection). Whether a repository records a per-workspace event journal is up to the implementation's configuration.

12.6.1 Event Journal

The `EventJournal` of a workspace instance is acquired by calling either

```
EventJournal ObservationManager.getEventJournal()
```

or

```
EventJournal getEventJournal(int eventTypes,
                             String absPath,
                             boolean isDeep,
                             String[] uuid,
                             String[] nodeTypeName,
                             boolean noLocal).
```

Events reported by this `EventJournal` instance will be filtered according to the current session's access rights, any additional restrictions specified through implementation-specific configuration and, in the case of the second signature, by the parameters of the method. These parameters are interpreted in the same way as in the method `addEventListener`.

An `EventJournal` is an extension of `EventIterator` that provides the additional method `skipTo(Calendar date)`.

```
void EventJournal.skipTo(Calendar date)
```

12.6.2 Journaling Configuration

An implementation is free to limit the scope of journaling both in terms of coverage (that is, which parts of a workspace may be observed and which events are reported) and in terms of time and storage space. For example, a repository can limit the size of a journal log by stopping recording after it has reached a certain size, or by recording only the tail of the log (deleting the earliest event when a new one arrives). Any such mechanisms are assumed to be within the scope of implementation configuration.

12.6.3 Event Bundling in Journaled Observation

In journaled observation dispatching is done by the implementation writing to the event journal.

If event bundling is supported a `PERSIST` event is dispatched when a persistent change is made to workspace bracketing the set of events associated with that change. This exposes event bundle boundaries in the event journal.

Note that a `PERSIST` event will never appear within an `EventIterator` since, in asynchronous observation, the iterator itself serves to define the event bundle.

In repositories that do not support event bundling, `PERSIST` events do not appear in the event journal.

12.7 Importing Content

Whether events are generated for each node and property addition that occurs when content is imported into a workspace (see §11 *Import*) is left up to the implementation.

12.8 Exceptions

The method `EventListener.onEvent` does not specify a `throws` clause. This does not prevent a listener from throwing a `RuntimeException`, although any listener that does should be considered to be in error.

13 Workspace Management

A repository may support *workspace management*, which enables the creation and deletion of workspaces through the JCR API. A repository that supports this feature must support the semantics of multiple workspaces (see §3.10 *Multiple Workspaces*) and support cross-workspace operations (see §10.7.2 *Copying Across Workspaces* and §10.8 *Cloning and Updating Nodes*).

Whether an implementation supports workspace management can be determined by querying the repository descriptor table with

```
Repository.OPTION_WORKSPACE_MANAGMENT_SUPPORTED.
```

A return value of `true` indicates support (see §24.2 *Repository Descriptors*).

13.1 Creation and Deletion of Workspaces

The method

```
void Workspace.createWorkspace(String name)
```

creates a new workspace with the specified name. The new workspace will contain only a root node. The new workspace can be accessed through a login specifying its name.

```
void Workspace.createWorkspace(String name,
                               String srcWorkspace)
```

creates a new workspace with the specified name initialized with a clone of the content of the workspace `srcWorkspace` (see §10.8.1 *Cloning Nodes Across Workspaces*). Semantically, this method is equivalent to creating a new workspace and manually cloning `srcWorkspace` to it. However, this method may assist some implementations in optimizing subsequent `Node.update` and `Node.merge` calls between the new workspace and its source. The new workspace can be accessed through a login specifying its name.

```
void Workspace.deleteWorkspace(String name)
```

Deletes the workspace with the specified name from the repository, deleting all content within it.

14 Shareable Nodes

A repository may support *shareable nodes*. This section describes the syntax and behavior of the Java API for shareable nodes. For details on the shareable nodes model see §3.9 *Shareable Nodes Model*.

Whether an implementation supports shareable nodes can be determined by querying the repository descriptor table with

```
Repository.OPTION_SHAREABLE_NODES_SUPPORTED.
```

A return value of `true` indicates support (see §24.2 *Repository Descriptors*).

14.1 Creation of Shared Nodes

Cloning a `mix:shareable` node into the same workspace is the standard way of creating a shared node.

Given workspace `W`, and an existing `mix:shareable` node at `/A/B/C`, the call

```
W.clone("W", "/A/B/C", "/X/Y/Z", false)
```

will create a new node at `/X/Y/Z` that shares with `/A/B/C`.

Note that if the `removeExisting` flag is set to `true`, the `Workspace.clone` does not create a shared node, but instead behaves identically to a `Workspace.move`.

14.1.1 Shared Node Creation on Restore

If `VersionManager.restore`, `restoreByLabel`, `merge` or `update` is called and this call would create a node with the same identifier as that of an existing `mix:shareable` node in the same workspace without at the same time removing that existing node (that is, `removeExisting` is set to `false`), then the new node is created and is added to the shared set of the existing `mix:shareable` node.

14.1.2 Shared Node Creation on Import

During import the behavior of the `IMPORT_UUID_COLLISION_THROW` indicates that if an incoming referenceable node has the same identifier as an existing `mix:shareable` node in the workspace, the incoming node is created and added to the shared set of the existing `mix:shareable` node (see §3.9.1 *mix:shareable*). Note that if the import in question is a `Session` import (see §11.6.2 *Session Event-Based Import* and §11.7.2 *Session Stream-Based Import*) new shared transient nodes will be created. These nodes are not considered to be *new*, in the sense that `Node.isNew` will return `false`.

14.2 Shared Set

The *shared set* of a node consists of all nodes (including itself) with which it shares. This set is retrieved with

```
NodeIterator Node.getSharedSet().
```

14.3 Removing Shared Nodes

The method

```
void Node.removeShare()
```

removes the node from its shared set without affecting the other nodes in the set. The method

```
void Node.removeSharedSet()
```

removes the node and all the members of its shared set.

In the first case, assuming more than one member in the shared set, the children of the removed node are unaffected since they still have at least one other node as parent. In the second case, however, the children of the shared set are removed.

In cases where the shared set consists of a single node, or when these methods are called on a non-shareable node, their behavior is identical to `Node.remove()`.

When applied to a shared node with at least one other member in its shared set, the method

```
void Item.remove
```

or

```
void Session.removeItem
```

may behave as `Node.removeShare()` or as `Node.removeSharedSet()`. Which behavior is adopted is an implementation issue.

The behavior of `Node.remove()` is permitted to vary across repositories because the details of the underlying implementation will make one or the other of the behaviors more natural for that repository. In particular if a repository implements a shared set by one "primary" parent (that controls the lifetime of the child) and zero or more "secondary" parents (that reference that child), then `Item.remove` is most naturally interpreted differently on the primary parent and one of the secondary parents. To force that repository to do a `Node.removeShare` on the primary parent would require that implementation to pick one of the secondary parents as the new primary parent, and change all of the other secondary parents to refer to that new primary parent.

For all three methods, the removal is dispatched on `Session.save()`.

14.4 Transient Layer

When a change is made to a shared node in the transient layer, `Node.isModified` becomes `true` and that change is visible in all nodes in the shared set of that node. After a transient shared node is dispatched, `Node.isModified` becomes `false` for all nodes in the shared set of that node.

14.5 Copy

The new nodes created by a `copy` are never in the shared set of any node that existed before the `copy`, but if two nodes `A` and `B` in the source of a `copy` are in

the same shared set S, then the two resulting nodes A' and B' in the destination of the copy must both be in the same shared set S', where S and S' are disjoint.

14.6 Share Cycles

In an implementation that forbids share cycles, any session-write method that can create a shared node will cause a ShareCycleException to be thrown either immediately or on save, if persisting the change would result in a share cycle.

Similarly, any workspace-write method that can create a shared node will throw a ShareCycleException if completion of the operation would result in a share cycle.

In an implementation that does not prevent share cycles, checking for cycles is left to the repository user.

14.7 Export

When more than one shared node in a given shared set is exported to an XML document, the first node in that shared set is exported in the normal fashion (with all of its properties and children), but any subsequent shared node in that shared set is exported as a special node of type nt:share, which contains only the jcr:uuid property of the shared node and the jcr:primaryType property indicating the type nt:share. Note that nt:share only appears in a serialization document, and never appears as a node type of a node in a repository.

14.8 Import

When an XML element with node type nt:share is imported into a repository that does not support shared nodes, the import must fail (getImportContentHandler will throw a SAXException, while importXML will throw an UnsupportedRepositoryOperationException).

14.9 Observation

When a property of a shared node is modified, or when a child item is added to or deleted from a shared node, although that property or child node modification is performed on every node in the shared set of that node, only one event is fired for the shared set. Which node in the shared set is identified in the event is implementation-defined.

14.10 Locking

When a lock is added or removed from a shared node, it is automatically added or removed from every node in the shared set of that node.

If at least one share-ancestor of a node N holds a deep locked then that lock applies to N, resulting in N being locked.

14.11 Node Type Constraints

All the nodes in a shared set always have the same declared primary node type and the same set of assigned mixin node types. Since different nodes in the shared set may have different parents, those parents must be of an appropriate node type to have a child of with these types.

If the members of a shared set correspond to child node definitions (in their respective parents) with conflicting *protected* settings, the effective protected value of all the members of the shared set will be the logical `OR` of the protected settings of the set of child node definitions.

14.12 Versioning

If a node is versionable then all nodes within its shared set share the same version history. Under full versioning this follows logically from the fact that the nodes all share the same `jcr:versionHistory` reference (see §3.13.2.2 *mix:versionable*), pointing to a single common `nt:versionHistory` node (see §3.13.5.1 *nt:versionHistory*).

On check-in of a node *N* within the shared set, its versionable state is determined just as in the non-shared case, but because the node is shared, the resulting version will also reflect the versionable state of any node *N'* in the shared set of *N*.

On check-in of a parent *M* of a shared node *N* the contribution of *N* to the versionable state of *M* is determined according to the OPV of *N*. Note that the OPV of two nodes *N* and *N'* in the same shared set (with parent node *M* and *M'*, respectively) may *differ* because the OPV of *N* is determined by the node type of *M*, while that of *N'* is determined by the node type of *M'*.

14.13 Restore

The effect of shared nodes on `restore` falls into three cases:

- A `restore` that causes the creation of a new shared node (see §14.1.1 *Shared Node Creation on Restore*).
- A `restore` that causes the removal of a shared node: In this case the particular shared node is removed but its descendants continue to exist below the remaining members of the shared set.
- A `restore` causes a change to the state of a shared node: Any change is reflected in all nodes in its shared set.
- A `restore` that causes a change below a shared node: The subgraph is changed as usual and the change is visible through many paths.

14.14 IsSame

If node `/a/b` shares with node `/a/c` then these two nodes are considered "the same" according to the `Item.isSame()` method. Additionally, if the shared nodes have a property `p`, then `/a/b/p` and `/a/c/p` are also considered "the same". If they have a child node `x` then, similarly, `/a/b/x` and `/a/c/x` are also the "the same".

14.15 RemoveMixin

If an attempt is made to remove the `mix:shareable` mixin node type from a node in a shared set the implementation may either throw a `ConstraintViolationException` or allow the removal and change the subgraph in some implementation-specific manner. One possibility is to replace the node

with a copy that has no children (if this does not violate the node type restrictions of that node). Another possibility is to give the node a copy of all of its descendants (unless the resulting copy operation would be unfeasible, as would be the case if a share cycle were involved).

14.16 Query

If a query matches two or more nodes in a shared set, whether all of these nodes or just one is returned in the query result is an implementation issue.

This variability is allowed since different implementations might have different "natural" behaviors, and it would be expensive for an implementation to compute the answer that is "unnatural" for that implementation.

If a query matches a descendant node of a shared set, it appears in query results only once.

15 Versioning

A repository may support *simple versioning* or *full versioning*. This section describes the syntax and behavior of the Java API for both types of versioning. Details on the underlying concepts, data structures and node types can be found in §3.13 *Versioning Model*.

Whether an implementation supports simple versioning can be determined by querying the repository descriptor table with

```
Repository.OPTION_SIMPLE_VERSIONING_SUPPORTED.
```

Whether it supports full versioning can be determined by querying

```
Repository.OPTION_VERSIONING_SUPPORTED.
```

A return value of `true` indicates support (see §24.2 *Repository Descriptors*).

15.1 Creating a Versionable Node

A new versionable node is created by assigning it the appropriate mixin type: `mix:simpleVersionable` under simple versioning or `mix:versionable` under full versioning. This may be done either to an existing node, through a `Node.addMixin` or at node creation, through assignment of a primary type that inherits from the mixin. Some repositories may also automatically assign a versionable mixin on creation of certain nodes (see §3.7.6 *Node Type Inheritance* and §10.10 *Node Type Assignment*).

Under both simple and full versioning, on persist of a new versionable node `N` that neither corresponds nor shares with an existing node:

- The `jcr:isCheckedOut` property of `N` is set to `true` and
- A new `VersionHistory` (`H`) is created for `N`. `H` contains one `Version`, the root version (V_0) (see §3.13.5.2 *Root Version*).

Additionally, under full versioning:

- A new `nt:versionHistory` node is created and bound to the `VersionHistory` object `H` .
    - The `jcr:versionableUuid` property of `H` is set to the identifier of `N`.
    - If `N` is the result of a copy operation then the `jcr:copiedFrom` property of `H` is set as described in §15.1.4 *Copying Versionable Nodes and Version Lineage*. Otherwise this property is not added.
    - A new `nt:versionLabels` node (`L`) is created as the `jcr:versionLabels` child node of `H`.
    - A new `nt:version` node is created and bound to V_0. This node becomes the `jcr:rootVersion` child node of `H`.
        - A new `nt:frozenNode` node (`F`) is created as the `jcr:frozenNode` child node of V_0. `F` does not hold any state

information about N except the node type and identifier information found in jcr:frozenPrimaryType, jcr:frozenMixinTypes, and jcr:frozenUuid properties (see §3.13.4 *Frozen Nodes*).

- The REFERENCE property jcr:versionHistory of N is initialized to the identifier of H. This constitutes a reference from N to its version history.
- The REFERENCE property jcr:baseVersion of N is initialized to the identifier of V_0. This constitutes a reference from N to its current base version.
- The multi-value REFERENCE property jcr:predecessors of N is initialized to contain a single identifier, that of V_0 (the same as jcr:baseVersion).

15.1.1 VersionHistory Object

The version history of a versionable node is represented by a VersionHistory object acquired through

```
VersionHistory Node.getVersionHistory()
```

or

```
VersionHistory
      VersionManager.getVersionHistory(String absPath)
```

where absPath is the absolute path to the node.

Conversely, given a VersionHistory, the versionable node to which it belongs can be found through

```
String VersionHistory.getVersionableIdentifier()
```

which returns the identifier of the versionable node, which can then be used to get the node itself (see §5.1.4 *Getting a Node by Identifier*).

15.1.1.1 Root Version

The root version of a version history is accessed through

```
Version VersionHistory.getRootVersion().
```

Under full versioning the root version can also be accessed through a Node.getNode or an equivalent standard content access method, since it also exists as an nt:version child node of the nt:versionHistory node, called jcr:rootVersion.

15.1.1.2 Versions

The full set of versions within a version history can be retrieved through

```
VersionIterator VersionHistory.getAllVersions().
```

If the version graph of this history is linear then the versions are returned in order of creation date, from oldest to newest. Otherwise the order of the returned versions is implementation-dependent.

Alternatively, the method

```
VersionIterator VersionHistory.getAllLinearVersions()
```

returns an iterator over all the versions in the *line of descent* from the root version to the base version that is bound to the workspace through which this `VersionHistory` was acquired.

Within a version history `H`, `B` is the base version bound to workspace `W` if and only if there exists a versionable node `N` in `W` whose version history is `H` and `B` is the base version of `N`.

The line of descent from version V_1 to V_2, where V_2 is an eventual successor of V_1, is the ordered list of versions starting with V_1 and proceeding through each direct successor to V_2.

The versions are returned in order of creation date, from oldest to newest.

Note that in a simple versioning repository the behavior of this method is equivalent to returning all versions in the version history in order from oldest to newest.

Versions can also be retrieved directly by name, using

```
Version VersionHistory.getVersion(String versionName),
```

or by label, using,

```
Version VersionHistory.getVersionByLabel(String label)
```

(see §15.4 *Version Labels*).

15.1.1.3 Frozen Nodes

The frozen nodes within each of the versions within the history can be accessed directly from the `VersionHistory` through

```
NodeIterator VersionHistory.getAllFrozenNodes()
```
and

```
NodeIterator VersionHistory.getAllLinearFrozenNodes().
```

These methods return the frozen nodes within the version history corresponding to, and in the same order as, the `Version` objects returned by `VersionHistory.getAllVersions` and `VersionHistory.getAllLinearVersions`, respectively.

15.1.1.4 VersionHistory Extends Node

The `VersionHistory` interface extends `Node`. Under simple versioning version histories are not represented by nodes in content, so the methods inherited from `Node` are not required to function and may instead throw a `RepositoryException`. Under full versioning the `VersionHistory` object

represents the corresponding `nt:versionHistory` node and its `Node` methods must function accordingly.

15.1.2 Getting the Base Version

The method

```
Version VersionManager.getBaseVersion(String absPath)
```

returns the current base version of the versionable node at `absPath`.

15.1.3 Moving Versionable Nodes

When an existing versionable node is moved to a new location with `Workspace.move` or `Session.move`, it maintains the same version history and no changes are made to that history.

15.1.4 Copying Versionable Nodes and Version Lineage

Under both simple and full versioning, when an existing versionable node `N` is copied to a new location either in the same workspace or another, and the repository preserves the versionable mixin (see §10.7.4 *Dropping Mixins on Copy*):

- A copy of `N`, call it `M`, is created, as usual.
- A new, empty, version history for `M`, call it H_M, is also created.

Under full versioning:

- The properties `jcr:versionHistory`, `jcr:baseVersion` and `jcr:predecessors` of `M` are not copied from `N` but are initialized as usual.
- The `jcr:copiedFrom` property of H_M is set to point to the base version of `N`.

15.1.4.1 Version Lineage

The `jcr:copiedFrom` property allows an application to determine the *lineage* of a version across version histories that were produced by copying a versionable node to a new location.

15.1.5 Cloning Versionable Nodes

Under both simple and full versioning, when a versionable node `N` is cloned to another workspace:

- A clone of `N`, call it `N'`, is created, as usual.
- `N'` is initialized to have the same version history and base version as `N`.

Under full versioning:

- The `jcr:versionHistory`, `jcr:baseVersion` and `jcr:predecessors` properties of `N` are copied to `N'` unchanged.

15.1.6 Sharing Versionable Nodes

Under both simple and full versioning, when a new node `N'` is added to the shared set of a shareable, versionable node `N`:

- The shared node `N'` is created, as usual.
- `N'` is initialized to have the same version history and base version as `N`. Unlike in the case of cloning (see §15.1.5 *Cloning Versionable Nodes*) the base versions of `N` and `N'` will always remain identical.

Under full versioning:

- Because nodes in the same shared set have identical properties, `mix:versionable` nodes in the same shared set will necessarily have identical `jcr:versionHistory`, `jcr:baseVersion` and `jcr:predecessors` properties.

15.2 Check-In: Creating a Version

A new *version* of a versionable node is created using

```
Version VersionManager.checkin(String absPath)
```

where `absPath` is the absolute path of the node.

On check-in of a versionable node `N` with version history `H`:

- If `N` is not `mix:simpleVersionable` or `mix:versionable`, an `UnsupportedRepositoryOperationException` is thrown, otherwise,
- if `N` has unsaved changes pending, an `InvalidItemStateException` is thrown, otherwise,
- if `N` is already *checked-in*, this method has no effect and returns the *base version* (see §3.13.6.2 *Base Version*) of `N`, otherwise,
- if `N` has a `jcr:mergeFailed` property present, a `VersionException` is thrown (notice that this is enforced in any case due to the `ABORT` setting of the `jcr:mergeFailed` property's `OnParentVersion` attribute).

Otherwise:

- The subgraph rooted at `N` is made *read-only* (see §15.2.2 *Read-Only on Check-In*).
- A new `Version`, `V`, is created with a system-determined *version name* (see §15.2.1.1 *Version Name*) and a *created date* (see §15.2.1.2 *Created Date*) as part of its state. Under full versioning, a new `nt:version` node is bound to `V` and added as a child node of `H`, with the version name as its node name and the created date as the value of its `jcr:created` property.
- The *versionable state* of `N` is recorded in the *frozen node* `F` of `V` as described in §3.13.9 *Versionable State*. Under full versioning, `F` is added as the `jcr:frozenNode` child node of `V`.

- `V` is added to the version history of `N` as the direct successor of the *base version* of `N`. Under full versioning:
    - The `jcr:predecessors` property of `N` is copied to the `jcr:predecessors` property of `V`.
    - The `jcr:predecessors` property of `N` is set to the empty array.
    - A reference to `V` is added to the `jcr:successors` property of each of the `nt:version` nodes referred to by the `jcr:predecessors` property of `V`.
- The base version of `N` is changed to `V`. Under full versioning, the `jcr:baseVersion` property of `N` is changed to refer to `V`.
- The `jcr:isCheckedOut` property of `N` is set to `false`. This change is a workspace-write and therefore does not require a `save`.
- `N` is now *checked-in*.
- `V` is returned.

15.2.1 Version Object

A version is represented by a `Version` object.

15.2.1.1 Version Name

The name given to a version is automatically generated and must be unique within its version history. How the name is generated is up to the implementation. The name of a version is retrieved with the method

```
String Item.getName(),
```

inherited by `Version`. Under simple versioning this is the only inherited method that is required to function (see §15.2.1.7 *Version Extends Node*).

15.2.1.2 Created Date

```
Calendar Version.getCreated()
```

returns a timestamp indicating the date and time that the version was created. The precision of the timestamp is implementation-dependent.

15.2.1.3 Containing History

```
VersionHistory Version.getContainingHistory()
```

returns the `VersionHistory` that contains this `Version`.

15.2.1.4 Predecessors

```
Version[] Version.getPredecessors()
```

returns the direct predecessors of this `Version`. Under simple versioning this set will be at most of size 1. Under full versioning, this set maybe of size greater than 1, indicating a merge within the version graph.

The method

```
Version Version.getLinearPredecessor()
```

returns the direct predecessor of this `Version` along the same line of descent returned by `VersionHistory.getAllLinearVersions` in the current workspace (see §3.1.8.2 *Current Session and Workspace*), or `null` if no such direct predecessor exists. Note that under simple versioning the behavior of this method is equivalent to getting the unique direct predecessor (if any) of this version.

15.2.1.5 Successors

```
Version[] Version.getSuccessors()
```

returns the direct successors of this `Version`. Under simple versioning this set will be at most of size 1. Under full versioning, this set maybe of size greater than 1, indicating a branch within the version graph.

The method

```
Version Version.getLinearSuccessor()
```

returns the direct successor of this `Version` along the same line of descent returned by `VersionHistory.getAllLinearVersions` in the current workspace (see §3.1.8.2 *Current Session and Workspace*), or `null` if no such direct successor exists. Note that under simple versioning the behavior of this method is equivalent to getting the unique direct successor (if any) of this version.

15.2.1.6 Frozen Node

The frozen node of a version is access with

```
Node Version.getFrozenNode().
```

Under simple versioning without in-content version store the frozen node has no parent and therefore methods that depend on a node being within the workspace tree (`Item.getPath()`, `Item.getParent()`, etc.) throw `RepositoryException`. Under full versioning a frozen node is the child of an `nt:version` within the in-content version store and so has all the characteristics of a normal node.

15.2.1.7 Version Extends Node

The `Version` interface extends `Node`. Under simple versioning, however, versions are not represented by nodes in content, consequently the inherited methods, other than `Item.getName()` (see §15.2.1.1 *Version Name*), are not required to function. These methods may throw a `RepositoryException`. Under full versioning the methods of `Version` inherited from `Node` function on the actual node in content that backs that version (see §3.13.3.1 *nt:version*).

15.2.2 Read-Only on Check-In

When a versionable node is checked in, it and its subgraph become *read-only.* The effect of read-only status on a node depends on the on-parent-version (OPV) status of each of its child items.

When a node `N` becomes read-only:

- No property of `N` can be added, removed or have its value changed *unless* it has an *on-parent-version* setting of `IGNORE`.
- No child node of `N` can be added or removed *unless* it has an *on-parent-version* setting of `IGNORE`.
- Every existing child node of `N` becomes read-only unless it has an *on-parent-version* setting of `IGNORE` or has an *on-parent-version* setting of `VERSION` and is itself versionable.

These restrictions apply to all methods with the exception of `VersionManager.restore, VersionManager.restoreByLabel` (see §15.7 *Restoring a Version*), `VersionManager.merge` (see §15.9 *Merge*) and `Node.update` (see §10.8.3 *Updating Nodes Across Workspaces*). These operations do not respect checked-in status.

Note that `remove` of a read-only node is possible, as long as its parent is not read-only, since removal is an alteration of the parent node.

15.3 Check-Out

A checked-in node is checked-out using

```
void VersionManager.checkout(String absPath),
```

where `absPath` is the absolute path of the node.

The checked-out state indicates to the repository and other clients that the latest version of `N` is "being worked on" and will typically be checked-in again at some point in the future, thus creating a new version.

On `checkout` of a node `N`:

- If `N` is already checked-out, this method has no effect.
- If `N` is not versionable, an `UnsupportedRepositoryOperationException` is thrown.

Otherwise,

- The `jcr:isCheckedOut` property of `N` is set to `true`.
- `N` and all nodes and properties in the subgraph of `N` lose their read-only status.
- Under full versioning, the current value of the `jcr:baseVersion` property of `N` is copied to the `jcr:predecessors` property of `N`.

This method is a workspace-write. There is no need to call `save`.

15.3.1.1 Testing for Checked-Out Status

Only the actual versionable node has a `jcr:isCheckedOut` property, however, the checked-in read-only effect extends into the subgraph of the versionable node (see §15.2.2 *Read-Only on Check-In*). The method

```
boolean VersionManager.isCheckedOut(String absPath)
```

returns `false` if the node at `absPath` is read-only due to a check-in operation. The method returns `false` otherwise.

Alternatively, the method

```
boolean Node.isCheckedOut()
```

can also be used directly on the node in question.

15.3.2 Checkpoint

The method

```
Version VersionManager.checkpoint(String absPath)
```

is a shortcut for `checkin` followed immediately by `checkout`.

15.4 Version Labels

A version label is a JCR name (see §3.2 *Names*) associated with a version. A version may have zero or more labels. Within a given version history, a particular label may appear a maximum of once. Labels are typically used to add application-level information to a stored version.

Under simple versioning labels are added, accessed and removed only through the version-label-specific API.

Under full versioning version labels are also exposed in content. Each `nt:versionHistory` node has a subnode called `jcr:versionLabels` of type `nt:versionLabels`:

15.4.1.1 nt:versionLabels

```
[nt:versionLabels]
  - * (REFERENCE) protected ABORT
      < 'nt:version'
```

Each version label is stored as a `REFERENCE` property whose name is the label name and whose target is the `nt:version` node within the `nt:versionHistory` to which the label applies. Dereferencing a label property is equivalent to calling `VersionHistory.getVersionByLabel`.

15.4.1.2 Adding a Version Label

The method

```
void VersionHistory.
    addVersionLabel(String versionName,
```

```
                String label,
                boolean moveLabel).
```

adds the specified `label` to the version with the specified `versionName`. The `label` must be a JCR name in either qualified or expanded form and therefore must conform to the syntax restrictions that apply to such names. In particular a colon (":") should not be used unless it is intended as a prefix delimiter in a qualified name (see §3.2.5 *Lexical Form of JCR Names*).

In a full versioning system, `VersionHistory.addVersionLabel` adds the appropriate `REFERENCE` to the `nt:versionLabels` node. The addition of a label is a workspace-write and therefore does not require a `save`.

If the specified label is already assigned to a version in this history and `moveLabel` is `true` then the label is removed from its current location and added to the version with the specified `versionName`. If `moveLabel` is `false`, then an attempt to add a label that already exists in this version history will throw a `LabelExistsVersionException`.

15.4.1.3 Testing for a Version Label

The method

```
boolean VersionHistory.hasVersionLabel(String label)
```

returns `true` if any version in the version history has the given `label`. The method

```
boolean VersionHistory.hasVersionLabel(Version version,
                                       String label)
```

returns `true` if the specified `version` has the specified `label`.

15.4.1.4 Getting Version Labels

The method

```
String[] VersionHistory.getVersionLabels()
```

returns all the version labels on all the versions in the version history. The method

```
String[] VersionHistory.getVersionLabels(Version version)
```

returns all version labels on the specified `version`.

15.4.1.5 Removing a Version Label

The method

```
void VersionHistory.removeVersionLabel(String label)
```

removes the specified `label` from this version history.

In a full versioning system, `VersionHistory.removeVersionLabel` removes the appropriate `REFERENCE` from the `nt:versionLabels`. The change is a workspace-write and therefore does not require a `save`.

15.5 Searching Version Histories

In simple versioning, version histories are not searchable from within the JCR API. In order to make version histories searchable under JCR, version storage must be exposed in content. Since simple versioning repositories *may* expose version storage (it is simply not *required*), searchable versions are effectively an optional extension of simple versioning (see §3.13.7 *Version Storage* and §6 *Query*).

Under full versioning, the exposure of version storage as content in the workspace allows the stored versions and their associated version meta-data to be searched or traversed just like any other part of the workspace.

15.6 Retrieving Version Storage Nodes

When an `nt:versionHistory` or `nt:version` node is acquired through a query or directly through a `getNode`, the actual Java type of the returned object must be `VersionHistory` (in the case `nt:versionHistory` nodes) or `Version` (in the case of `nt:version` nodes). This allows the application to cast the returned object to either `Version` or `VersionHistory` and use it in methods that take those types.

15.7 Restoring a Version

Restoring a versionable node to the state recorded in an earlier version can be done with

```
void VersionManager.restore (Version version,
                             boolean removeExisting).
```

Given a version `V` and a boolean flag `B`, and letting `N` be the versionable node in this workspace of which `V` is a version and `F` be the frozen node of `V`, on `restore(V, B)`, if `N` has unsaved changes pending, an `InvalidItemStateException` is thrown, otherwise:

15.7.1 Simple vs. Full Versioning Before Restore

Under simple versioning, if `N` is checked-in then it is automatically checked-out before the `restore` is performed.

Under full versioning the `restore` methods work regardless of whether the node in question is checked-out or checked-in.

Under both simple and full versioning, the changes are made through workspace-write and therefore do not require `save`.

15.7.2 Restoring Type and Identifier

The primary type, mixin types and identifier of `N` are set as follows:

- The jcr:primaryType property of N (and, semantically, the actual primary node type of N) is set to the value recorded in the jcr:frozenPrimaryType of F.
- The jcr:mixinTypes property of N (and, semantically, the actual mixin node types of N) is set to the value(s) recorded in the jcr:frozenMixinTypes of F.
- The jcr:uuid property of N (and, semantically, the actual identifier of N) is set to the value recorded in the jcr:frozenUuid of F.

15.7.3 Restoring Properties

For each property P present on F (other than jcr:frozenPrimaryType, jcr:frozenMixinTypes and jcr:frozenUuid):

- If P has an OPV of COPY or VERSION then F/P is copied to N/P, replacing any existing N/P.
- F will never have a property with an OPV of IGNORE, INITIALIZE, COMPUTE or ABORT (see §15.2 *Check-In: Creating a Version*).

For each property P present on N but not on F:

- If P has an OPV of COPY, VERSION or ABORT then N/P is removed. Note that while a node with a child item of OPV ABORT cannot be versioned, it is legal for a previously versioned node to have such a child item added to it and then for it to be restored to the state that it had before that item was added, as this step indicates.
- If P has an OPV of IGNORE then no change is made to N/P.
- If P has an OPV of INITIALIZE then, if N/P has a default value (either defined in the node type of N or implementation-defined) its value is changed to that default value. If N/P has no default value then it is left unchanged.
- If P has an OPV of COMPUTE then the value of N/P may be changed according to an implementation-specific mechanism.

15.7.4 Identifier collision

An identifier collision occurs when a node exists outside the subgraph rooted at A with the same identifier as a node that would be introduced by the restore operation. The result in such a case is governed by the removeExisting flag. If removeExisting is true, then the incoming node takes precedence, and the existing node (and its subgraph) is removed (if possible; otherwise a RepositoryException is thrown). If removeExisting is false, then an ItemExistsException is thrown and no changes are made.

15.7.5 Chained Versions on Restore

Each child node C of N where C has an OPV of VERSION and C is mix:versionable, is represented in F not as a copy of N/C but as special node containing a reference to the version history of C. On restore, the following occurs.

- If the workspace currently has an already existing node corresponding to C's version history and the removeExisting flag of the restore is set to true, then that instance of C becomes the child of the restored N.
- If the workspace currently has an already existing node corresponding to C's version history and the removeExisting flag of the restore is set to false then an ItemExistsException is thrown.
- If the workspace does not have an instance of C then one is restored from C's version history:
    - If the restore was initiated through a restoreByLabel where L is the specified label and there is a version of C with the label L then that version is restored.
    - If the version history of C does not contain a version with the label L or the restore was initiated by a method call that does not specify a label then the workspace in which the restore is being performed will determine which particular version of C will be restored. This determination depends on the configuration of the workspace and is outside the scope of this specification.

15.7.6 Restoring Child Nodes

For each child node C present on F:

- If C has an OPV of COPY or VERSION:
    - B is true, then F/C and its subgraph is copied to N/C, replacing any existing N/C and its subgraph *and* any node in the workspace with the same identifier as C or a node in the subgraph of C is removed.
    - B is false, then F/C and its subgraph is copied to N/C, replacing any existing N/C and its subgraph *unless* there exists a node in the workspace with the same identifier as C, or a node in the subgraph of C, in which case an ItemExistsException is thrown , all changes made by the restore are rolled back leaving N unchanged.

      Under full versioning each child node C of N where C has an OPV of VERSION and C is versionable, is represented in F not as a copy of N/C but as special node of type nt:versionedChild containing a reference to the version history of C. On restore, N/C in the workspace is replaced by a version of C. The determination of which version of C to use is implementation-dependent (see §15.7.5 *Chained Versions on Restore*).

In a repository that supports orderable child nodes, the relative ordering of the set of child nodes C that are copied from F is preserved.

- `F` will never have a child node with an OPV of `IGNORE`, `INITIALIZE`, `COMPUTE` or `ABORT` (see §15.2 *Check-In: Creating a Version*).

For each child node `C` present on `N` but not on `F`:

- If `C` has an OPV of `COPY`, `VERSION` or `ABORT` then `N/C` is removed. Note that while a node with a child item of OPV `ABORT` cannot be versioned, it is legal for a previously versioned node to have such a child item added to it and then for it to be restored to the state that it had before that item was added, as this step indicates.
- If `C` has an OPV of `IGNORE` then no change is made to `N/C`.
- If `C` has an OPV of `INITIALIZE` then `N/C` is re-initialized as if it were newly created, as defined in its node type.
- If `C` has an OPV of `COMPUTE` then `N/C` may be re-initialized according to an implementation-specific mechanism.

15.7.7 Simple vs. Full Versioning after Restore

Under simple versioning `N` is automatically checked-in.

Under full versioning the `jcr:isCheckedOut` property of `N` is set to `false` (though the other elements of a check-in are not performed). Additionally, the `jcr:baseVersion` property of `N` is set to `V`. Note that after the next check-out (see §15.3 *Check-Out*) and subsequent check-in of `N` the version `V` will acquire an additional direct successor, forming a branch.

15.7.8 Restore Variants

The method

```
void VersionManager.restore(String absPath,
                            Version version,
                            boolean removeExisting)
```

takes the `Version` object and a target path. This method only works in cases where no node exists at `absPath`. It is used to restore nodes that have been removed or to introduce new subgraphs into a workspace based on state stored in a version.

15.7.8.1 Restore by Version Name

The method

```
void VersionManager.restore(String absPath,
                            String versionName,
                            boolean removeExisting)
```

takes a version name instead of the actual `Version` object. The version to be restored is identified by name from within the version history of the node at `absPath`. This method requires that the node at `absPath` exist and be a versionable node.

15.7.8.2 Restore by Version Label

The method

```
void VersionManager.restoreByLabel(String absPath,
                                   String versionLabel,
                                   boolean removeExisting)
```

takes a version label instead of a `Version` object (see §15.2.1 *Version Object*). The version to be restored is identified by label from within the version history of the node at `absPath`. This method requires that the node at `absPath` exist and be a versionable node.

15.7.8.3 Restoring a Group of Versions

The method

```
void VersionManager.
  restore(Version[] versions, boolean removeExisting)
```

is used to simultaneously restore multiple versions. This may be necessary in cases where sequential restoration is impossible due to a cycle of `REFERENCE` properties in the nodes to be restored.

15.8 Removing a Version

In some implementations it may be possible to remove versions from within a version history using `VersionHistory.removeVersion`. In such cases the version graph must be automatically repaired so that the direct successor of the removed version becomes the direct successor of the direct predecessor of the removed version.

The method

```
void VersionHistory.removeVersion(String versionName)
```

removes the named version from this version history and automatically repairs the version graph. If the version to be removed is `V`, `V`'s direct predecessor set is `P` and `V`'s direct successor set is `S`, then the version graph is repaired s follows:

- For each member of `P`, remove the reference to `V` from its direct successor list and add references to each member of `S`.
- For each member of `S`, remove the reference to `V` from its direct predecessor list and add references to each member of `P`.

This change is a workspace-write; there is no need to call `save`.

15.9 Merge

The method

```
NodeIterator VersionManager.
    merge(String absPath, String srcWorkspace,
          boolean bestEffort, boolean isShallow)
```

performs the first step in a *merge* of two corresponding nodes:

The `merge` method can be called on a versionable or non-versionable node.

Like `update`, `merge` does not respect the checked-in status of nodes. A `merge` may change a node even if it is currently checked-in.

If `this` node (the one on which `merge` is called) does not have a corresponding node in the indicated workspace, then the `merge` method returns quietly and no changes are made.

If `isShallow` is `true` and `this` node, despite having a corresponding node, is nevertheless non-versionable then the merge method also returns quietly and no changes are made.

Otherwise, the following happens:

If `isShallow` is `true` then a merge test is performed on `this` node, call it `N`. If `isShallow` is `false` then a merge test is performed recursively on each versionable node, `N` within the subgraph rooted at `this` node.

The merge test is performed by comparing `N` with its corresponding node in `srcWorkspace`, call it `N'`.

The merge test is done by comparing *the base version of* `N` (call it `V`) and *the base version of* `N'` (call it `V'`).

For any versionable node `N` there are three possible outcomes of the merge test: *update*, *leave* or *failed*.

If `N` does not have a corresponding node then the merge result for `N` is *leave*.

If `N` is currently checked-in then:

- If `V'` is an eventual successor of `V`, then the merge result for `N` is *update*.
- If `V'` is an eventual predecessor of `V` or if `V` and `V'` are identical (i.e., are actually the same version), then the merge result for `N` is *leave*.
- If `V` is neither an eventual successor of, eventual predecessor of, nor identical with `V'`, then the merge result for `N` is *failed*. This is the case where `N` and `N'` represent divergent branches of the version graph.

If `N` is currently checked-out then:

- If `V'` is an eventual predecessor of `V` or if `V` and `V'` are identical (i.e., are actually the same version), then the merge result for `N` is *leave*.
- If any other relationship holds between `V` and `V'`, then the merge result for `N` is *fail*.

If `bestEffort` is `false` then the first time a merge result of *fail* occurs, the entire merge operation on this subgraph is aborted, no changes are made to the subgraph and a `MergeException` is thrown. If no merge result of *fail* occurs then:

- Each versionable node `N` with result *update* is updated to reflect the state of `N'`. The state of a node in this context refers to its set of properties and child node links.
- Each versionable node `N` with result *leave* is left unchanged, *unless `N` is the child of a node with status* update *and `N` does not have a corresponding node in `srcWorkspace`, in which case it is removed.*

If `bestEffort` is `true` then:

- Each versionable node `N` with result *update* is updated to reflect the state of `N'`. The state of a node in this context refers to its set of properties and child node links.
- Each versionable node `N` with result *leave* is left unchanged, unless `N` is the child of a node with status *update* and `N` does not have a corresponding node in `srcWorkspace`. I such a case, `N` is removed.
- Each versionable node `N` with result *failed* is left unchanged except that the identifier of `V'` (which is, in some sense, the "offending" version; the one that caused the merge to fail on that `N`) is added to the multi-value `REFERENCE` property `jcr:mergeFailed` of `N`. If the identifier of `V'` is already in `jcr:mergeFailed`, it is not added again. The `jcr:mergeFailed` property never contains repeated references to the same version. If the `jcr:mergeFailed` property does not yet exist then it is created. If present, the `jcr:mergeFailed` property will always contain at least one value. If not present on a node, this indicates that no merge failure has occurred on that node. Note that the presence of this property on a node will in any case prevent it from being checked-in because the `OnParentVersion` setting of `jcr:mergeFailed` is `ABORT`.
- This property can later be used by the application to find those nodes in the subgraph that have failed to merge and thus require special attention (see §15.9.2 *Merging Branches*). This property is multi-valued so that a record of successive failed merges can be kept.

In either case, (regardless of whether `bestEffort` is `true` or `false`) for each non-versionable node (including both referenceable and non-referenceable), if the merge result of its *nearest versionable ancestor* is *update,* or if it has *no versionable ancestor*, then it is updated to reflect the state of its corresponding node. Otherwise, it is left unchanged. The definition of corresponding node in this context is the same as usual: the match is done by identifier.

Note that a deep `merge` performed on a subgraph with no versionable nodes at all (or indeed in a repository that does not support versioning in the first place) will be equivalent to an update.

The `merge` method returns a `NodeIterator` over all versionable nodes in the subgraph that received a merge result of fail.

Note that if `bestEffort` is `false`, then merge will either return an empty iterator (since no merge failure occurred) or throw a `MergeException` (on the first merge failure that was encountered).

If `bestEffort` is `true`, then the iterator will contain all nodes that received a fail during the course of this merge operation.

All changes made by `merge` are workspace-write, and therefore this method does not require a `save`.

15.9.1 Merge Algorithm

The above declarative description can also be expressed in pseudo-code as follows:

let `ws'` be the workspace against which the `merge` is done.
let `bestEffort` be the flag passed to `merge`.
let `isShallow` be the flag passed to `merge`.
let `failedset` be a set of identifiers, initially empty.
let `startnode` be the node on which `merge` was called.
`domerge(startnode)`.
return the nodes with the identifiers in `failedset`.

```
domerge(n)
   let n' be the corresponding node of n in ws'.
   if no such n' doleave(n).
   else if n is not versionable doupdate(n, n').
   else if n' is not versionable doleave(n).
   let v be base version of n.
   let v' be base version of n'.
   if v' is an eventual successor of v and
      n is not checked-in doupdate(n, n').
   else if v is equal to or an eventual predecessor of v' doleave(n).
   else dofail(n, v').

dofail(n, v')
   if bestEffort = false throw MergeException.
   else add identifier of v' (if not already present) to the
      jcr:mergeFailed property of n,
      add identifier of n to failedset,
       if isShallow = false
          for each versionable child node c of n domerge(c).

doleave(n)
   if isShallow = false
      for each child node c of n domerge(c).

doupdate(n, n')
   replace set of properties of n with those of n'.
   let S be the set of child nodes of n.
   let S' be the set of child nodes of n'.
   judging by the name of the child node:
```

let C be the set of nodes in S and in S'
let D be the set of nodes in S but not in S'.
let D' be the set of nodes in S' but not in S.
remove from n all child nodes in D.
for each child node of n' in D' copy it (and its subgraph) to n
as a new child node (if an incoming node has the same
identifier as a node already existing in this workspace,
the already existing node is removed).
for each child node m of n in C domerge(m).

15.9.2 Merging Branches

When a merge test on a node N fails, this indicates that the two base versions V and V' are on separate branches of the version graph. Consequently, determining the result of the merge is not simply a matter of determining which version is the eventual successor of the other in terms of version history. Instead, the subgraph of N' must be merged into the subgraph of N according to some domain specific criteria which must be performed at the application level, for example, through a merge tool provided to the user.

The jcr:mergeFailed property is used to tag nodes that fail the merge test so that an application can find them and deal appropriately with them. The jcr:mergeFailed property is multi-valued so that information about merge failures is not lost if more than one successive merge is attempted before being dealt with by the application.

After the subgraph of N' is merged into N, the application must also merge the two branches of the version graph. This is done by calling N.doneMerge(V') where V' is retrieved by following the reference stored in the jcr:mergeFailed property of N. This has the effect of moving the reference-to-V' from the jcr:mergeFailed property of N to its jcr:predecessors property.

If, on the other hand, the application chooses not to join the two branches, then cancelMerge(V') is performed. This has the effect of removing the reference to V' from the jcr:mergeFailed property of N without adding it to jcr:predecessors.

Once the last reference in jcr:mergeFailed has been either moved to jcr:predecessors (with doneMerge) or just removed from jcr:mergeFailed (with cancelMerge) the jcr:mergeFailed property is automatically removed, thus enabling this node to be checked-in, creating a new version (note that before the jcr:mergeFailed is removed, its OnParentVersion setting of ABORT prevents check-in). This new version will have a direct predecessor connection to each version for which doneMerge was called, thus joining those branches of the version graph.

All changes made by doneMerge and cancelMerge are workspace-write and therefore do not require save.

15.9.3 Merging Activities

In repositories that support activities (see §15.12 *Activities*) merging an activity into another workspace is done with the method

```
VersionManager.merge(Node activityNode).
```

(see §15.12.7 *Merging an Activity into Another Workspace*).

15.10 Serialization of Version Storage

Serialization of version information can be done in the same way as normal serialization by serializing the subgraph below `/jcr:system/jcr:versionStorage`. The special status of these nodes with respect to versioning is transparent to the serialization mechanism.

The serialized content of the source version storage can be imported as "normal" content on the target repository, but it will not actually be interpreted and integrated into the repository as version storage data unless it is integrated into or used to replace the target repository's own version storage.

Methods for doing this kind of "behind the scenes" alteration to an existing version storage (whether based on the serialized version storage of another repository, or otherwise) are beyond the scope of this specification.

15.11 Versioning within a Transaction

In a repository that supports both versioning and transactions, all versioning operations must be fully transactional, meaning that they can be bracketed within a transaction and rolled-back just like any other set of operations.

15.12 Activities

Activities provide a way of grouping together a set of logically related changes performed in a workspace and then later merging this set of changes into another workspace.

Before starting to make a particular set of changes, the user sets the *current activity*. Each subsequent `checkout` made within the scope of that activity will associate that activity with that checked-out versionable, and will create a version that is tagged with the specified activity when that versionable is subsequently checked-in.

Abstractly, therefore, an activity is a set of changes that produce new versions. However, if that set includes changes that produce more than one version within a particular version history, then those versions must all be on the same line of descent, that is, there must be a non-branching sequence of direct successor relationships beginning at the root version of the version history that reaches every version in the set. This ensures that there is always at most one "latest" version that contains all changes in a given version history for a given activity.

15.12.1 Support for Activities

Support for activities is an optional addition to the full versioning feature. An implementation that supports versioning *may* support activities.

Whether a particular implementation supports activities can be determined by querying the repository descriptor table with

```
Repository.OPTION_ACTIVITIES_SUPPORTED.
```

A return value of `true` indicates support for activities (see §24.2 *Repository Descriptors*).

15.12.2 Related Node Types

Activities are represented by nodes of node type `nt:activity`:

```
[nt:activity] > mix:referenceable
  - jcr:activityTitle (STRING) mandatory autocreated protected
```

The relationship between version and activity is modeled by the property `jcr:activity`, in the `mix:versionable` and `nt:version` node types:

```
[mix:versionable] > mix:simpleVersionable, mix:referenceable
  mixin
  - jcr:versionHistory (REFERENCE) mandatory protected IGNORE
      < 'nt:versionHistory'
  - jcr:baseVersion (REFERENCE) mandatory protected IGNORE
      < 'nt:version'
  - jcr:predecessors (REFERENCE) mandatory protected multiple
      IGNORE < 'nt:version'
  - jcr:mergeFailed (REFERENCE) protected multiple ABORT
      < 'nt:version'
  - jcr:activity (REFERENCE) protected IGNORE < 'nt:activity'
  - jcr:configuration (REFERENCE) protected IGNORE
      < 'nt:configuration'
```

```
[nt:version] > mix:referenceable
  - jcr:created (DATE) mandatory autocreated protected
      ABORT
  - jcr:predecessors (REFERENCE) protected multiple ABORT
      < 'nt:version'
  - jcr:successors (REFERENCE) protected multiple ABORT
      < 'nt:version'
  - jcr:activity (REFERENCE) protected ABORT < 'nt:activity'
  + jcr:frozenNode (nt:frozenNode) protected ABORT
```

15.12.3 Activity Storage

Activities are persisted as nodes of type `nt:activity` under system-generated node names in activity storage below `/jcr:system/jcr:activities`.

The organization of this subgraph is left up to the implementation (for example, there may be intervening nodes structuring the activity storage).

Similar to the `/jcr:system/jcr:versionStorage` subgraph, the activity storage is a single repository wide store, but is reflected into each workspace. However access control may be employed so that different sessions see different parts of the tree.

15.12.3.1 Activity Storage is Read-Only

The activity storage subgraph is not writable through the core write methods (see §10.2 *Core Write Methods*). It can only be altered through the activity-specific write methods described in this section.

15.12.4 Creating an Activity

Activities are created using:

```
Node VersionManager.createActivity(String title)
```

creates a new `nt:activity` node at an implementation-determined location in `the /jcr:system/jcr:activities` subgraph and returns it.

The name of the `nt:activity` node is automatically generated by the repository. The repository *may* use the `title` parameter as a hint to give a value to the `jcr:activityTitle` property of the new node. The new node addition is dispatched immediately and therefore does not require a `save`.

15.12.5 Setting the Current Activity

```
Node VersionManager.setActivity(Node activity)
```

is called by the user to set the current activity on the session by specifying a previously created `nt:activity` node. Changing the current activity is done by calling `setActivity` again. Cancelling the current activity is done by calling `setActivity(null)` and results in the session having no current activity. The method returns the *previously set* `nt:activity` node or `null` if no such node exists.

Assuming,

- the current activity of session `S` is represented by node `A` and
- node `N` is a versionable node with version history `H`,

then each `checkout` of node `N` made through `S` while `A` is in effect causes the following:

- If there exists another workspace with node `N'` where `N'` also has version history `H`, `N'` is checked out and the `jcr:activity` property of `N'` references `A`, then the check-out fails with an `ActivityViolationException` indicating which workspace currently has the check-out.
- If there is a version in `H` that is not an eventual predecessor of `N` but whose `jcr:activity` references `A`, then the check-out fails with an `ActivityViolationException`.
- Otherwise, the `jcr:activity` property of `N` is set to reference `A` and when `N` is subsequently checked in, the `jcr:activity` property of the new version is set to reference `A`, and the `jcr:activity` property of `N` is removed.

15.12.6 Getting the Current Activity

The method

```
Node VersionManager.getActivity()
```

returns the node representing the current activity or `null` if there is no current activity.

15.12.7 Merging an Activity into Another Workspace

Once an activity has been completed, the changes that it records can be imported into another workspace. This is done with a variant of the merge method:

```
NodeIterator VersionManager.merge(Node activityNode)
```

This method merges the changes that were made under the specified activity into `this` workspace.

An activity `A` will be associated with a set of versions through the `jcr:activity` reference of each version node in the set. We call each such associated version a member of `A`.

For each version history `H` that contains one or more members of `A`, one such member will be the latest member of `A` in `H`. The latest member of `A` in `H` is the version in `H` that is a member of `A` and that has no eventual successor versions that are also members of `A`.

The set of versions that are the latest members of `A` in their respective version histories is called the change set of `A`. It fully describes the changes made under the activity `A`.

This method performs a shallow merge, with `bestEffort` equal to `true`, into this workspace (see §15.9 *Merge*) of each version in the change set of the activity specified by `activityNode`. If there is no corresponding node in this workspace for a given member of the change set, that member is ignored.

This method returns a `NodeIterator` over all versionable nodes in the subgraph that received a merge result of *fail* (see §15.9.1 *Merge Algorithm*).

All changes made through this method are workspace-write and therefore do not require `save`.

15.12.8 Removing an Activity

Some repositories may support

```
void VersionManager.removeActivity(Node activityNode)
```

which removes the specified activity node from the activity storage and automatically removes all `REFERENCE` properties referring to that node in all workspaces, with the exception of `REFERENCE` properties in version storage. The existence of a `REFERENCE` to the activity node from within version storage will cause an exception to be thrown. Changes made through this method are workspace-write and therefore do not require `save`.

15.13 Configurations and Baselines

A *configuration* is the subgraph of a specifically designated versionable node (called the *configuration root node*) in a workspace, minus any parts of that subgraph that are themselves designated as configurations. A *baseline* is the state of a configuration at some point in time, recorded in version storage as a version object.

15.13.1 Support for Configurations and Baselines

Support for configurations and baselines is an optional addition to the full versioning feature. An implementation that supports full versioning *may* support configurations and baselines. Whether a particular implementation supports configurations and baselines can be determined by querying the repository descriptor table with

```
Repository.OPTION_BASELINES_SUPPORTED.
```

A return value of `true` indicates support for configurations and baselines (see §24.2 *Repository Descriptors*).

15.13.2 Configuration Proxy Nodes

Each configuration in a given workspace is represented by a distinct *proxy node* of type `nt:configuration` located in *configuration storage* within the same workspace under `/jcr:system/jcr:configurations/`. The configuration storage in a particular workspace is specific to that workspace. It is not a common repository-wide store mirrored into each workspace, as is the case with version storage.

The proxy node of a configuration is used to perform certain operations on that configuration. In particular, version operations performed on the proxy node act not only on that node itself but also on the configuration it represents, as a whole. Creating a baseline, for example, is done by performing a `checkin` on a configuration's proxy node.

15.13.2.1 nt:configuration

Every configuration proxy node is of type `nt:configuration`:

```
[nt:configuration] > mix:versionable
 - jcr:root (REFERENCE) mandatory autocreated protected
```

This node type is a subtype of `mix:versionable` and adds a single property, the `REFERENCE` property `jcr:root`, which points to the root node of the configuration that this proxy represents.

Since every configuration proxy node is versionable, each has a version history. The versions within this history store state information about configuration represented by the proxy node, in addition to information about the proxy node itself.

15.13.2.2 Structure of Configuration Storage

The organization of configuration storage is left up to the implementation (for example, there may be intervening nodes structuring the storage). It is expected that access control will also be employed to ensure that only sessions with appropriate authorization may create or have access to particular configurations.

15.13.3 Creating a Configuration

A configuration is created by designating a `mix:versionable` node `N` in the workspace as a configuration root node. This is done by calling

```
Node VersionManager.
    createConfiguration(String absPath)
```

where `absPath` is the path of `N`.

On creation of a new configuration with root `N`, a new proxy node `C`, of type `nt:configuration`, is created under `/jcr:system/jcr:configurations/` and a new version history H_C is created for `C` with a root version B_0. Note that H_C is also called a *baseline history* and its contained versions, including B_0, are called *baselines*. The baselines within H_C store not only the state of `C` but also the state of the configuration represented by `C` (see 15.13.4.1 *Creating a Baseline*).

The properties of `C` and `N` are initialized as follows:

- `N/jcr:configuration` points to `C`.
- `C/jcr:root` points to `N`.
- `C/jcr:versionHistory` points to H_C.
- `C/jcr:baseVersion` points to B_0.

The `createConfiguration` call will fail if

- the node at `absPath` is not `mix:versionable`.
- the node at `absPath` is already a configuration root (i.e., if it already has a `jcr:configuration` property).
- There exists in the subgraph at `N` a versionable node that has never been checked-in (i.e., one whose base version is still its root version).

The `createConfiguration` method is workspace-write. Therefore the changes it makes are dispatched immediately and a `save` is not required.

15.13.4 Baselines

A baseline records the state of a configuration at some particular time and is represented by a version (i.e., an `nt:version` node) of the `nt:configuration` node in question.

15.13.4.1 Creating a Baseline

A baseline is created by performing a `checkin` on a configuration proxy node (i.e., a node of type `nt:configuration` found in configuration storage). Note that since `nt:configuration` is subtype of `mix:versionable`, a configuration node will have its own version history, *distinct from the version history of its configuration root node*.

On `checkin` of the configuration proxy node `C`:

- The state of the `C` is recorded in a new baseline `B` just as it would be in a normal version.
- In addition, the current base version of every versionable node in the configuration is also recorded.

How the configuration state information is stored is up to the repository. It need not be stored as content in the substructure of the `nt:version` node. For example, some repositories are likely to have some efficient internal mechanisms involving lists of identifiers, possibly stated as a delta against the direct predecessor baseline. The only requirement is that baselines be "restoreable".

15.13.4.2 Restoring a Baseline

Using the method

```
VersionManager.restore(Version version, boolean removeExisting)
```

where `version` is a baseline `Version` object, `C` is the `nt:configuration` node whose version history contains `version` and `N` is the configuration root node pointed to by `C/jcr:root`:

- The state of `C` is restored to the state recorded in `version` and `C/jcr:baseVersion` is set to point to `version` (as in the restore of any normal version).
- Each versionable node `M` in the subgraph below and including `N` is restored to the state recorded in `V` where `V` is the version of `M` in `M`'s version history that was recorded in `B` (i.e., the base version of `M` at the time `B` was created).

The `removeExisting` parameter behaves just as in a normal restore expect that that it applies to all nodes restored below `N`. The same behavior applies for the multi-version signature of restore,

```
VersionManager.restore(Version[] versions,
                       boolean removeExisting)
```

except that multiple baselines may be restored simultaneously.

15.13.4.3 Creating a Configuration from an Existing Baseline

The method

```
VersionManager.restore(String absPath,
                       Version version,
                       boolean removeExisting),
```

where `version` is a baseline `Version` object and `absPath` is a path to a location where no node exists but which has a suitable parent node, creates a new configuration at `absPath` by restoring the baseline `version`. A configuration proxy node `C` with `C/jcr:root` pointing to the root node of the new configuration at `absPath` is automatically created in configuration storage. If a node already exists at `absPath`, the method fails. The variant signatures

```
VersionManager.restore(String absPath,
                       String versionName,
                       boolean removeExisting),
```

and

```
VersionManager.restoreByLabel(String absPath,
                              String label,
                              boolean removeExisting),
```

work identically except that the baseline to be restored is identified either by name or by label instead of being passed in as a `Version` object.

16 Access Control Management

A repository may support *access control management*, enabling the following:

- Privilege discovery: Determining the privileges that a user has in relation to a node.
- Assigning access control policies: Setting the privileges that a user has in relation to a node using access control policies specific to the implementation.

Whether a particular implementation supports access control can be determined by querying the repository descriptor table with

```
Repository.OPTION_ACCESS_CONTROL_SUPPORTED.
```

A return value of `true` indicates support (see §24.2 *Repository Descriptors*).

16.1 Access Control Manager

Access control is exposed through a

```
javax.jcr.security.AccessControlManager
```

acquired from the `Session` using

```
AccessControlManager Session.getAccessControlManager().
```

16.2 Privilege Discovery

A privilege represents the ability to perform a particular set of operations on a node. Each privilege is identified by a JCR name.

JCR defines a set of standard privileges within the `Privilege` interface. An implementation may add additional privileges, using an appropriate implementation-specific namespace for their names.

16.2.1 Aggregate Privileges

A privilege may be an *aggregate privilege*. Aggregate privileges are sets of other privileges. Granting, denying, or testing an aggregate privilege is equivalent to individually granting, denying, or testing each privilege it contains. The privileges contained by an aggregate privilege may themselves be aggregate privileges if the resulting privilege graph is acyclic.

16.2.2 Abstract Privileges

A privilege may be an *abstract privilege*. Abstract privileges cannot themselves be granted or denied, but can be individually tested and can be composed into aggregate privileges which are granted or denied.

Abstract privileges facilitate application interoperability against repositories supporting different privilege granularities. For example, consider aggregate privilege *p* containing privileges *p1* and *p2*. In repository A, *p1* and *p2* are not abstract and can therefore be individually granted, whereas in repository B both

p1 and *p2* are abstract and cannot be individually granted. For both repositories, however, an application can test whether a user has privilege *p1*, even though in repository B, *p1* can only be acquired through non-abstract privilege *p*.

A privilege can be both aggregate and abstract.

16.2.3 Standard Privileges

A repository must support the following standard privileges identified by the string constants of `javax.jcr.security.Privilege`:

- `jcr:read`: The privilege to retrieve a node and get its properties and their values.
- `jcr:modifyProperties`: The privilege to create, remove and modify the values of the properties of a node.
- `jcr:addChildNodes`: The privilege to create child nodes of a node.
- `jcr:removeNode`: The privilege to remove a node.
- `jcr:removeChildNodes`: The privilege to remove child nodes of a node.

  In order to actually remove a node requires `jcr:removeNode` on that node and `jcr:removeChildNodes` on the parent node. The distinction is provided in order to distinguish implementations that internally model a "remove" as a "delete" from those that model it as an "unlink". A repository that uses the "delete" model can have `jcr:removeChildNodes` in every access control policy, so that removal is effectively controlled by `jcr:removeNode`. Conversely, a repository that uses the "unlink" model can have `jcr:removeNode` in every access control policy.
- `jcr:write`: An aggregate privilege that contains:
  - `jcr:modifyProperties`
  - `jcr:addChildNodes`
  - `jcr:removeNode`
  - `jcr:removeChildNodes`
- `jcr:readAccessControl`: The privilege to read the access control settings of a node.
- `jcr:modifyAccessControl`: The privilege to modify the access control settings of a node.
- `jcr:lockManagement`: The privilege to lock and unlock a node (see §17 *Locking*).
- `jcr:versionManagement`: The privilege to perform versioning operations on a node (see §15 *Versioning*).

- `jcr:nodeTypeManagement`: The privilege to add and remove mixin node types and change the primary node type of a node (see §10.10 *Node Type Assignment*).
- `jcr:retentionManagement`: The privilege to perform retention management operations on a node (see §20 *Retention and Hold*).
- `jcr:lifecycleManagement`: The privilege to perform lifecycle operations on a node (see §18 *Lifecycle Management*).
- `jcr:all`: An aggregate privilege that contains:
    - `jcr:read`
    - `jcr:write`
    - `jcr:readAccessControl`
    - `jcr:modifyAccessControl`
    - `jcr:lockManagement`
    - `jcr:versionManagement`
    - `jcr:nodeTypeManagement`
    - `jcr:retentionManagement`
    - `jcr:lifecycleManagement`

Whether a privilege is abstract is an implementation variant, with the exception that `jcr:all` is never an abstract privilege. For example, a repository unable to separately control the abilities to add child nodes, remove child nodes, and set properties could make `jcr:modifyProperties`, `jcr:addChildNodes`, and `jcr:removeChildNodes` abstract privileges within the aggregate privilege `jcr:write`.

Similarly, whether any one of these privileges is aggregate is an implementation variant, with the exception that `jcr:write` and `jcr:all` are always aggregate privileges.

A repository should also add all implementation-defined privileges to `jcr:all`.

The standard privilege names are defined as expanded form JCR names in string constants of `javax.jcr.security.Privilege`.

16.2.4 Supported Privileges

The privileges available for a particular node can be determined through

```
Privilege[]
  AccessControlManager.
    getSupportedPrivileges(String absPath)
```

where `absPath` is the location of the node. Note that this method does not return the privileges *held* by a `Session` with respect to the specified node, but rather the

privileges *supported* by the repository with respect to that node (see §16.3.7 *Testing Privileges*).

16.2.5 Retrieving Privileges by Name

A `Privilege` object can be obtained from the `AccessControlManager` through

```
Privilege
  AccessControlManager.
    privilegeFromName(String privilegeName)
```

where `privilegeName` identifies an existing `Privilege` (see §16.3.6 *Privilege Object*). Since the privilege name is a JCR name it may be passed in either qualified or expanded form (see §3.2.6 *Use of Qualified and Expanded Names*).

16.2.6 Privilege Object

The characteristics of a `Privilege` object are exposed through the following methods:

```
String Privilege.getName()
```

returns the name of this privilege. Since the privilege name is a JCR name it must be returned in qualified form (see §3.2.6 *Use of Qualified and Expanded Names*).

```
boolean Privilege.isAbstract()
```

returns whether the privilege is abstract.

```
boolean Privilege.isAggregate()
```

returns whether the privilege is aggregate.

```
Privilege[] Privilege.getDeclaredAggregatePrivileges().
```

If this privilege is aggregate, this method returns the privileges directly contained within it. Otherwise, it returns an empty array.

```
Privilege[] Privilege.getAggregatePrivileges().
```

If this privilege is aggregate, this method returns the privileges it contains, the privileges contained by any aggregate privileges among those, and so on (i.e., the transitive closure of privileges contained by the initial privilege). Otherwise, it returns an empty array.

16.2.7 Testing Privileges

The method

```
boolean AccessControlManager.
  hasPrivileges(String absPath, Privilege[] privileges)
```

returns whether the `Session` has the specified privileges for the node at `absPath`. Testing an aggregate privilege is equivalent to testing each non-aggregate privilege among the set returned by calling `Privilege.getAggregatePrivileges()`.

The method

```
Privilege[] AccessControlManager.getPrivileges(String absPath)
```

returns the privileges the session has for absolute path `absPath`. The returned privileges are those for which `hasPrivileges` would return `true`.

The set of *privileges* held by a session with respect to a particular node are the result of

- access control policies applied using JCR (see §16.4 *Access Control Policies*),
- privilege affecting mechanisms external to JCR, if any.

The set of privileges reported by the privilege test methods reflects the current net *effect* of these mechanisms. It does not reflect unsaved access control policies.

16.3 Access Control Policies

The privileges granted to a user can be controlled by assigning *access control policies* to nodes. The content and semantics of these policies are implementation specific and may be based on any mechanism, including access control lists or role-responsibility assignments. JCR does not expose the internals of policies, nor does it provide a mechanism for defining them. However, it does provide a marker interface `AccessControlPolicy` and two derived interfaces `NamedAccessControlPolicy` and `AccessControlList` (see §16.6 *Access Control Lists*). Furthermore, JCR provides means to:

- Find which polices are available to be bound to a node.
- Bind a policy to a node.
- Get the policies bound to a given node (including transient modifications).
- Get the policies that affect access to a given node.
- Unbind a policy from a node.

In addition to these methods, any *effect* that a policy has on a node is always reflected in the information returned by the privilege discovery methods (see §16.2.7 *Testing Privileges*). Note that the *scope* of the effect of an access control policy may not be identical to the node to which that policy is bound (see §16.4.2 *Binding a Policy to a Node*).

16.3.1 Applicable Policies

```
AccessControlPolicyIterator
  AccessControlManager.getApplicablePolicies(String absPath)
```

returns a list of access control policies that are capable of being applied to the node at `absPath`. The mechanism for defining the set of policies applicable to a particular node is implementation-dependent. For a given node, the set of applicable policies available at a specific time may depend on the set of policies

bound to the node at that time. Therefore, the set returned by this method may vary between calls as policies are bound and unbound.

16.3.2 Binding a Policy to a Node

The method

```
void AccessControlManager.
  setPolicy(String absPath, AccessControlPolicy policy)
```

binds a `policy` to the node at `absPath`. The behavior of the call

```
acm.setPolicy(absPath, policy)
```

differs depending on how the `policy` object was originally acquired. If `policy` was acquired through

```
acm.getApplicablePolicies(absPath)
```

then `policy` is added to the node at `absPath`. On the other hand, if `policy` was acquired through

```
acm.getPolicies(absPath)
```

then that `policy` object (after, presumably, being altered) replaces its older version on the node at `absPath` (see §16.3.4 *Getting the Bound Policies*)

16.3.3 Binding vs. Effect

A policy is *bound* to a node upon completion of the `setPolicy` call but only *takes effect* upon `Session.save`.

16.3.4 Getting the Bound Policies

The method

```
AccessControlPolicy[]
  AccessControlManager.getPolicies(String absPath)
```

returns the policies bound to the node at `absPath`. If this method is called from the `AccessControlManager` of a `Session` which holds pending, unsaved policy bindings, then the policies returned will reflect the transient state instead of the persisted state. If there are no policies bound to the node at `absPath` through the JCR API this method returns an empty array.

16.3.5 Scope of a Policy

When an access control policy takes effect, it may affect the accessibility characteristics not only of the node to which it is bound but also of nodes elsewhere in the workspace.[20] The method

```
AccessControlPolicy[]
  AccessControlManager.getEffectivePolicies(String absPath)
```

performs a best-effort search to determine the policies in effect on the node at `absPath`.

16.3.6 Default Access Control

If a node has no effective policy assigned through the JCR API, then an implementation-specific default policy must be in effect and this policy must be returned by `AccessControlManager.getEffectivePolicies`. The default privileges for the node are determined by the implementation in accordance with this default policy.

16.3.7 Removing a Policy

The method

```
void AccessControlManager.
      removePolicy(String absPath, AccessControlPolicy policy)
```

removes the specified `AccessControlPolicy` from the node at `absPath`. An `AccessControlPolicy` can only be removed if it was previously bound to the specified node through this API. The effect of the removal only takes place upon `Session.save()`.

16.3.8 Interaction with the Transient Layer and Transactions

Changes to access control are session-write operations (see §10.1.1 *Session-Write*) and interact with the transient layer and persistent store no differently than other such operations:

- A node which has had a policy set or removed is marked as modified until the changes are saved.

---

[20] One common case is a policy that affects both its bound node and the subgraph below that node. However, any such *deepness* attribute is internal to the policy and, like any other internal characteristic of a policy, opaque to the JCR API except insofar as it is part of the human-readable name and description. Note also that, strictly speaking, a policy is not required to affect even its bound node, though such an implementation would be uncommon.

- The access control modifications can be reverted by calling `Session.refresh(false)`.
- The changes are visible to sessions other than the session making the change *no earlier than* its being dispatched (i.e., saved if outside a transaction, committed if within a transaction).
- Depending on the repository implementation, the changes may not be reflected in another session until that session reacquires the modified node (for example, by calling `Session.refresh`).

16.3.9 Access to Properties

Access to a property is controlled by the effective access control policies of its parent node.

16.3.10 Access Control Restrictions

A repository may restrict which nodes may be access controlled. For example a document-centric repository might allow only `nt:hierarchyNode` nodes to be access controlled. A repository may automatically add access control policies to a newly created node based upon an implementation-determined default.

16.3.11 Exposing Policies in Content

A repository may expose a node's access control policies as child nodes or properties. If it does so, then the add, remove and save semantics of the item must match those of the policy it represents.

16.3.12 Interaction with Protected Properties

Many features of JCR expose repository metadata as protected properties defined by mixin node types. For example, locking status is exposed by the properties `jcr:lockOwner` and `jcr:lockIsDeep` defined by `mix:lockable`. Changes to protected properties can only be made indirectly through a feature-specific API (for example, `Node.lock`), not through a generic write method like `Node.setProperty`. Such changes *are not* governed by the `jcr:modifyProperties` privilege, but rather by the particular feature-specific privilege, for example, `jcr:lockManagement` (see §16.2.3 *Standard Privileges*).

16.3.13 Interaction with Versioning

JCR does not mandate a specific approach to access control of versioning nodes. Whatever approach is taken, any restrictions placed on operations as a consequence of access control are *in addition* to the restrictions imposed by the versioning feature itself (for example, checked-in nodes being immutable).

16.4 Named Access Control Policies

The `NamedAccessControlPolicy` extends the `AccessControlPolicy` marker interface. A `NamedAccessControlPolicy` represents an opaque, immutable policy with a name, which must be a JCR name. The name is accessed through

```
String NamedAccessControlPolicy.getName().
```

16.5 Access Control Lists

`AccessControlList` extends the `AccessControlPolicy` marker interface. An `AccessControlList` represents a list of `AccessControlEntry` objects. Before being bound to a node, the `AccessControlList` is mutable.

16.5.1 Access Control Entries

An `AccessControlEntry` represents the association of one or more `javax.jcr.security.Privilege` objects with a specific `java.security.Principal`. These are accessed through

```
Privilege[] AccessControlEntry.getPrivileges()
```

and

```
java.security.Principal AccessControlEntry.getPrincipal().
```

16.5.2 Getting the Access Control Entries

```
AccessControlEntry[]
      AccessControlList.getAccessControlEntries()
```

returns all access control entries present on the `AccessControlList` policy. It reflects the current state of the policy including modifications that have not yet been persisted.

16.5.3 Adding an Access Control Entry

```
boolean AccessControlList.addAccessControlEntry(
      java.security.Principal prinicipal,
      Privilege[] privileges)
```

adds an access control entry consisting of the specified `principal` and the specified `privileges` to the `AccessControlList` policy and returns `true` if the `AccessControlList` was thereby modified.

How the entries are grouped within the list is implementation-specific. An implementation may, for example, combine the specified `privileges` with those added by a previous call to `addAccessControlEntry` for the same `Principal`. However, a call to `addAccessControlEntry` for a given `Principal` can never remove a `Privilege` added by a previous call.

16.5.4 Removing an Access Control Entry

```
void AccessControlList.
      removeAccessControlEntry(AccessControlEntry ace)
```

removes the specified `AccessControlEntry` from the `AccessControlList` policy. This method is guaranteed to affect only the privileges of the principal defined within the specified `AccessControlEntry`. Only exactly those entries obtained from `AccessControlList.getAccessControlEntries` can be removed through this API.

16.5.5 Modification vs. Effect

An access control entry is added to or removed from an `AccessControlList` upon completion of the `addAccessControlEntry` or `removeAccessControlEntry` call, respectively. However, those modifications only *take effect* once the policy has been bound to a node through `AccessControlManager.setPolicy` and saved.

16.5.6 Privileges to Manage Entries

The user must have the `jcr:modifyAccessControl` privilege to add or remove access control entries and the `jcr:readAccessControl` privilege to read access control entries from an `AccessControlList`.

16.5.7 Principal Discovery

The discovery of `java.security.Principal`s is outside the scope of this specification.

16.6 Privileges Permissions and Capabilities

In JCR, the terms *privilege*, *permission* and *capability* have precise and distinct meanings.

16.6.1 Privileges

The set of *privileges* held by a session with respect to a particular node are the result of access control policies applied using JCR and any other privilege affecting mechanisms external to JCR that may exist, if any.

16.6.2 Permissions

Testing for *permissions* is a feature that all repositories must support regardless of whether they support access control management.

In repositories that do support access control management, the permissions encompass the restrictions imposed by privileges, but also include any additional policy-internal refinements with effects too fine-grained to be exposed through privilege discovery. A common case may be to provide finer-grained access restrictions to individual properties or child nodes of the node to which the policy applies.

In the case of a policy that does not define any refinements, testing privileges is equivalent to using these methods with the following mapping:

The action	**on *I*, a**	**is equivalent to**
`add_node`	node	`jcr:addChildNode` on the parent of I.
`set_property`	property	`jcr:modifyProperties` on the parent of I.
`remove`	node	`jcr:removeChildNodes` on the parent of I and `jcr:removeNode` on I.
`remove`	property	`jcr:modifyProperties` on the parent of I.

`read`	node	`jcr:read` on *I*.
`read`	property	`jcr:read` on the parent of *I*.

16.6.3 Capabilities

Capabilities encompass the restrictions imposed by permissions, but also include any further restrictions unrelated to access control. These include constraints enforced by node types, versioning or any other JCR or implementation-specific mechanism. Capabilities are reported by `Session.hasCapability` (see §9.2 *Capabilities*). The reporting of capabilities is always subject to practical limitations, but should be as accurate as possible, given the design of the implementation.

17 Locking

A repository may support *locking*, which enables a user to temporarily prevent other users from changing a node or subgraph of nodes.

Whether an implementation supports locking can be determined by querying the repository descriptor table with

```
Repository.OPTION_LOCKING_SUPPORTED.
```

A return value of `true` indicates support (see §24.2 *Repository Descriptors*).

17.1 Lockable

A lock is placed on a node by calling `LockManager.lock` (see §17.11.1 *LockManager.lock*). The node on which a lock is placed is called the *holding node* of that lock. Only nodes with mixin node type `mix:lockable` (inherited as part of their primary node type or explicitly assigned) may hold locks. The definition of `mix:lockable` is:

```
[mix:lockable]
  mixin
  - jcr:lockOwner (STRING) protected IGNORE
  - jcr:lockIsDeep (BOOLEAN) protected IGNORE
```

17.2 Shallow and Deep Locks

A lock can be specified as either *shallow* or *deep*. A shallow lock applies only to its holding node and its properties. A deep lock applies to its holding node and all its descendants. Consequently, there is a distinction between a lock *being held by* a node and a lock *applying to* a node. A lock always applies to its holding node. However, if it is a deep lock, it also applies to all nodes in the holding node's subgraph. When a lock applies to a node, that node is said to be *locked*.

Since a deep lock applies to all nodes in the lock-holding node's subgraph, this may include both `mix:lockable` nodes and non-`mix:lockable` nodes. The deep lock applies to both categories of node equally and it *does not* add any `jcr:lockOwner` or `jcr:lockIsDeep` properties to any of the deep-locked `mix:lockable` nodes. However, if any such nodes exist and they already have these properties, this means that they are already locked, and hence the attempt to deep lock above them will fail.

Additionally, assuming a deep lock exists above a `mix:lockable` node, any attempt to lock this lower level `mix:lockable` node will also fail, because it is already locked from above.

17.3 Lock Owner

Initially, the session through which a lock is placed is the *owner* of that lock. This means the session has the power to alter the locked node and to remove the lock. In the case of open-scoped locks (as opposed to session-scoped, see §17.7 *Session-Scoped and Open-Scoped Locks*) control of the lock may be given to another session during the lifetime of that lock. In some implementations giving control of a lock to another session will remove control from the previous session,

in others, more than one session may simultaneously own the same open-scoped lock.

Repositories may support client-specified lock owner information. If this is the case, the `jcr:lockOwner` property will be set to the value supplied upon lock creation, and will not change during the lifetime of the lock. Otherwise, when a lock is created, the `jcr:lockOwner` property is set to the user ID bound to the locking `Session` (that is, the string returned by `Session.getUserID`) or another implementation-dependent string identifying the user.

In implementations that do not support client-specified lock owner information, when an open-scoped lock is moved to a new owner, or assigned an additional one (if supported), the `jcr:lockOwner` property may be automatically altered to reflect the change.

Strictly speaking it is the session, not the user, that owns a particular lock at a particular time. The `jcr:lockOwner` property is used for informational purposes, so that a client application can, for example, display this information to other users. For this reason the user is sometimes informally referred to as the lock owner.

In implementations that record the user ID in `jcr:lockOwner`, that user will not automatically have the ability to alter the locked node if accessing it through another session. Transfer (or, if supported, addition) of ownership must be done explicitly from one session to another and is not governed by the user ID associated with a session.

17.4 Placing and Removing a Lock

When `LockManager.lock` is performed on a `mix:lockable` node, the properties defined in that node type are automatically created and set as follows:

- `jcr:lockOwner` is set to the supplied owner info, the user ID associated with the session that set the lock (this is the value returned by `Session.getUserID`) or another implementation-dependent string identifying the user.

`jcr:lockIsDeep` is set to reflect whether the lock is deep or not.

When `LockManager.unlock` is performed on a locked `mix:lockable` node, through a session that owns the lock these two properties are removed.

Additionally, the content repository may give permission to some sessions to remove locks for which they are not the owner. Typically such "lock-superuser" capability is intended to facilitate administrational clean-up of orphaned open-scoped locks.

An attempt to call `LockManager.lock` or `LockManager.unlock` for a node that is not `mix:lockable` will throw a `LockException`, as will an attempt to lock an already locked node or unlock an already unlocked node.

17.5 Lock Token

The method `LockManager.lock` returns a `Lock` object. If the lock is open-scoped the lock will contain a lock token. A lock token is a string that uniquely identifies a particular lock and acts as a key granting lock ownership to any session that hold the token.

In order to use the lock token as a key, it must be added to the session, thus permitting that session to alter the nodes to which the lock applies or to remove the lock. When a lock token is attached to a `Session`, the session becomes an owner of the lock.

The method `LockManager.lock` automatically adds the lock token for a newly placed open-scoped lock to the current session.

The client can also control which lock tokens are attached to the session through the `LockManager` methods `addLockToken`, `removeLockToken` and `getLockTokens`.

17.6 Session-Scoped and Open-Scoped Locks

When a lock is placed on a node, it can be specified to be either a session-scoped lock or an open-scoped lock. A session-scoped lock automatically expires when the session through which the lock owner placed the lock expires. An open-scoped lock does not expire until it is explicitly unlocked, it times out or an implementation-specific limitation intervenes.

In the case of open-scoped locks, the lock token must be attached to the current session in order to alter any nodes locked by that token's lock.

In the case of session-scoped locks, the user need not explicitly do anything since the lock is automatically associated with the session and expires with it in any case.

With open–scoped locks the token is automatically attached to the session. However, the user must additionally ensure that a reference to the lock token is preserved separately so that it can later be attached to another session since, presumably, an open-scoped lock is being used to avoid co-expiration with the initial session. It is for handling these cases of attaching an existing lock token from a previous session to a new session that the methods `LockManager.addLockToken`, `LockManager.removeLockToken` and `LockManager.getLockTokens` are provided (see §17.11 *LockManager Object*).

To determine an existing lock's scoping, the method `Lock.isSessionScoped` is provided.

If a `Lock` is session-scoped, the method `Lock.isLockOwningSession` can be used to determine whether the current session is the lock owner.

An implementation *may* support simultaneous ownership of open-scoped locks across sessions.

17.7 Effect of a Lock

If a lock applies to a node (i.e., the node either holds the lock or is a descendant of a node holding a deep lock), then to all sessions except the lock-owning

session, the same restrictions apply with respect to the node as would apply if the node were protected (see §3.7.2.2 *Protected*).

Removing a node is considered an alteration of *its parent*. This means that a node within the scope of a lock may be removed by a session that is not an owner of that lock, assuming no other restriction prevents the removal. Similarly, a locked node and its subgraph may be moved by a non-lock-owning session if no restriction prevents the alteration of the source and destination parent nodes.

Locked nodes can always be read and copied by any session with sufficient access privileges.

When an action is prevented due to a lock, a `LockException` is thrown either immediately or on the subsequent `save`. Implementations may differ on which of these behaviors is used to enforce locking.

There is at most one lock on any node at one time.

17.8 Timing Out

Implementations may support client-supplied timeout information, but are not required to do so. Additionally, an implementation may remove (unlock) any lock at any time due to implementation-specific criteria.

17.9 Locks and Persistence

When a new node is added below a deep lock by that lock's owning session `LockManager.isLocked(Node)` will report true *even before the node is persisted*[21]. However, since the node is not visible to other `Session`s, its locked status has no effect until it is persisted.

17.10 Locks and Transactions

Locking and unlocking are treated just like any other operation in the context of a transaction. For example, consider the following series of operations:

begin
  `lock`
  do A
  `save`
  do B
  `save`
  `unlock`
commit

[21] Recall that *outside a transaction* persistence of transient state occurs immediately upon a `Session.save` while, within a transaction, the effect of any `Session.save` calls is deferred until commit of the transaction.

In this example the `lock` and `unlock` have no effect. This series of operations is equivalent to:

begin
  do A
  `save`
  do B
  `save`
commit

The reason for this is that changes to a workspace are only made visible to other `Session`s upon commit of the transaction, and this includes changes in the locked status of a node. As a result, if a lock is enabled and then disabled within the same transaction, its effect never makes it to the persistent workspace and therefore it does nothing.

In order to use locks properly (that is, to prevent the "lost update problem"), locking and unlocking must be done in separate transactions. For example:

begin
  `lock`
commit

begin
  do A
  `save`
  do B
  `save`
  `unlock`
commit

This series of operations would ensure that the actions *A* and *B* are protected by the lock.

17.11 LockManager Object

The methods for locking, unlocking and querying the locking status of a node are found in the `LockManager`, acquired through

```
LockManager Workspace.getLockManager().
```

17.11.1 Locking a Node

```
Lock LockManager.lock(String absPath,
                      boolean isDeep,
                      boolean isSessionScoped,
                      long timeout,
                      String ownerInfo)
```

places a lock on the node at `absPath`. If successful, the node is said to *hold* the lock.

If `isDeep` is `true` then the lock *applies* to the specified node and all its descendant nodes; if `false`, the lock applies only to the specified node. On a

successful lock, the `jcr:lockIsDeep` property of the locked node is set to this value.

If `isSessionScoped` is `true` then this lock will expire upon the expiration of the current session (either through an automatic or explicit `Session.logout`); if `false`, this lock does not expire until it is explicitly unlocked, it times out, or it is automatically unlocked due to an implementation-specific limitation.

The `timeout` parameter specifies the number of seconds until the lock times out (if it is not refreshed in the meantime, see §10.11.1 *Refresh*). An implementation may use this information as a hint or ignore it altogether. Clients can discover the actual timeout by inspecting the returned `Lock` object.

The `ownerInfo` parameter can be used to pass a string holding owner information relevant to the client. An implementation may either use or ignore this parameter. If it uses the parameter it must set the `jcr:lockOwner` property of the locked node to this value and return this value on `Lock.getLockOwner`. If it ignores this parameter the `jcr:lockOwner` property (and the value returned by `Lock.getLockOwner`) is set to either the value returned by `Session.getUserID` of the owning session or an implementation-specific string identifying the owner.

The method returns a `Lock` object representing the new lock.

If the lock is open-scoped the returned lock will include a lock token. The lock token is also automatically added to the set of lock tokens held by the current `Session`.

The addition or change of the properties `jcr:lockIsDeep` and `jcr:lockOwner` are persisted immediately; there is no need to call `save`.

It is possible to lock a node even if it is checked-in (see §15.2.2 *Read-Only on Check-In*).

17.11.2 Getting a Lock

```
Lock LockManager.getLock(String absPath)
```

returns the `Lock` object that applies to the node at absPath. This may be either a lock on the node itself or a deep lock on a node above that node.

If the current session holds the lock token for this lock and the lock is open-scoped, then the returned `Lock` object contains that lock token (accessible through `Lock.getLockToken`). If this `Session` does not hold the applicable lock token and the lock is open-scoped, the returned `Lock` object *may* return the lock token. Otherwise, the returned `Lock` object will not contain the lock token and its `Lock.getLockToken` method will return `null` (see §17.12.4 *Getting a Lock Token*).

17.11.3 Unlocking a Node

```
void LockManager.unlock(String absPath)
```

Removes the lock, and the properties `jcr:lockOwner` and `jcr:lockIsDeep`, from the node at `absPath`. These changes are persisted automatically; there is no need

to call `save`. As well, the corresponding lock token is removed from the set of lock tokens held by the current session.

If this node does not currently hold a lock or holds a lock for which this `Session` is not the owner, then a `LockException` is thrown.

The system may give permission to a non-owning session to unlock a lock. Typically such "lock-superuser" capability is intended to facilitate administrational clean-up of orphaned open-scoped locks.

It is possible to unlock a node even if it is checked-in (see §15.2.2 *Read-Only on Check-In*).

17.11.4 Testing for Lock Holding

```
boolean LockManager.holdsLock(String absPath)
```

returns `true` if the node at `absPath` holds a lock; otherwise returns `false`. To *hold* a lock means that the node has actually had a lock placed on it specifically, as opposed to having a lock *apply* to it due to a deep lock held by a node above.

17.11.5 Testing for Locked Status

```
boolean LockManager.isLocked(String absPath)
```

returns `true` if the node at absPath is locked either as a result of a lock held by the specified node or by a deep lock on a node above that node; otherwise returns `false`.

Alternatively, the method

```
boolean Node.isLocked()
```

can be used directly on the node in question.

17.11.6 Adding a Lock Token

```
void LockManager.addLockToken(String lockToken)
```

adds the specified lock token to the current session. Holding a lock token makes this session the owner of the lock specified by that particular lock token. If the implementation does not support simultaneous lock ownership this method will transfer ownership of the lock corresponding to the specified `lockToken` to the current session, otherwise the current session will become an additional owner of that lock. In either case, if the implementation does not support client-specified lock owner information, this method may cause a change in the `jcr:lockOwner` property (and the value returned by `Lock.getLockOwner`) of the lock corresponding to the specified `lockToken` (see §17.5 *Lock Token*).

17.11.7 Getting Lock Tokens

```
String[] LockManager.getLockTokens()
```

returns an array containing all lock tokens currently held by the current session. Note that any such tokens will represent open-scoped locks, since session–scoped locks do not have tokens.

17.11.8 Removing a Lock Token

```
void LockManager.removeLockToken(String lockToken)
```

Removes the specified `lockToken` from the current session, causing the session to no longer be an owner of the lock associated with the `lockToken`. If the implementation does not support client-specified lock owner information, this method may cause a change in the `jcr:lockOwner` property (and the value returned by `Lock.getLockOwner`) of the lock corresponding to the specified `lockToken` (see §17.5 *Lock Token*).

17.12 Lock Object

The `Lock` object represents a lock on a particular node. It is acquired either on lock creation through `LockManager.lock` or after lock creation through `LockManager.getLock`.

17.12.1 Getting the Lock Owner

```
String Lock.getLockOwner()
```

returns the value of the `jcr:lockOwner` property. This is either the client-supplied lock owner information, the user ID bound to the session that holds the lock or an implementation-specific string identifying the user (see §4.4.1 *User*).

The lock owner's identity is only provided for informational purposes. It does not govern who can perform an unlock or make changes to the locked nodes; that depends entirely upon the session that holds the lock token.

17.12.2 Testing Lock Depth

```
boolean Lock.isDeep()
```

returns `true` if this is a deep lock; `false` otherwise.

17.12.3 Getting the Lock Holding Node

```
Node Lock.getNode()
```

returns the lock holding node. Note that `N.getLock().getNode()` (where `N` is a locked node) will only return `N` if `N` is the lock holder. If `N` is in the subgraph of the lock holder, `H`, then this call will return `H`.

17.12.4 Getting a Lock Token

```
String Lock.getLockToken()
```

may return the lock token for this lock. If this lock is open-scoped and the current session holds the lock token for this lock, then this method will return that lock token. If the lock is open-scoped and the current session does not hold the lock token, it *may* return the lock token. Otherwise this method will return `null`.

17.12.5 Testing Lock Aliveness

```
boolean Lock.isLive()
```

returns `true` if this `Lock` object represents a lock that is currently in effect. If this lock has been unlocked either explicitly or due to an implementation-specific limitation (like a timeout) then it returns `false`. Note that this method is intended for those cases where one is holding a `Lock` Java object and wants to find out whether the lock (the repository-level entity that is attached to the lockable node) that this object originally represented still exists. For example, a timeout or explicit `unlock` will remove a lock from a node but the `Lock` Java object corresponding to that lock may still exist, and in that case its `isLive` method will return `false`.

17.12.6 Testing Lock Scope

```
boolean Lock.isSessionScoped()
```

Returns `true` if this is a session-scoped lock and the scope is bound to the current session. Returns `false` otherwise.

17.12.7 Testing Lock Owning Session

```
boolean Lock.isLockOwningSession()
```

Returns `true` if the current session is the owner of this lock, either because it is session-scoped and bound to this session or open-scoped and this session currently holds the token for this lock. Returns `false` otherwise.

17.12.8 Getting Seconds Remaining

```
long Lock.getSecondsRemaining()
```

If this lock's time-to-live is governed by a timer, the number of remaining seconds until time out is returned. If this lock's time-to-live is not governed by a timer, then this method returns `Long.MAX_VALUE`.

17.12.9 Refreshing a Lock

```
void Lock.refresh()
```

If this lock's time-to-live is governed by a timer, this method resets that timer. If this lock's time-to-live is not governed by a timer, then this method has no effect.

17.13 LockException

When a method fails due to the presence or absence of a lock on a particular node a `LockException` is thrown.

`LockException` extends `RepositoryException`, adding the method

```
String LockException.getFailureNodePath(),
```

which returns the absolute path of the node that caused the error, or `null` if the implementation chooses not to, or cannot, return a path.

18 Lifecycle Management

A repository may support *lifecycle management*, enabling users to:

- Discover the state of a node within a lifecycle.
- Promote or demote nodes through a lifecycle by following a transition from the current state to a new state.

The names and semantics of the supported lifecycle states and transitions are implementation-specific.

Whether an implementation supports lifecycle management can be determined by querying the repository descriptor table with

```
Repository.OPTION_LIFECYCLE_SUPPORTED.
```

A return value of `true` indicates support (see §24.2 *Repository Descriptors*).

18.1 mix:lifecycle

```
[mix:lifecycle]
  mixin
  - jcr:lifecyclePolicy (REFERENCE) protected INITIALIZE
  - jcr:currentLifecycleState (STRING) protected INITIALIZE
```

Only nodes with mixin node type `mix:lifecycle` may participate in a lifecycle. The mixin adds two properties:

- `jcr:lifecyclePolicy`: This property is a reference to another node that contains lifecycle policy information. The definition of the referenced node is not specified.
- `jcr:currentLifecycleState`: This property is a string identifying the current lifecycle state of this node. The format of this string is not specified.

18.2 Node Methods

The `Node` interface provides the following methods related to lifecycles. If the node does not have the `mix:lifecycle` mixin, the methods will return `UnsupportedRepositoryOperationException`.

```
void Node.followLifecycleTransition(String transition)
```

causes the lifecycle state of this node to undergo the specified transition.

This method may change the value of the `jcr:currentLifecycleState` property, in most cases it is expected that the implementation will change the value to that of the passed `transition` parameter, though this is an implementation-specific issue. If the `jcr:currentLifecycleState` property is changed the change is persisted immediately, there is no need to call `save`.

```
String[] Node.getAllowedLifecycleTransitions()
```

returns the list of valid state transitions for this node.

19 Node Type Management

A repository may support *node type management*. Depending on implementation-specific limitations (see §19.3 *Node Type Registration Restrictions*), this feature may include some or all of the following:

- Adding a node type to the registry.
- Removing a node type from the registry.
- Updating the definition of a registered node type that *is not* currently in use as the node type of any node in the repository.
- Updating the definition of a registered node type that *is* currently in use as the node type of a node in the repository.
- Import of node type definitions to the repository.
- Export of node types from the repository.

Whether a particular implementation supports node type management and the restrictions in place with regard to this feature can be determined by querying the repository descriptor table with the constants listed in §24.2.4 *Node Type Management*.

19.1 NodeTypeDefinition

The `NodeTypeDefinition` interface provides methods for discovering the static definition of a node type. These are accessible both before and after the node type is registered. Its subclass `NodeType` adds methods that are relevant only when the node type is "live"; that is, after it has been registered.

In implementations that support node type registrations, `NodeTypeDefinition` serves as the superclass of both `NodeType` and `NodeTypeTemplate`. In implementations that do not support node type registration, only objects implementing the subclass `NodeType` will be encountered.

19.2 NodeTypeManager

The `NodeTypeManager` interface provides the following methods related to registering node types. For methods of this interface that are related to node type discovery, see §8 *Node Type Discovery*. In implementations that do not support node type management, the methods of `NodeTypeManager` will throw an `UnsupportedRepositoryOperationException`.

19.2.1 Creating a NodeTypeTemplate

```
NodeTypeTemplate NodeTypeManager.createNodeTypeTemplate()
```

returns an empty `NodeTypeTemplate` which can then be used to define a node type and passed to `registerNodeType`.

```
NodeTypeTemplate NodeTypeManager.
  createNodeTypeTemplate(NodeTypeDefinition ntd)
```

returns a `NodeTypeTemplate` holding the specified `NodeTypeDefinition`. This template may then be altered and passed to `registerNodeType`.

19.2.2 Creating a NodeDefinitionTemplate

```
NodeDefinitionTemplate NodeTypeManager.
  createNodeDefinitionTemplate()
```

returns an empty `NodeDefinitionTemplate` which can then be used to create a child node definition and attached to a `NodeTypeTemplate`.

19.2.3 Creating a PropertyDefinitionTemplate

```
PropertyDefinitionTemplate NodeTypeManager.
  createPropertyDefinitionTemplate()
```

returns an empty `PropertyDefinitionTemplate` which can then be used to create a property definition and attached to a `NodeTypeTemplate`.

19.2.4 Registering a Node Type

```
NodeType NodeTypeManager.
  registerNodeType(NodeTypeDefinition ntd, boolean allowUpdate)
```

registers a new node type or updates an existing node type using the specified definition and returns the resulting `NodeType` object. Typically, the object passed to this method will be a `NodeTypeTemplate` (a subclass of `NodeTypeDefinition`) acquired from `NodeTypeManager.createNodeTypeTemplate` and then filled-in with definition information. If `allowUpdate` is `true` then an attempt to change the definition of an already registered node type will be made (see §19.2.4.1 *Updating Node Types*), otherwise an attempt to register a node type with the same name as an already registered one will fail immediately.

```
NodeTypeIterator NodeTypeManager.
  registerNodeTypes(NodeTypeDefinition[] ntds,
                    boolean allowUpdate)
```

registers or updates the specified array of `NodeTypeDefinition` objects. This method is used to register or update a set of node types with mutual dependencies. It returns an iterator over the resulting `NodeType` objects. The effect of the method is "all or nothing"; if an error occurs, no changes are made.

19.2.4.1 Updating Node Types

A repository that supports node type management may support updates to a node type already in use as the type of an existing node. The extent of any such capability is implementation dependent. For example, some implementations may permit only changes which do not invalidate existing content, while others may allow larger changes. How any resulting incompatibilities are resolved is also implementation dependent. Any changes to the type of an exiting node must take effect in accordance with the *node type assignment behavior* of the repository (see §10.10.1 *Node Type Assignment Behavior*).

19.2.5 Unregistering a Node Type

```
void NodeTypeManager.unregisterNodeType(String nodeTypeName)
```

unregisters the specified node type.

```
void NodeTypeManager.
  unregisterNodeTypes(String[] nodeTypeNames)
```

unregisters the specified set of node types. This method is used to unregister a set of node types with mutual dependencies.

19.2.6 Testing for Node Types

```
boolean NodeTypeManager.hasNodeType(String nodeTypeName)
```

returns `true` if a node type with the specified name is registered and returns `false` otherwise.

19.3 Node Type Registration Restrictions

A repository *must* prevent the registration of any node type that uses a reserved namespace either in its name or in the name of any of its item definitions (see 3.4 *Namespace Mapping*).

A repository *may* restrict the range of node types that can be registered according to implementation-specific criteria. This is most relevant in cases where a JCR repository is built on top of an existing content store which has intrinsic limitations that restrict the space of supported node types.

19.4 Templates

Node types are defined programmatically by setting the attributes of template objects and passing these to the `NodeTypeManager`.

The `NodeTypeTemplate` is a container holding the node type's attributes and its property and child node definitions, which are themselves represented by `NodeDefinitionTemplate` and `PropertyDefinitionTemplate` objects, respectively.

The user registers a node type by first acquiring a `NodeTypeTemplate` and the necessary `PropertyDefinitionTemplate` or `NodeDefinitionTemplate` objects through the `NodeTypeManager` (see §19.2 *NodeTypeManager*). The attributes of these objects are then set, with the appropriate `PropertyDefinitionTemplate` and `NodeDefinitionTemplate` objects added to the `NodeTypeTemplate` object. The resulting `NodeTypeTemplate` object is then passed to a registration method of the `NodeTypeManager`.

19.4.1 NodeTypeTemplate

`NodeTypeTemplate`, like `NodeType`, is a subclass of `NodeTypeDefinition`, so it shares with `NodeType` those methods that are relevant to a static definition. In addition to the methods inherited from `NodeTypeDefinition`, `NodeTypeTemplate` provides methods for setting the attributes of the definition. The setter methods are named appropriately according to the attribute that they set (see 3.6.1 *Node*

Type Definition Attributes). Consult the Javadoc for details on the method signatures.

19.4.1.1 Setting Property and Child Node Definitions

Setting the property definitions within a node type template is done by adding `PropertyDefinitionTemplate` objects to the mutable `List` object retrieved from

```
List NodeTypeTemplate.getPropertyDefinitionTemplates().
```

Similarly, setting the child node definitions is done by adding `NodeDefinitionTemplate` objects to the mutable `List` object retrieved from

```
List NodeTypeTemplate.getNodeDefinitionTemplates().
```

19.4.1.2 Default Values of Node Type Attributes

See the corresponding get methods for each attribute in `NodeTypeDefinition` (see §19.1 *NodeTypeDefinition*) for the default values assumed when a new empty `NodeTypeTemplate` is created.

19.4.2 PropertyDefinitionTemplate

The `PropertyDefinitionTemplate` interface extends `PropertyDefinition` (see §8.4 *PropertyDefinition Object*) with the addition of write methods, enabling the characteristics of a child property definition to be set, after which the `PropertyDefinitionTemplate` is added to a `NodeTypeTemplate`. The setter methods are named appropriately according to the attribute that they set (see §3.7.2 *Item Definition Attributes* and §3.7.3 *Property Definition Attributes*). Consult the Javadoc for details on the method signatures.

19.4.2.1 Default Values of Property Definition Attributes

See the corresponding get methods for each attribute in `PropertyDefinition` (see §8.4 *PropertyDefinition Object*) for the default values assumed when a new empty `PropertyDefinitionTemplate` is created.

19.4.3 NodeDefinitionTemplate

The `NodeDefinitionTemplate` interface extends `NodeDefinition` (see §8.5 *NodeDefinition Object*) with the addition of write methods, enabling the characteristics of a child node definition to be set, after which the `NodeDefinitionTemplate` is added to a `NodeTypeTemplate`. The setter methods are named appropriately according to the attribute that they set (see §3.7.2 *Item Definition Attributes* and §3.7.4 *Child Node Definition Attributes*). Consult the Javadoc for details on the method signatures.

19.4.3.1 Default Values of Child Node Definition Attributes

See the corresponding get methods for each attribute in `NodeDefinition` (see §8.5 *NodeDefinition Object*) for the default values assumed when a new empty `NodeDefinitionTemplate` is created.

20 Retention and Hold

A repository may support *retention and hold*, which enables an external retention management application to apply retention policies to repository content and supports the concepts of hold and release[22].

Whether a particular implementation supports these features can be determined by querying the repository descriptor table with

```
Repository.OPTION_RETENTION_SUPPORTED.
```

a return value of `true` indicates support (see §24.2 *Repository Descriptors*).

This API is intended for use by a retention and hold management system (often external to the repository). It should not be used as a substitute for normal access control.

20.1 Retention Manager

Retention and hold is exposed through a

```
javax.jcr.retention.RetentionManager
```

acquired from the `Session` using

```
RetentionManager Session.getRetentionManager().
```

All changes made through the retention and hold API are session-mediated and therefore require a `Session.save()` to go into effect.

20.2 Placing a Hold

The method

```
Hold RetentionManager.
   addHold(String absPath, String name, boolean isDeep)
```

places a hold on the node at `absPath`. If `isDeep` is `false`, a *shallow hold* is placed. If `isDeep` is `true`, a *deep hold* is placed. The method returns the resulting `Hold` object. The hold only takes effect upon `Session.save()`. A node may have more than one hold.

The format and interpretation of the `name` is application-dependent. It need not be unique.

[22] In some systems this feature is called "freeze" or "legal hold" (when the hold is applied due to legal requirements).

20.3 Effect of a Hold

A shallow hold in effect on a node `N` has the same effect as would be the case if `N` were protected.

A deep hold in effect on a node `N` has the same effect as would be the case if `N` and all nodes in its subgraph were protected (see §3.7.2.2 *Protected*).

20.4 Getting the Holds present on a Node

The method

```
Hold[] RetentionManager.getHolds(String absPath)
```

returns all holds on the node at `absPath`.

20.5 Removing a Hold

The method

```
void RetentionManager.
  removeHold(String absPath, Hold hold)
```

removes the specified hold from the node at `absPath`. The removal only takes effect upon `Session.save()`.

20.6 Hold Object

The `Hold` interface defines two methods:

```
String Hold.getName()
```

which returns the name of the hold, and

```
boolean Hold.isDeep()
```

which reports whether the hold is deep or shallow.

20.7 Setting a Retention Policy

```
void RetentionManager.
  setRetentionPolicy(String absPath, RetentionPolicy policy)
```

sets the retention policy of the node at `absPath` to that defined in the specified retention policy object. The policy only takes effect upon `Session.save()`.

20.8 Getting a Retention Policy

```
RetentionPolicy RetentionManager.
  getRetentionPolicy(String absPath)
```

returns the retention policy on the node at `absPath` or `null` if no retention policy has been set on the node.

20.9 Effect of a Retention Policy

Interpretation and enforcement of a retention policy is an implementation issue. However, in all cases a retention policy in effect on a node `N`:

- prevents the removal of `N` and
- prevents the addition and removal of all child nodes of `N` and
- prevents the addition, removal and change of all properties of `N`.

20.10 RetentionPolicy object

The `RetentionPolicy` interface defines one method:

```
String RetentionPolicy.getName()
```

which returns the name of the policy.

20.11 Removing a Retention Policy

```
void RetentionManager.removeRetentionPolicy(String absPath)
```

removes the current retention policy on this node, if any. The removal only takes effect upon a call to `Session.save()`.

21 Transactions

A repository may support *transactions*.

Whether a particular implementation supports transactions can be determined by querying the repository descriptor table with

```
Repository.OPTION_TRANSACTIONS_SUPPORTED.
```

A return value of `true` indicates support for transactions (see *Repository Descriptors*).

A repository that supports transactions must adhere to the Java Transaction API (JTA) specification[23].

The actual methods used to control transaction boundaries are not defined by this specification. For example, there are no *begin*, *commit* or *rollback* methods in JCR API. These methods are defined by the JTA specification.

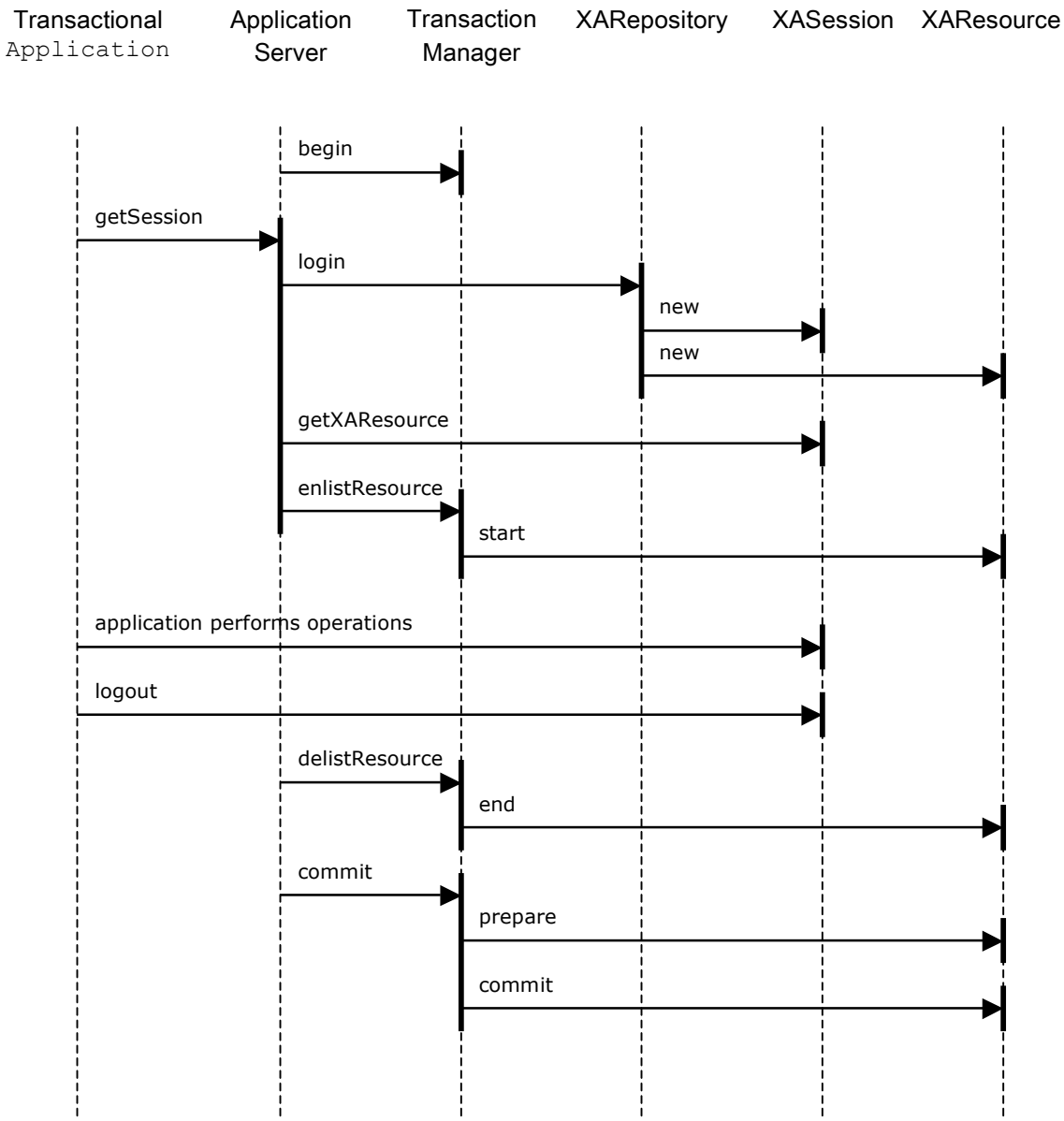
The JTA provides for two general approaches to transactions, container managed transactions and user managed transactions. In the first case, container managed transactions, the transaction management is taken care of by the application server and it is entirely transparent to the application using the API. The JTA interfaces `javax.transaction.TransactionManager` and `javax.transaction.Transaction` are the relevant ones in this context (though the client, as mentioned, will never have a need to use these).

In the second case, user managed transactions, the application using the API may choose to control transaction boundaries from within the application. In this case the relevant interface is `javax.transaction.UserTransaction`. This is the interface that provides the methods `begin`, `commit`, `rollback` and so forth. Note that behind the scenes the `javax.transaction.TransactionManager` and `javax.transaction.Transaction` are still employed, but again, the client does not deal with these.

A content repository implementation must support both of these approaches if it supports transactions.

[23] See `http://java.sun.com/products/jta/index.html`.

21.1 Container Managed Transactions: Sample Request Flow



21.2 User Managed Transactions: Sample Code

```
// Get user transaction (for example, through JNDI)
UserTransaction utx = ...

// Get a node
Node n = ...

// Start a user transaction
utx.begin();

// Do some work
n.setProperty("myapp:title", "A Tale of Two Cities")
n.save();
```

```
// Do some more work
n.setProperty("myapp:author", "Charles Dickens")
n.save();

// Commit the user transaction
utx.commit();
```

21.3 Save vs. Commit

Throughout this specification we often mention the distinction between *transient* and *persistent* levels. The persistent level refers to the (one or more) workspaces that make up the actual content storage of the repository. The transient level refers to in-memory storage associated with a particular `Session` object.

In these discussions we usually assume that operations occur outside the context of transactions; it is assumed that `save` and other workspace-altering methods immediately effect changes to the persistent layer, causing those changes to be made visible to other sessions.

This is not the case, however, once transactions are introduced. Within a transaction, changes made by `save` (or other, workspace-direct, methods) are transactionalized and are only persisted and published (made visible to other sessions), upon commit of the transaction. A rollback will, conversely, revert the effects of any saves or workspace-direct methods called within the transaction.

Note, however, that changes made in the transient storage are *not* recorded by a transaction. This means that a rollback will not revert changes made to the transient storage of the `Session`. After a rollback the `Session` object state will still contain any pending changes that were present before the rollback.

21.4 Single Session Across Multiple Transactions

Because modifications in the transient layer are not transactionalized, the possibility exists for some implementations to allow a `Session` to be shared across transactions. This possibility arises because in JTA, an `XAResource` may be successively associated with different global transactions and in many implementations the natural mapping will be to make the `Session` implement the `XAResource`. The following code snippet illustrates how an `XAResource` may be shared across two global transactions:

```
// Associate the resource (our Session) with a global
// transaction xid1
res.start(xid1, TMNOFLAGS);

// Do something with res, on behalf of xid1
// ...

// Suspend work on this transaction
res.end(xid1, TMSUSPEND);

// Associate (the same!) resource with another
// global transaction xid2
res.start(xid2, TMNOFLAGS);

// Do something with res, on behalf of xid2
// ...
```

```
// End work
res.end(xid2, TMSUCCESS);

// Resume work with former transaction
res.start(xid1, TMRESUME);

// Commit work recorded when associated with xid2
res.commit(xid2, true);
```

In cases where the `XAResource` corresponds to a `Session` (that is, probably most implementations), items that have been obtained in the context of `xid1` would still be valid when the `Session` is effectively associated with `xid2`. In other words, all transactions working on the same `Session` would share the transient items obtained through that `Session`.

In those implementations that adopt a copy-on-read approach to transient storage (see §10.11.9 *Seeing Changes Made by Other Sessions*) this will mean that the a session is disassociated from a global transaction. This is however, outside the scope of this specification.

22 Same-Name Siblings

A repository may support *same-name siblings (SNS)*, which enables a node to have more than one child node with the same name.

Whether a particular implementation supports same-name siblings can be determined by querying the repository descriptor table with

```
Repository.NODE_TYPE_MANAGEMENT_SAME_NAME_SIBLINGS_SUPPORTED.
```

A return value of `true` indicates support for transactions (see *Repository Descriptors*).

22.1 Scope of Same-Name Siblings

Same-name sibling capability is defined *per child node* in the node type definition of the parent node using the same-name sibling attribute of the child node definition. Therefore, whether a particular child node can have sibling node with the same name depends on that child node's *scoping child node definition* (see §3.7.2.1 *Item Definition Name*).

A repository supports same-name siblings by permitting the registration of node types (or by providing built-in node types) with child node definitions that have a same-name sibling attribute of `true`. Disallowing same-name siblings consists in preventing the availability of such node types.

22.2 Addressing Same-Name Siblings by Path

A particular node within a same-name sibling group can be addressed by embedding an array-like notation within the path. For example the path `/a/b[2]/c[3]` specifies the third child node called `c` of the second child node called `b` of the node `a` below the root.

The indexing of same-name siblings begins at `1`, not `0`. This is done for backwards compatibility with JCR 1.0 and in particular the support in that specification for XPath, which uses a base-1 index.

A name in a content repository path that does not explicitly specify an index implies an index of `1`. For example, `/a/b/c` is equivalent to `/a[1]/b[1]/c[1]`.

The indexing is based on the order in which child nodes are returned in the iterator acquired through `Node.getNodes()`.

Same-name siblings are indexed by their position relative to each other in this larger ordered set. For example, the order of child nodes returned by a `getNodes` on some parent might be:

```
[A, B, C, A, D]
```

In this case, `A[1]` refers the first node in the list and `A[2]` refers to the fourth node in the list.

If a node with same-name siblings is removed, this decrements by one the indices of all the siblings with indices greater than that of the removed node. In

other words, a removal compacts the array of same-name siblings and causes the minimal re-numbering required to maintain the original order but leave no gaps in the numbering.

The relative ordering of a set of same-name sibling nodes is not guaranteed to be persistent unless the nodes are specified to also be orderable (see §23 *Orderable Child Nodes*). Non-orderable same-name siblings can only be relied upon to act as an anonymous, unordered collection of nodes, though an implementation is free to make the ordering more stable.

22.3 Reading and Writing Same-Name Siblings

22.3.1 Getting a Same-Name Sibling Set

```
NodeIterator Node.getNodes(String namePattern)
```

can be used to retrieve a same-name sibling set. This method returns an iterator over all the child nodes of the calling node that have the specified pattern. Making `namePattern` just a name, without wildcards, retrieves all the child nodes with that name, see §5.2.2 *Iterating Over Child Items*.

22.3.2 Getting a Particular Same-Name Sibling Node

In the method

```
Node Node.getNode(String relPath),
```

if `relPath` contains a path element that refers to a node with same-name sibling nodes without explicitly including an index using the array-style notation (`[x]`), then the index `[1]` is assumed.

22.3.3 Getting a Node's Index

```
int Node.getIndex()
```

returns the index of this node within the ordered set of its same-name sibling nodes. This index is the one used to address same-name siblings using the square-bracket notation, e.g., `/a[3]/b[4]`. For nodes that do not have same-name-siblings, this method will always return `1`.

22.3.4 When a Same-Name Sibling is a Primary Item

In cases where the primary child item of a node specifies the name of a set of same-name sibling child nodes, the node returned by

```
Item Node.getPrimaryItem()
```

will be the one among the same-name siblings with index `[1]`.

22.3.5 Removing a Same-Name Sibling Node

If a node with same-name siblings is removed using

```
void Node.remove()
```

this decrements by one the indices of all the siblings with indices greater than that of the removed node. In other words, a removal compacts the array of same-name siblings and causes the minimal re-numbering required to maintain the original order but leave no gaps in the numbering.

22.4 Properties Cannot Have Same-Name Siblings

Properties cannot have sibling properties of the same name. However, they may have multiple values (see §3.6.3 *Single and Multi-Value Properties*).

22.5 Effect of Access Denial on Read of Same-Name Siblings

In most cases, the nodes and properties to which a user does not have read access will simply appear not to exist on a read attempt (see §5.5 *Effect of Access Denial on Read*).

However, a repository that supports same-name siblings *may* violate this general rule in the case where a user is denied access to a subset of same-name sibling nodes. In such a case, a repository may choose not to compact the indices of the same-name-sibling set (thus "hiding" the any inaccessible nodes), but instead allow "holes" to appear in the index count.

For example, consider the nodes `M/N`, `M/N[2]` and `M/N[3]` with identifiers `x`, `y` and `z`, respectively:

```
M/N     (x)
M/N[2]  (y)
M/N[3]  (z)
```

On `M.getNodes()`, a user with no read access to the node with identifier `y` will observe one of two behaviors, depending on the implementation. A repository that compacts indices on read denial will return

```
M/N     (x)
M/N[2]  (z)
```

while a repository that does not compact indices will return

```
M/N     (x)
M/N[3]  (z)
```

Which behavior is followed is implementation-determined. Note however, that in the case where a subset of same-name siblings is actually removed (as opposed to hidden from certain users), index compaction is required (see §22.2.5 *Removing a Same-Name Sibling Node*).

23 Orderable Child Nodes

A repository may support *orderable child nodes*, which enables persistent, client-controlled ordering of a node's child nodes.

Whether a particular implementation supports orderable child nodes can be determined by querying the repository descriptor table with

```
Repository.NODE_TYPE_MANAGEMENT_ORDERABLE_CHILD_NODES_SUPPORTED.
```

A return value of `true` indicates support for transactions (see *Repository Descriptors*).

23.1 Scope of Orderable Child Nodes

The orderable child nodes setting is defined *per node type*. Whether the child nodes of a node *N* are orderable depends on the node type of *N*.

A repository supports orderable child nodes by permitting the registration of node types with an orderable child node setting of `true`. Disallowing orderable child nodes consists in preventing the availability of such node types.

For a given `NodeType T`:

- If `T.hasOrderableChildNodes()` returns `true` then *all* nodes with primary type *`T`* *must* have orderable child nodes.
- If `T.hasOrderableChildNodes()` returns `false` then *some* nodes with primary type `T` *may* have orderable child nodes.

Only the primary node type of a node is relevant to the orderable status of its child nodes. This setting on a mixin node type of a node has no meaning.

If a node has orderable child nodes then at any time its child node set has a *current order*, reflected in the iterator returned by `Node.getNodes()` (see §5.2.2 *Iterating Over Child Items*). If a node does not have orderable child nodes then the order of nodes returned by `Node.getNodes` is not guaranteed and may change at any time.

23.2 Ordering Child Nodes

If a node has orderable child nodes then their *current order* can be changed using

```
void Node.orderBefore(String srcChildRelPath,
                      String destChildRelPath).
```

This method moves the child node at `srcChildRelPath` and inserts it immediately before its sibling at `destChildRelPath` in the child node list. To place the node `srcChildRelPath` at the end of the list, a `destChildRelPath` of `null` is used.

Apart from the case where `destChildRelPath` is `null`, both of these arguments must be relative paths of depth `1`, in other words, they must be the names of child nodes, possibly suffixed with an index. (see §3.2 *Names* and §3.4 *Paths*).

If `srcChildRelPath` and `destChildRelPath` are the identical, then no change is made.

Changes to the current order are visible immediately through the current `Session` and are persisted to the workspace on `Session.save`.

23.3 Adding a New Child Node

When a child node is added to a node that has orderable child nodes it is added to the end of the list.

23.4 Orderable Same-Name Siblings

If a node supports orderable child nodes *and* same-name siblings then the order of the nodes within a set of same-name siblings must be persisted and be re-orderable by the client. For example, given the following initial ordering of child nodes,

```
[A, B, C, A, D]
```

a call to

```
orderBefore("A[2]","A[1]")
```

will cause the child node currently called `A[2]` to be moved to the position before the child node currently called `A[1]`, the resulting order will be:

```
[A, A, B, C, D]
```

where the first `A` is the one that was formerly after `C` and the second `A` is the one that was formerly at the head of the list.

Note, however, that after the completion of this operation *the indices of the two nodes have now switched*, due to their new positions relative to each other. What was formerly `A[2]` is now `A[1]` and what was formerly `A[1]` is now `A[2]`.

23.5 Non-orderable Child Nodes

When a node does not support orderable child nodes this means that it is left up to the implementation to maintain the order of child nodes. Applications should not, in this case, depend on the order of child nodes returned by `Node.getNodes()`, as it may change at any time.

23.6 Properties are Never Orderable

Properties are never client orderable, the order in which properties are returned by `Node.getProperties()` is always maintained by the implementation and can change at any time.

24 Repository Compliance

A JCR implementation must support the *basic repository features*:

- Repository acquisition and user authentication and authorization (see §4 *Connecting*)
- Reading through path, identifier and browse access (see §5 *Reading*)
- Query (see §6 *Query*)
- Export (see §7 *Export*)
- Node Type Discovery (see §8 *Node Type Discovery*)
- Permission and capability checking (see §9 *Permissions and Capabilities*)

These features must be supported by all JCR repositories.

In addition, a repository *may* support any subset of the *additional features* defined in sections §10 to §23.

The presence of each additional feature is individually testable either through querying the value of a repository descriptor (see §24.2 *Repository Descriptors*) or testing for the availability of a specific node type (see §24.3 *Node Type-Related Features*), thus allowing an application to programmatically determine the capabilities of a specific JCR implementation.

An implementation that supports all the additional features defined in this specification is a *fully-compliant repository*.

24.1 Definition of Support

By indicating support for testable feature, a repository asserts that it fully conforms to the semantics of that feature as defined in this specification, with two possible exceptions:

- aspects of the feature clearly indicated as being optional (i.e., *should*, *may*, *should not*), and
- aspects of the feature testable by their own repository descriptors (for example, whether a repository supports joins is separately testable from whether it supports searches in general).

However, to indicate that it supports a testable feature, a repository is only required to support that feature in some, not all, contexts. For example, a repository may restrict its support for a feature based on access control, path, or other criteria.

24.2 Repository Descriptors

Repository descriptors are used to test support for repository features that have a behavioral (as opposed to a data-model) aspect.

Each descriptor is identified by a unique *key*, which is a string. An implementation must recognize all the standard keys defined in this specification and may recognize additional implementation-specific keys. The full set of valid keys (both standard and implementation-specific) for an implementation is returned by

```
String[] Repository.getDescriptorKeys().
```

The method

```
boolean Repository.isStandardDescriptor(String key)
```

returns `true` if `key` is the name of a standard descriptor defined within this specification and `false` if it is either a valid implementation-specific key or not a valid key.

The method

```
boolean Repository.isSingleValueDescriptor(String key)
```

returns `true` if `key` is the name of a single value descriptor and `false` otherwise.

The value of a particular descriptor is found by passing that descriptor's key to either

```
Value Repository.getDescriptorValue(String key)
```

or

```
Value[] Repository.getDescriptorValues(String key).
```

depending on whether that key is defined to return a single or a multiple value.

The JCR 1.0 method

```
String Repository.getDescriptor()
```

is still supported as a convenience method. The call

```
String s = repository.getDescriptor(key);
```

is equivalent to

```
Value v = repository.getDescriptorValue(key);
String s = (v == null) ? null : v.getString();
```

24.2.1 Repository Information

Key	Descriptor
`SPEC_VERSION_DESC`	`STRING`: The version of the specification that this repository implements. For JCR 2.0 the value of this descriptor is "2.0".
`SPEC_NAME_DESC`	`STRING`: The name of the specification that this repository implements. For JCR 2.0 the value of this descriptor is "Content Repository for Java Technology API".

REP_VENDOR_DESC	STRING: The name of the repository vendor.
REP_VENDOR_URL_DESC	STRING: The URL of the repository vendor.
REP_NAME_DESC	STRING: The name of this repository implementation.
REP_VERSION_DESC	STRING: The version of this repository implementation.

24.2.2 General

Key	Descriptor
WRITE_SUPPORTED	BOOLEAN: Returns `true` if and only if repository content can be *updated* through the JCR API , as opposed to having read-only access (see §10 *Writing*).
IDENTIFIER_STABILITY	STRING: Returns one of the following `javax.jcr.Repository` constants indicating the *stability of non-referenceable identifiers*: • IDENTIFIER_STABILITY_METHOD_DURATION: Identifiers may change between method calls • IDENTIFIER_STABILITY_SAVE_DURATION: Identifiers are guaranteed stable within a single save/refresh cycle. • IDENTIFIER_STABILITY_SESSION_DURATION: Identifiers are guaranteed stable within a single session. • IDENTIFIER_STABILITY_INDEFINITE_DURATION: Identifiers are guaranteed to be stable forever. Note that *referenceable* identifiers always have this level of stability. See 3.7 *Identifiers* and §3.8 *Referenceable Nodes*.
OPTION_XML_IMPORT_SUPPORTED	BOOLEAN: Returns `true` if and only if *XML import* is supported (see §11 *Import*).
OPTION_UNFILED_CONTENT_SUPPORTED	BOOLEAN: Returns `true` if and only if *unfiled content* is supported (see §3.12 *Unfiled Content*).
OPTION_SIMPLE_VERSIONING_ SUPPORTED	BOOLEAN: Returns `true` if and only if *simple versioning* is supported (see §3.13 *Versioning Model* and §15 *Versioning*).
OPTION_ACTIVITIES_SUPPORTED	BOOLEAN: Returns `true` if and only if *activities* are supported (see §15.12 *Activities*).
OPTION_BASELINES_SUPPORTED	BOOLEAN: Returns `true` if and only if *configurations and baselines* are supported (see §3.13 *Versioning Model* and §15.13 *Configurations and Baselines*).

Key	Descriptor
OPTION_ACCESS_CONTROL_SUPPORTED	BOOLEAN: Returns true if and only if *access control* is supported (see §16 *Access Control Management*).
OPTION_LOCKING_SUPPORTED	BOOLEAN: Returns true if and only if *locking* is supported (see §17 *Locking*).
OPTION_OBSERVATION_SUPPORTED	BOOLEAN: Returns true if and only if *asynchronous observation* is supported (see §12 *Observation*).
OPTION_JOURNALED_OBSERVATION_SUPPORTED	BOOLEAN: Returns true if and only if *journaled observation* is supported (see §12 *Observation*).
OPTION_RETENTION_SUPPORTED	BOOLEAN: Returns true if and only if *retention and hold* are supported (see §20 *Retention and Hold*).
OPTION_LIFECYCLE_SUPPORTED	BOOLEAN: Returns true if and only if *lifecycle management* is supported (see §18 *Lifecycle Management*).
OPTION_TRANSACTIONS_SUPPORTED	BOOLEAN: Returns true if and only if *transactions* are supported (see §21 *Transactions*).
OPTION_WORKSPACE_MANAGEMENT_SUPPORTED	BOOLEAN: Returns true if and only if *workspace management* is supported (see §13 *Workspace Management*).
OPTION_NODE_AND_PROPERTY_WITH_SAME_NAME_SUPPORTED	BOOLEAN: Returns true if and only if *node and property with same name* is supported (see §5.1.8 *Node and Property with Same Name*).

24.2.3 Node Operations

Key	Descriptor
OPTION_UPDATE_PRIMARY_NODE_TYPE_SUPPORTED	BOOLEAN: Returns true if and only if *the primary node type of an existing node can be updated* (see §10.10.2 *Updating a Node's Primary Type*).
OPTION_UPDATE_MIXIN_NODE_TYPES_SUPPORTED	BOOLEAN: Returns true if and only if *the mixin node types of an existing node can be added and removed* (see §10.10.3 *Assigning Mixin Node Types*).
OPTION_SHAREABLE_NODES_SUPPORTED	BOOLEAN: Returns true if and only if *the creation of shareable nodes* is supported (see §3.9 *Shareable Nodes Model* and §14 *Shareable Nodes*).

24.2.4 Node Type Management

These repository descriptors characterize the types of nodes an API consumer may register (see §19 *Node Type Management*). They do not constrain a repository's built-in node types (see §3.7 *Node Types*).

Key	Descriptor
OPTION_NODE_TYPE_MANAGEMENT_SUPPORTED	BOOLEAN: Returns true if and only if *node type management* is supported.
NODE_TYPE_MANAGEMENT_INHERITANCE	STRING: Returns one of the following javax.jcr.Repository constants indicating *the level of support for node type inheritance*: • NODE_TYPE_MANAGEMENT_INHERITANCE_MINIMAL: Registration of primary node types is limited to those which have only nt:base as supertype. Registration of mixin node types is limited to those without any supertypes. • NODE_TYPE_MANAGEMENT_INHERITANCE_SINGLE: Registration of primary node types is limited to those with exactly one supertype. Registration of mixin node types is limited to those with at most one supertype. • NODE_TYPE_MANAGEMENT_INHERITANCE_MULTIPLE: Primary node types can be registered with one or more supertypes. Mixin node types can be registered with zero or more supertypes.
NODE_TYPE_MANAGEMENT_OVERRIDES_SUPPORTED	BOOLEAN: Returns true if and only if *override of inherited property or child node definitions* is supported (see §3.7.6 *Node Type Inheritance*).
NODE_TYPE_MANAGEMENT_PRIMARY_ITEM_NAME_SUPPORTED	BOOLEAN: Returns true if and only if *primary items* are supported (see §3.7.1.7 *Primary Item*).
NODE_TYPE_MANAGEMENT_ORDERABLE_CHILD_NODES_SUPPORTED	BOOLEAN: Returns true if and only if *preservation of child node ordering* is supported (see §5.2.2.1 *Child Node Order Preservation*).
NODE_TYPE_MANAGEMENT_RESIDUAL_DEFINITIONS_SUPPORTED	BOOLEAN: Returns true if and only if *residual property and child node definitions* are supported (see §3.7.2.1.2 *Item Definition Name and Residual Definitions*).
NODE_TYPE_MANAGEMENT_AUTOCREATED_DEFINITIONS_SUPPORTED	BOOLEAN: Returns true if and only if *autocreated properties and child nodes* are supported (see §3.7.2.3 *Auto-Created*).
NODE_TYPE_MANAGEMENT_SAME_NAME_SIBLINGS_SUPPORTED	BOOLEAN: Returns true if and only if *same-name sibling child nodes* are supported (see §3.7.4.3 *Same-Name Siblings*).
NODE_TYPE_MANAGEMENT_PROPERTY_TYPES	LONG[]: Returns an array holding the javax.jcr.PropertyType constants for the property types (including UNDEFINED, if supported) that a registered node type can specify, or a zero-length array if registered node types cannot specify property definitions (see §3.6.1 *Property*

	Types).
NODE_TYPE_MANAGEMENT_MULTIVALUED_PROPERTIES_SUPPORTED	boolean: Returns true if and only if *multi-value properties* are supported (see §3.6.3 *Single and Multi-Value Properties*).
NODE_TYPE_MANAGEMENT_MULTIPLE_BINARY_PROPERTIES_SUPPORTED	BOOLEAN: Returns true if and only if *registration of a node types with more than one BINARY property* is permitted (see §3.6.1.7 *BINARY*).
NODE_TYPE_MANAGEMENT_VALUE_CONSTRAINTS_SUPPORTED	BOOLEAN: Returns true if and only *value-constraints* are supported (see §3.7.3.6 *Value Constraints*).
NODE_TYPE_MANAGEMENT_UPDATE_IN_USE_SUPORTED	BOOLEAN: Returns true if and only the update of node types is supported for node types currently in use as the type of an existing node in the repository.

24.2.5 Query

Key	Descriptor
QUERY_LANGUAGES	STRING[]: Returns an array holding the constants representing the supported query languages, or a zero-sized array if query is not supported (see §6 *Query*).
QUERY_STORED_QUERIES_SUPPORTED	BOOLEAN: Returns true if and only if *stored queries* are supported (see §6.9.7 *Stored Query*).
QUERY_FULL_TEXT_SEARCH_SUPPORTED	BOOLEAN: Returns true if and only if *full-text search* is supported (see §6.7.19 *FullTextSearch*).
QUERY_JOINS	STRING: Returns one of the following javax.jcr.Repository constants indicating *the level of support for joins in queries*: • QUERY_JOINS_NONE: Joins are not supported. Queries are limited to a single selector. • QUERY_JOINS_INNER: Inner joins are supported. • QUERY_JOINS_INNER_OUTER: Inner and outer joins are supported. See §6.7.5 *Join*.

24.2.6 Deprecated Descriptors

Key	Descriptor
LEVEL_1_SUPPORTED	BOOLEAN: Returns true if and only if

	• OPTION_XML_EXPORT_SUPPORTED = true and • QUERY_LANGUAGES is of non-zero length. These semantics are identical to those in JCR 1.0. This constant is **deprecated**.
LEVEL_2_SUPPORTED	BOOLEAN: Returns true if and only if • LEVEL_1_SUPPORTED = true, • WRITE_SUPPORTED = true and • OPTION_XML_IMPORT_SUPPORTED = true. These semantics are identical to those in JCR 1.0. This constant is **deprecated**.
OPTION_QUERY_SQL_SUPPORTED	BOOLEAN: Returns true if and only if the (deprecated) JCR 1.0 SQL query language is supported . This constant is **deprecated**.
QUERY_XPATH_POS_INDEX	BOOLEAN: Returns false unless the (deprecated) JCR 1.0 XPath query language is supported. If JCR 1.0 XPath is supported then this descriptor has the same semantics as in JCR 1.0. This constant is **deprecated**.
QUERY_XPATH_DOC_ORDER	BOOLEAN: Returns false unless the (deprecated) JCR 1.0 XPath query language is supported. If JCR 1.0 XPath is supported then this descriptor has the same semantics as in JCR 1.0. This constant is **deprecated**.

24.2.7 Implementation-Specific Descriptors

Implementers are free to introduce their own descriptors. The descriptor keys should use Java package-style names in namespaces controlled by the implementer. The `Repository.isStandardDescriptor` method must return false for these keys.

24.3 Node Type-Related Features

The node type registry is used to test support for features which correspond to a JCR-defined node type. For example, support for *referenceable nodes* as a feature is equivalent to support for the node type `mix:referenceable`. Such features are more data model-oriented than the behavioral features reported by descriptors.

Testing for the availability of a particular node type is done using

```
boolean NodeTypeManager.hasNodeType(String nodeTypeName)
```

Any node types associated with a particular feature are described in the section describing that feature.

The presence of the indicated node types in the node type registry (tested with `NodeTypeManager.hasNodeType`, see §8.1 *NodeTypeManager Object*) indicates support for the corresponding feature.

Node Type	Feature
`mix:referenceable`	Referenceable nodes (see §3.8 *Referenceable Nodes*).

`mix:created`
`mix:mimeType`
`mix:lastModified`
`mix:title`
`mix:language`
`nt:hierarchyNode`
`nt:file`
`nt:linkedFile`
`nt:folder`
`nt:resource`
`nt:address`
	Standard application node types, a repository can support a subset (see §3.7.11 *Standard Application Node Types*).

`mix:etag`	Entity tags (see §3.7.12 *Entity Tags*).
`nt:unstructured`	Unstructured content (see §3.7.13 *Unstructured Content*).

`nt:nodeType`
`nt:propertyDefinition`
`nt:childNodeDefinition`
	Node type definition storage in content (see §3.7.14 *Node Type Definition Storage*).

24.4 Implementation Issues

JCR adapters built against some existing repositories may require a connection to the back-end repository to determine whether a feature is supported. Using methods on `Repository` (as opposed, for example, to methods on `Session`) to test support for a feature is therefore potentially problematic. However, several approaches are open to such adapters:

- Establish a transient connection to the back-end (for example, using service-to-service authentication or as "guest") to determine support for a feature.
- Determine the features supported by the back-end upon application deployment, and store this in configuration file locally available to the JCR adapter at runtime.
- Report the feature set supported by the type of back end, which may be a superset of the feature set supported by the specific instance of that back-end type.

25 Appendix

25.1 Treatment of Identifiers

A number of different methods in the API transfer node state from one location to another. They often differ in how they treat the identifier of the node. Some methods always behave the same way in this regard, others have various options that control their behavior. The following table summarizes the behaviors of the methods.

<table>
<tr><th>Method</th><th>Referenceable Identifiers</th><th>Non-referenceable Identifiers</th></tr>
<tr><td>Save</td><td colspan="2">Identifiers must be preserved, with the possible exception of the first save of a new node (see §3.7.1 Identifier Assignment). The state of a transient node is saved to the persistent node with the same identifier.</td></tr>
<tr><td>Copy
(within a workspace)</td><td colspan="2">New identifiers must be created.</td></tr>
<tr><td>Copy
(between workspaces)</td><td>New referenceable identifiers must be created.</td><td>New non-referenceable identifiers may be created. The stability of non-referenceable identifiers is a repository implementation variant.</td></tr>
<tr><td>Move</td><td>Referenceable identifiers must be preserved.</td><td rowspan="3">Non-referenceable identifiers may be preserved. The stability of non-referenceable identifiers is a repository implementation variant.</td></tr>
<tr><td>Clone, Restore</td><td>Referenceable identifiers must be preserved. On conflict with an existing node a flag governs whether the existing node is removed or an exception thrown.</td></tr>
<tr><td>Update, Merge</td><td>Referenceable identifiers must be preserved. On conflict with an existing node, that node is replaced at its existing location in the target workspace.</td></tr>
<tr><td>Import</td><td colspan="2">A flag determines whether new identifiers are created or incoming ones preserved. On conflict with an existing node the options are to either replace the existing node in place, remove the existing node, or throw an exception.</td></tr>
</table>

25.2 Compact Node Type Definition Notation

The following grammar defines the compact node type definition (CND) notation used to define node types throughout this specification.

25.2.1 String Literals in CND Grammar

Throughout this section string literals that appear in the syntactic grammar defining CND must be interpreted as specified in §1.3.1 *String Literals in Syntactic Grammars*.

25.2.2 Variant Node Type Definitions

In a CND, the presence of a question mark ("`?`") indicates that an attribute in question can vary across repository implementations (see §3.7.10 *Base Primary Node Type* and 3.7.11 *Standard Application Node Types*).

In the case of the queryable node type attribute, the absence of an explicit keyword (either `query` or `noquery`) indicates that the attribute is a variant.

Such *variant node type definitions* cannot be instantiated in a repository as-is. If an implementation supports a variant node type its node type registry must contain a definition of that node type in which each variant attribute is resolved to a concrete value.

25.2.3 CND Grammar

```
/* A CND consists of zero or more blocks, each of which is
   either a namespace declaration or a node type definition.
   Namespace prefixes referenced in a node type definition
   block must be declared in a preceding namespace declaration
   block. */
Cnd ::= {NamespaceMapping | NodeTypeDef}

/* A namespace declaration consists of prefix/URI pair. The
   prefix must be a valid JCR namespace prefix, which is the
   same as a valid XML namespace prefix. The URI can in fact be
   any string. Just as in XML, it need not actually be a URI,
   though adhering to that convention is recommended. */
NamespaceMapping ::= '<' Prefix '=' Uri '>'
Prefix ::= String
Uri ::= String

/* A node type definition consists of a node type name followed
   by an optional supertypes block, an optional node type
   attributes block and zero or more blocks, each of which is
   either a property or child node definition. */
NodeTypeDef ::= NodeTypeName [Supertypes]
                [NodeTypeAttribute {NodeTypeAttribute}]
                {PropertyDef | ChildNodeDef}

/* The node type name is delimited by square brackets and must
   be a valid JCR name. */
NodeTypeName ::= '[' String ']'

/* The list of supertypes is prefixed by a '>'. If the node
   type is not a mixin then it implicitly has nt:base as a
   supertype even if neither nt:base nor a subtype of nt:base
```

```
   appears in the list or if this element is absent. A question
   mark indicates that the supertypes list is a variant. */
Supertypes ::= '>' (StringList | '?')

/* The node type attributes are indicated by the presence or
   absence of keywords. */
NodeTypeAttribute ::= Orderable | Mixin | Abstract | Query |
                      PrimaryItem

/* In the following, mention of a keyword, like 'orderable',
   refers to all the forms of that keyword, including short
   forms ('ord' and 'o', for example) */

/* If 'orderable' is present without a '?' then orderable child
   nodes is supported. If 'orderable' is present with a '?'
   then orderable child nodes is a variant. If 'orderable'
   is absent then orderable child nodes is not supported. */
Orderable ::= ('orderable' | 'ord' | 'o') ['?']

/* If 'mixin' is present without a '?' then the node type is a
   mixin. If 'mixin' is present with a '?' then the mixin
   status is a variant. If 'mixin' is absent then the node type
   is primary. */
Mixin ::= ('mixin' | 'mix' | 'm') ['?']

/* If 'abstract' is present without a '?' then the node type is
   abstract. If 'abstract' is present with a '?' then the
   abstract status is a variant. If 'abstract' is absent then
   the node type is concrete. */
Abstract ::= ('abstract' | 'abs' | 'a') ['?']

/* If 'query' is present then the node type is
   queryable. If 'noquery' is present then the node type is
   not queryable. If neither query nor noquery are present then
   the queryable setting of the node type is a variant. */
Query ::= ('noquery' | 'nq') | ('query' | 'q' )

/* If 'primaryitem' is present without a '?' then the string
   following it is the name of the primary item of the node
   type. If 'primaryitem' is present with a '?' then
   the primary item is a variant. If 'primaryitem' is absent
   then the node type has no primary item. */
PrimaryItem ::= ('primaryitem'| '!')(String | '?')

/* A property definition consists of a property name element
   followed by optional property type, default values, property
   attributes and value constraints elements. */
PropertyDef ::= PropertyName [PropertyType] [DefaultValues]
                [PropertyAttribute {PropertyAttribute}]
                [ValueConstraints]

/* The property name, or '*' to indicate a residual property
   definition, is prefixed with a '-'. */
PropertyName ::= '-' String

/* The property type is delimited by parentheses ('*' is a
   synonym for UNDEFINED). If this element is absent,
   STRING is assumed. A '?' indicates that this attribute is
   a variant. */
PropertyType ::= '(' ('STRING' | 'BINARY' | 'LONG' | 'DOUBLE' |
                     'BOOLEAN' | 'DATE' | 'NAME' | 'PATH' |
```

```
                        'REFERENCE' | 'WEAKREFERENCE' |
                        'DECIMAL' | 'URI' | 'UNDEFINED' | '*' |
                        '?') ')'

/* The default values, if any, are listed after a '='. The
   attribute is a list in order to accommodate multi-
   value properties. The absence of this element indicates that
   there is no static default value reportable. A '?' indicates
   that this attribute is a variant */
DefaultValues ::= '=' (StringList | '?')

/* The value constraints, if any, are listed after a '<'. The
   absence of this element indicates that no value constraints
   reportable within the value constraint syntax. A '?'
   indicates that this attribute is a variant */
ValueConstraints ::= '<' (StringList | '?')

/* A child node definition consists of a node name element
   followed by optional required node types, default node types
   and node attributes elements. */
ChildNodeDef ::= NodeName [RequiredTypes] [DefaultType]
                 [NodeAttribute {NodeAttribute}]

/* The node name, or '*' to indicate a residual property
   definition, is prefixed with a '+'. */
NodeName ::= '+' String

/* The required primary node type list is delimited by
   parentheses. If this element is missing then a required
   primary node type of nt:base is assumed. A '?' indicates
   that the this attribute is a variant. */
RequiredTypes ::= '(' (StringList | '?') ')'

/* The default primary node type is prefixed by a '='. If this
   element is missing then no default primary node type is set.
   A '?' indicates that this attribute is a variant */
DefaultType ::= '=' (String | '?')

/* The property attributes are indicated by the presence or
   absence of keywords. */
PropertyAttribute ::= Autocreated | Mandatory | Protected |
                      Opv | Multiple | QueryOps | NoFullText |
                      NoQueryOrder

/* The node attributes are indicated by the presence or
   absence of keywords. */
NodeAttribute ::= Autocreated | Mandatory | Protected |
                  Opv | Sns

/* If 'autocreated' is present without a '?' then the item
   is autocreated. If 'autocreated' is present with a '?' then
   the autocreated status is a variant. If 'autocreated' is
   absent then the item is not autocreated. */
Autocreated ::= ('autocreated' | 'aut' | 'a' )['?']

/* If 'mandatory' is present without a '?' then the item
   is mandatory. If 'mandatory' is present with a '?' then
   the mandatory status is a variant. If 'mandatory' is
   absent then the item is not mandatory. */
Mandatory ::= ('mandatory' | 'man' | 'm') ['?']
```

```
/* If 'protected' is present without a '?' then the item
   is protected. If 'protected' is present with a '?' then
   the protected status is a variant. If 'protected' is
   absent then the item is not protected. */
Protected ::= ('protected' | 'pro' | 'p') ['?']

/* The OPV status of an item is indicated by the presence of
   that corresponding keyword. If no OPV keyword is present
   then an OPV status of COPY is assumed. If the keyword 'OPV'
   followed by a '?' is present then the OPV status of the item
   is a variant.
Opv ::= 'COPY' | 'VERSION' | 'INITIALIZE' | 'COMPUTE' |
        'IGNORE' | 'ABORT' | ('OPV' '?')

/* If 'multiple' is present without a '?' then the property
   is multi-valued. If 'multiple' is present with a '?' then
   the multi-value status is a variant. If 'multiple' is
   absent then the property is single-valued. */
Multiple ::= ('multiple' | 'mul' | '*') ['?']

/* The available query comparison operators are listed after
   the keyword 'queryops'. If 'queryops' is followed by a '?'
   then this attribute is a variant. If this element is absent
   then the full set of operators is available. */
QueryOps ::= ('queryops' | 'qop')
             (('''Operator {','Operator}''') | '?')
Operator ::= '=' | '<>' | '<' | '<=' | '>' | '>=' | 'LIKE'

/* If 'nofulltext' is present without a '?' then the property
   does not support full text search. If 'nofulltext' is
   present with a '?' then this attribute is a variant. If
   'nofulltext' is absent then the property does support full
   text search. */
NoFullText ::= ('nofulltext' | 'nof') ['?']

/* If 'noqueryorder' is present without a '?' then query
   results cannot be ordered by this property. If
   'noqueryorder' is present with a '?' then this attribute is
   a variant. If 'noqueryorder' is absent then query results
   can be ordered by this property. */
NoQueryOrder ::= ('noqueryorder' | 'nqord') ['?']

/* If 'sns' is present without a '?' then the child node
   supports same-name siblings. If 'sns' is present with a '?'
   then this attribute is a variant. If 'sns' is absent then
   the child node does support same-name siblings. */
Sns ::= ('sns' | '*') ['?']

/* Strings */
StringList ::= String {',' String}
String ::= QuotedString | UnquotedString

/* Quotes are used to allow for strings (i.e., names, prefixes,
   URIs, values or constraint strings) with characters that
   would otherwise be interpreted as delimiters in CND. */
QuotedString ::= SingleQuotedString | DoubleQuotedString

/* Within a SingleQuotedString, single quote literals (') must
   be escaped. */
SingleQuotedString ::= ''' UnquotedString '''
```

```
/* Within a DoubleQuotedString, double quote literals (") must
   be escaped. */
DoubleQuotedString ::= '"' UnquotedString '"'
UnquotedString ::= XmlChar {XmlChar}
XmlChar ::= /* see §3.2.2 Local Names */
```

25.2.3.1 Case Insensitive Keywords

The keywords of CND, though defined above as terminal strings with specific cases, are in fact case-insensitive. For example, `STRING` can be written `string`, `String` or even `StRiNg`.

25.2.3.2 Escaping

The standard Java escape sequences are supported:

`\n` newline

`\t` tab

`\b` backspace

`\f` form feed

`\r` return

`\"` double quote

`\'` single quote

`\"` double quote

`\\` back slash

`\u`*`HHHH`* Unicode character in hexadecimal

25.2.3.3 Comments

Comments can be included in the notation using either of the standard Java forms. A comment is defined as:

```
Comment ::= LineComment | BlockComment
LineComment ::= "//" LineCommentText
BlockComment ::= "/*" BlockCommentText "*/"
LineCommentText ::= /* Any text ending in a newline */
BlockComment ::=  /* Any text except the end-block-comment
                     character pair */
```

A comment can appear between any two valid tokens of the CND grammar. Comments are not defined within the main CND grammar, but are intended to be stripped during preprocessing, prior to the actual parsing of the CND.

25.2.3.4 Extension Syntax

Vendor-specific extensions are supported through the extension syntax:

```
VendorExtension ::= "{" Vendorname VendorBody "}"
```

```
VendorName ::= /* A unique vendor-specific identifier
                  containing no whitespace */
VendorBody ::= /* Any string not including "}" */
```

Like a comment, an extension can appear between any two tokens of the CND grammar. Extensions are not defined within the main CND grammar, but are intended to be handled during preprocessing, prior to the actual parsing of the CND. The first whitespace-delimited token of the extension should be a unique vendor-specific identifier. The semantics of the extension body are implementation-specific.

25.2.3.5 Whitespace and Short Forms

The notation can be compacted by taking advantage of the following the fact that spacing around keychars ([ ] > , - ( ) = <), newlines and indentation are not required. So, the following is also well-formed:

```
[x]>y,z orderable mixin -p(DATE)=a,b primary mandatory
autocreated protected multiple VERSION <c,d
```

Additionally, though spaces are required around the keywords (orderable, mixin, date, mandatory, etc.), short forms for keywords can be used. So, this:

```
[x]>y,z o m-p(DATE)=a,b ! m a p * VERSION <c,d
```

is also well-formed.

25.2.4 Examples

Here is a "worst-case scenario" example that demonstrates all the features of the notation:

```
/*  An example node type definition */

// The namespace declaration
<ns = 'http://namespace.com/ns'>

// Node type name
[ns:NodeType]

// Supertypes
> ns:ParentType1, ns:ParentType2

// This node type is abstract
abstract

// This node type supports orderable child nodes
orderable

// This is a mixin node type
mixin

// This node type is not queryable
noquery

// ex:property is the primary item
primaryitem ex:property

// A property called 'ex:property' of type STRING
```

```
- ex:property (STRING)

// The default values for this (multi-value) property are...
= 'default1', 'default2'

// This property is...
mandatory autocreated protected

// ...and multi-valued.
multiple

// It has an on-parent-version setting of...
VERSION

// The constraint settings are...
< 'constraint1', 'constraint2'

// The supported query operators are...
queryops '=, <>, <, <=, >, >=, LIKE'

// The property is not full text searchable
nofulltext

// query results are not orderable by this property
noqueryorder

// A child node called ns:node which must be of
// at least the node types ns:reqType1 and ns:reqType2
+ ns:node (ns:reqType1, ns:reqType2)

// with default primary node type is...
= ns:defaultType

// This node is...
mandatory autocreated protected

// supports same name siblings
sns

// and has an on-parent-version setting of ...
VERSION
```