
Content Repository API for Java™ Technology Specification

Java Specification Request 170

version 0.16.3

4 March 2005

1 PREFACE	
1.1 Documents Included	10
2 INTRODUCTION	
2.1 Motivation	11
2.2 Goals	11
3 USE CASES	
3.1 Swappability	13
3.2 Resource Crunch (Personalization)	14
4 THE REPOSITORY MODEL	
4.1 API Basics	17
4.1.1 Traversal Access	18
4.1.2 Direct Access	18
4.1.3 Writing to the Repository	19
4.1.4 Nodes, Properties and Items	22
4.2 Compliance Levels	22
4.3 Same-Name Siblings	23
4.3.1 Index Notation	23
4.3.2 Support for Same Name Siblings is Optional	24
4.3.3 Properties Cannot Have Same Name Siblings	24
4.4 Orderable Child Nodes	24
4.4.1 Orderable Same Name Siblings	24
4.4.2 Non-orderable Child Nodes	25
4.4.3 Orderable Child Node Support is Optional	25
4.4.4 Properties are Never Orderable	25
4.5 Namespaces	25
4.6 Path Syntax	26
4.6.1 Names vs. Paths	27
4.7 Properties	27
4.7.1 Multi-Value Properties	28
4.7.2 Reference, Path and Name Property Types	28
4.7.3 No Null Values	29
4.8 Node Types	29
4.9 Referenceable Nodes	30
4.10 Workspaces	31
4.10.1 Single Workspace Repositories	32
4.10.2 Multiple Workspaces and Corresponding Nodes	33
4.11 Versioning	36
4.12 Metadata	39
4.13 Hierarchical versus Direct Access	39
5 EXAMPLE IMPLEMENTATIONS	
5.1 JCR on top of a File System	41
5.2 JCR on top of WebDAV	42
5.3 JCR on top of a Database	43
5.4 JCR on top of a File System with Structured Content	44
5.5 Namespace Prefixes in the Examples	45
6 LEVEL 1 REPOSITORY FEATURES	
6.1 Accessing the Repository	47
6.1.1 Repository	47
6.1.2 Credentials	49
6.2 Reading Repository Content	49
6.2.1 Session Read Methods	49
6.2.2 Workspace Read Methods	49
6.2.3 Node Read Methods	49
6.2.4 Property Read Methods	49
6.2.5 Property Types	49

6.2.6 Property Type Conversion	49
6.2.7 Value	49
6.2.8 Item Read Methods	49
6.2.9 Example	49
6.3 Namespaces	49
6.3.1 Namespace Registry	49
6.3.2 Prefix Syntax	49
6.3.3 Session Namespace Remapping	49
6.3.4 Internal Storage of Names and Paths	49
6.4 XML Mappings	49
6.4.1 System View XML Mapping	49
6.4.2 Document View XML Mapping	49
6.4.3 Escaping of JCR Names	49
6.4.4 Escaping of JCR Values	49
6.5 Exporting Repository Content	49
6.5.1 Encoding	49
6.6 Searching Repository Content	49
6.6.1 XPath in JCR	49
6.6.2 XPath over Document View	49
6.6.3 Adapting XPath to JCR	49
6.6.4 XPath Extensions	49
6.6.5 XPath Grammar	49
6.6.6 Search Scope	49
6.6.7 Query API	49
6.6.8 QueryManager	49
6.6.9 The Query Object	49
6.6.10 Persistent vs. Transient Queries	49
6.6.11 Query Results	49
6.6.12 Permissions	49
6.7 Node Types	49
6.7.1 Node Type Configuration	49
6.7.2 What Constitutes a Node Type	49
6.7.3 Node Type Discovery in Level 1	49
6.7.4 Primary and Mixin Node Types	49
6.7.5 Special Properties <i>jcr:primaryType</i> and <i>jcr:mixinTypes</i>	49
6.7.6 Property Definitions	49
6.7.7 Child Node Definitions	49
6.7.8 Inheritance Among Node Types	49
6.7.9 Discovering available Node Types	49
6.7.10 Discovering the Node Types of a Node	49
6.7.11 Discovering the Definition of a Node Type	49
6.7.12 ItemDef	49
6.7.13 PropertyDef	49
6.7.14 NodeDef	49
6.7.15 Residual Definitions	49
6.7.16 Value Constraints	49
6.7.17 Automatic Item Creation	49
6.7.18 Discovery of Constraints on Existing Items	49
6.7.19 Predefined Node Types	49
6.7.20 Storage of Node Type Definitions in Content	49
6.7.21 Predefined Mixin Node Types	49
6.7.22 Predefined Primary Node Types	49
6.8 System Node	49
7 LEVEL 2 REPOSITORY FEATURES	
7.1 Writing Repository Content	49
7.1.1 Saving by UUID and Path	49
7.1.2 Reflecting Item State	49
7.1.3 Adding Nodes	49
7.1.4 Adding and Writing Properties	49
7.1.5 Removing Nodes and Properties	49
7.1.6 Moving and Copying	49
7.1.7 Updating and Cloning Nodes across Workspaces	49

7.1.8 Referenceable Nodes	49
7.1.9 Treatment of UUIDs	49
7.1.10 Ordering Child Nodes	49
7.2 Adding and Deleting Namespaces	49
7.3 Importing Repository Content	49
7.3.1 Import from System View	49
7.3.2 Import from Document View	49
7.3.3 Respecting Property Semantics	49
7.3.4 Workspace Import Methods	49
7.3.5 Session Import Methods	49
7.3.6 Importing <i>jcr:root</i>	49
7.4 Assigning Node Types	49
7.4.1 The Special Properties <i>jcr:primaryType</i> and <i>jcr:mixinTypes</i>	49
7.4.2 Assigning a Primary Node Type	49
7.4.3 Assigning Mixin Node Types	49
7.4.4 Automatic Addition and Removal of Mixins	49
7.4.5 Serialization and Node Types	49
8 OPTIONAL REPOSITORY FEATURES	
8.1 Transactions	49
8.1.1 Container Managed Transactions: Sample Request Flow	49
8.1.2 User Managed Transactions: Sample Code	49
8.1.3 Save vs. Commit	49
8.2 Versioning	49
8.2.1 Versionable Nodes	49
8.2.2 Version Storage	49
8.2.3 The Base Version	49
8.2.4 Initializing the Version History	49
8.2.5 Check In	49
8.2.6 Check Out	49
8.2.7 Restoring a Version	49
8.2.8 Restoring a Group of Versions	49
8.2.9 Update	49
8.2.10 Merge	49
8.2.11 OnParentVersion Attribute	49
8.2.12 The OnParentVersionAction Class	49
8.2.13 Removal of Versions	49
8.2.14 Versioning API	49
8.2.15 Serialization of Version Storage	49
8.2.16 Versioning within a Transaction	49
8.3 Observation	49
8.3.1 Event Listeners	49
8.3.2 Listener Registration	49
8.3.3 Observation Manager	49
8.3.4 Event Production	49
8.3.5 Event Filtering	49
8.3.6 Event Bundles	49
8.3.7 Interpretation of Events	49
8.3.8 Deserializing Content	49
8.3.9 External Mechanisms	49
8.3.10 Location of Listeners	49
8.3.11 Persistence of Event Listeners	49
8.3.12 Vetoable Event Listeners	49
8.3.13 Exceptions	49
8.4 Access Control	49
8.4.1 JAAS	49
8.4.2 Checking Permissions	49
8.5 Locking	49
8.5.1 Discovery of Lock Capabilities	49
8.5.2 Lockable	49
8.5.3 Shallow and Deep Locks	49
8.5.4 Lock Owner	49

8.5.5 Placing and Removing a Lock	49
8.5.6 Lock Token	49
8.5.7 Session-scoped and Open-scoped Locks	49
8.5.8 Effect of a Lock	49
8.5.9 Timing Out	49
8.5.10 Locks and Transactions	49
8.5.11 Locking Methods	49
8.5.12 The Lock Object	49
8.5.13 Session Methods Related to the Lock Token	49
8.6 Searching Repository Content with SQL	49
8.6.1 The SQL Language	49
8.6.2 Database View	49
8.6.3 SQL EBNF	49
8.6.4 SQL Syntax in Detail	49
8.6.5 Query Results	49

Acknowledgements

This specification is the collaborative product of

David Nuescheler (specification lead, Day Software),

Peeter Piegaze (author, Day Software),

and other members of the JSR 170 expert group, including

Tim Anderson (Intalio),

Gordon Bell (Hummingbird),

Geoffery Clemm (IBM),

David Choy (IBM),

Jeff Collins (Vignette),

Stefan Guggisberg (Day Software),

Stefano Mazzocchi (Apache Software Foundation),

James Myers (Pacific Northwest National Laboratories),

James Owen (BEA),

Franz Pfeifroth (Fujitsu),

David Pitfield (Oracle),

Corprew Reed (FileNet),

Victor Spivak (Documentum),

David B. Victor (IBM),

as well as many others who contributed with corrections and suggestions.

License

Day Management AG is willing to license this specification to you ONLY UPON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THIS LICENSE AGREEMENT ("Agreement"). Please read the terms and conditions of this Agreement carefully. By downloading this Specification, you accept the terms and conditions of the Agreement. If you are not willing to be bound by it, select the "decline" button at the bottom of this page and the downloading process will not continue.

Content Repository API for Java™ Technology Specification ("Specification")

Version: 0.16.3

Status: pre-FCS

Specification Lead: Day Management AG ("Specification Lead")

Release: 4 March 2005

Copyright 2005 Day Management AG, Barfüsserplatz 6, 4001 Basel, Switzerland.

All rights reserved.

NOTICE

The Specification is protected by copyright and the information described therein may be protected by one or more patents or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of Specification Lead and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this Agreement.

Subject to the terms and conditions of this license, Specification Lead hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense) under Specification Lead's intellectual property rights to review the Specification only for the purposes of evaluation. This license includes the right to discuss the Specification (including the right to provide limited excerpts of text to the extent relevant to the point[s] under discussion) with other licensees (under this or a substantially similar version of this Agreement) of the Specification. Other than this limited license, you acquire no right, title or interest in or to the Specification or any other Specification Lead intellectual property, and the Specification may only be used in accordance with the license terms set forth herein. This license will expire on the earlier of: (i) two (2) years from the date of Release listed above; (ii) the date on which the final version of the Specification is publicly released; or (iii) the date on which the Java Specification Request (JSR) to which the Specification corresponds is withdrawn.

In addition, this license will terminate immediately without notice from Specification Lead if you fail to comply with any provision of this license. Upon termination, you must cease use of or destroy the Specification.

TRADEMARKS

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

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS" AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH CANNOT OR WILL NOT BE CORRECTED BY SPECIFICATION LEAD. SPECIFICATION LEAD MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. 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. SPECIFICATION LEAD 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.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SPECIFICATION LEAD OR ITS LICENSORS 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 SPECIFICATION LEAD AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will hold Specification Lead (and its licensors) harmless from any claims based on your use of the Specification for any purposes other than the limited right of evaluation as described above, and from any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

REPORT

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your evaluation of the Specification ("Feedback"). To the extent that you provide Specification Lead with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Specification Lead 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 Preface

This is version 0.16.3 of the Content Repository API for Java Technology (Java Specification Request 170). This version is an update to version 0.16.2 of the specification, which was the version originally submitted as the proposed final draft, as defined by the Java Community Process version 2.6.

1.1 Documents Included

The specification includes:

- This document in Adobe Portable Document Format (*jsr170-0.16.3-pfd.pdf*).
- This document in Microsoft Word Document Format (*jsr170-0.16.3-pfd.doc*) with "Track Changes" information showing changes between the previous version (0.16.2) and this version (0.16.3).
- A change log (*changes-0.16.2-to-0.16.3.txt*) describing changes between 0.16.2 and 0.16.3.
- The Java source code for the API package `javax.jcr` and subpackages.
- A jar file of the API package `javax.jcr` and subpackages.
- The Javadoc produced from the `javax.jcr` and subpackages.

In case of a discrepancy between this document and the Javadoc produced from the `javax.jcr` package, this document should be considered normative.

2 Introduction

2.1 Motivation

As the number of vendors offering proprietary content repositories has increased, the need for a common programmatic interface to these repositories has become apparent. The aim of the Java Content Repository (JCR) API specification is to provide such an interface and, in doing so, lay the foundations for a true industry-wide content infrastructure.

Application developers and custom solution integrators will be able to avoid the costs associated with learning the particular API of each repository vendor. Instead, programmers will be able to develop content-based application logic independently of the underlying repository architecture or physical storage.

Customers will also benefit by being able to exchange their underlying repositories without touching any of the applications built on top of them.

2.2 Goals

The guiding principles governing the design of the JCR API are:

It should not be tied to any particular underlying architecture, data source or protocol.

The API is, of course, essentially a set of Java interfaces, which can be implemented in a wide variety of ways. Hence, achieving this goal is not difficult in itself. The main challenge here is to allow enough flexibility in the API so that it can be used for both hierarchical and non-hierarchical repository models. This is done by providing for both *hierarchical, path-based addressing* of content items and *direct, UUID-based addressing*.

It should be easy to use from the programmer's point of view.

To this end, the API is designed to be as simple and straightforward as possible. In particular, it has a simple object model and concentrates on representing the core functionality of a content repository without venturing into areas that might be regarded as "content applications".

It should allow for relatively easy implementation on top of as wide a variety of existing content repositories as possible.

A concerted effort was made to ensure that it would be relatively easy to implement the API (especially at level 1, see below) on top of the repositories of most major vendors.

However, it should also standardize some more complex functionality needed by advanced content-related applications.

Recognizing that a tension exists between this aim and the previous one, this specification has been split into two compliance levels.

Level 1 defines a read-only repository. This includes functionality for the reading of repository content, introspection of content-type definitions, basic support for namespaces, export of content to XML and searching. This functionality should meet the needs of presentation templates and basic portal applications comprising about 80% of the existing code-base of content-related applications.

Level 2 additionally defines methods for writing content, assignment of types to content, further support for namespaces, and importing content from XML.

Finally, a number of independently optional features are defined that a compliant repository may support. These are transactions, versioning, observation, access control, locking and additional support for searching.

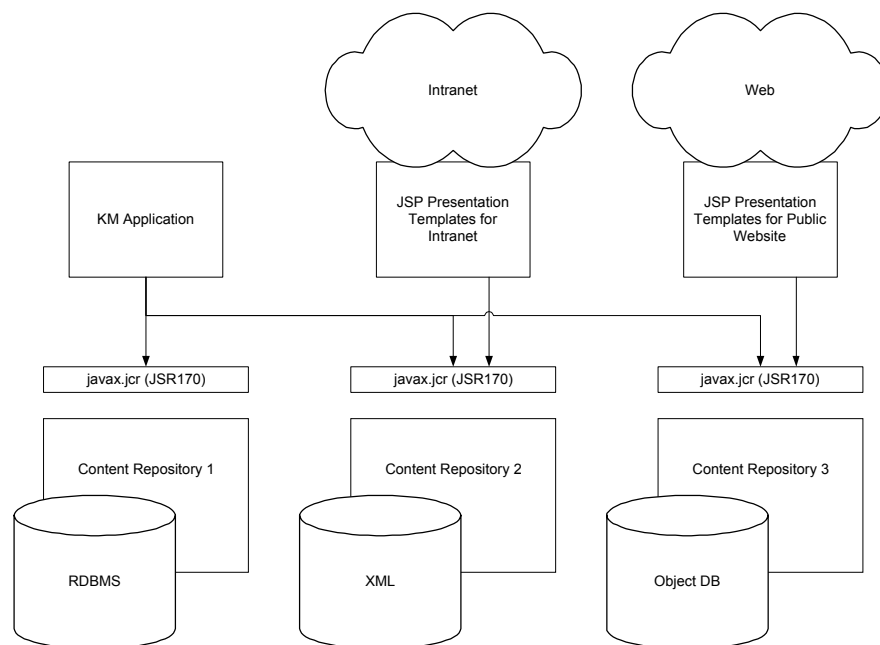
3 Use Cases

3.1 Swappability

ENT Corporation, a large distributed enterprise, has different content management systems in different divisions. Their Knowledge Management (KM) team has developed an idea for better discovery of corporate assets across the various repositories.

The team implements the application using the Java Content Repository (JCR) interface to the content management (CM) systems. As the KM team finds more CM systems across the enterprise, the application can easily harvest the new data as well as that from the existing systems.

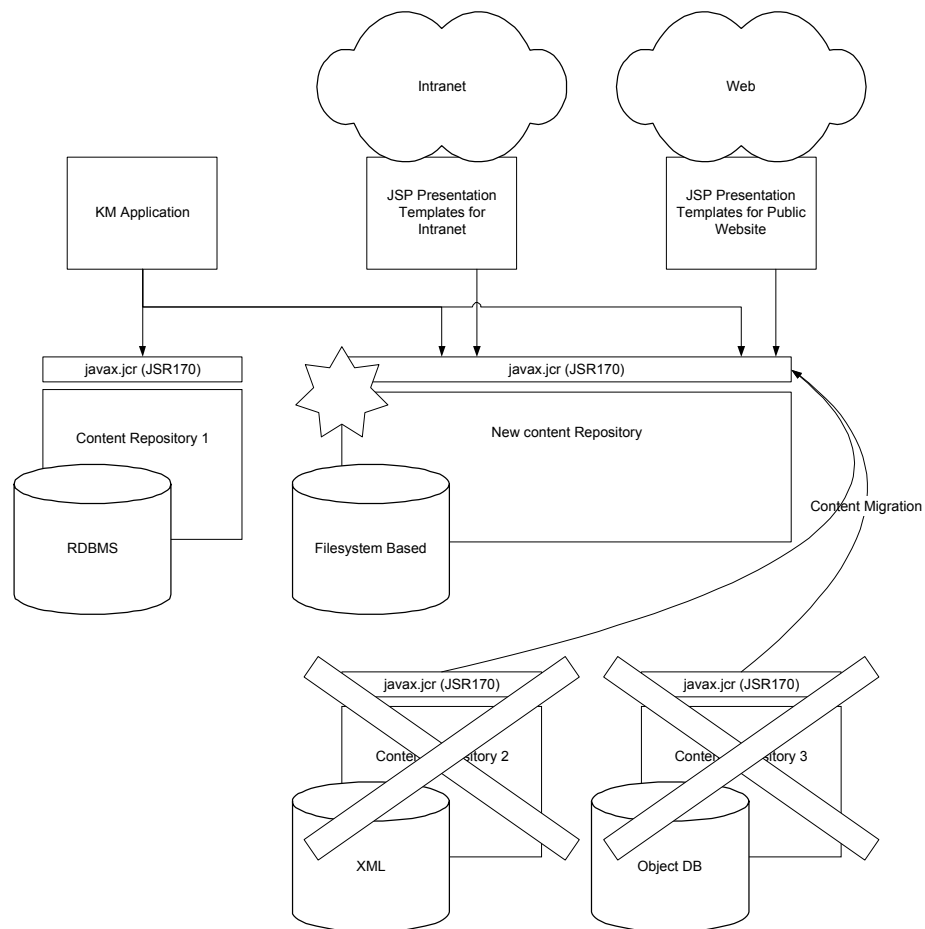
Additionally, the presentation templates that cover the corporate design guidelines for the organization have been developed based on the JCR specification in a Java framework around JSP.



A year later, two divisions switch their CM vendors. As part of the migration process, the KM discovery application is tested against the new CMS in those divisions before the site is made completely live. Other than having its configuration file updated, the application works as before.

Through its standardized interface, the JCR specification also allows simple migration of the content from the two old repositories to the new one; this means that the content can be converted from the

source repository to the newly purchased destination repository extremely easily.

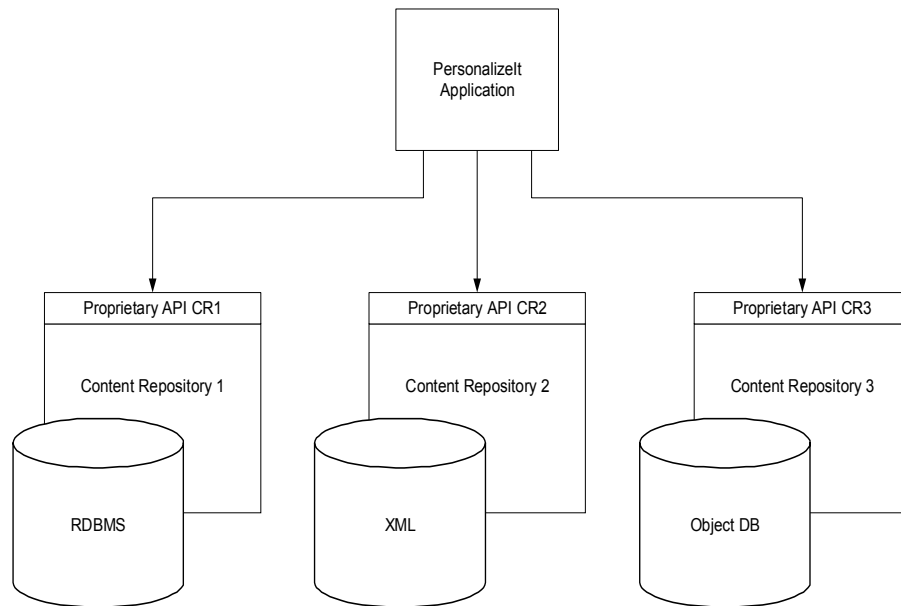


Furthermore, all the content presentation templates developed against the JCR specification keep working as before. In the case of ENT the development of this presentation logic was one of the major investments made when the intranet and the public Internet website were first developed. Being able to migrate this logic painlessly to work on top of the new repository is therefore of great benefit.

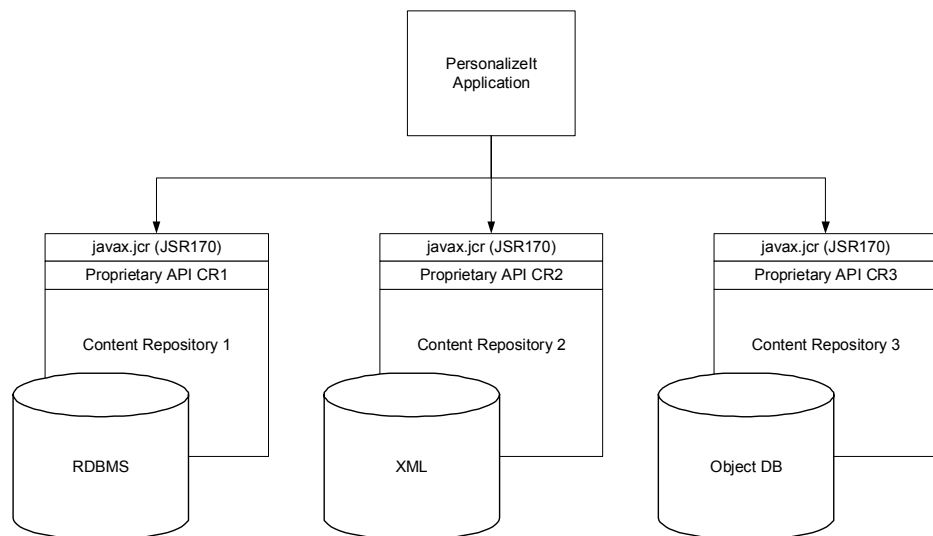
3.2 Resource Crunch (Personalization)

PersonalizeIt Software Corp., a vendor of personalization and portal software, needs to gain access to meta-information and content that is aggregated and managed by content management systems.

Since, historically, the CM market has been quite diverse, PersonalizeIt has had to integrate into many different proprietary APIs. This meant that PersonalizeIt had to spend a large amount of time familiarizing their developers with these APIs and had to maintain compatibility with all the different APIs as the various CM vendors' products evolved.

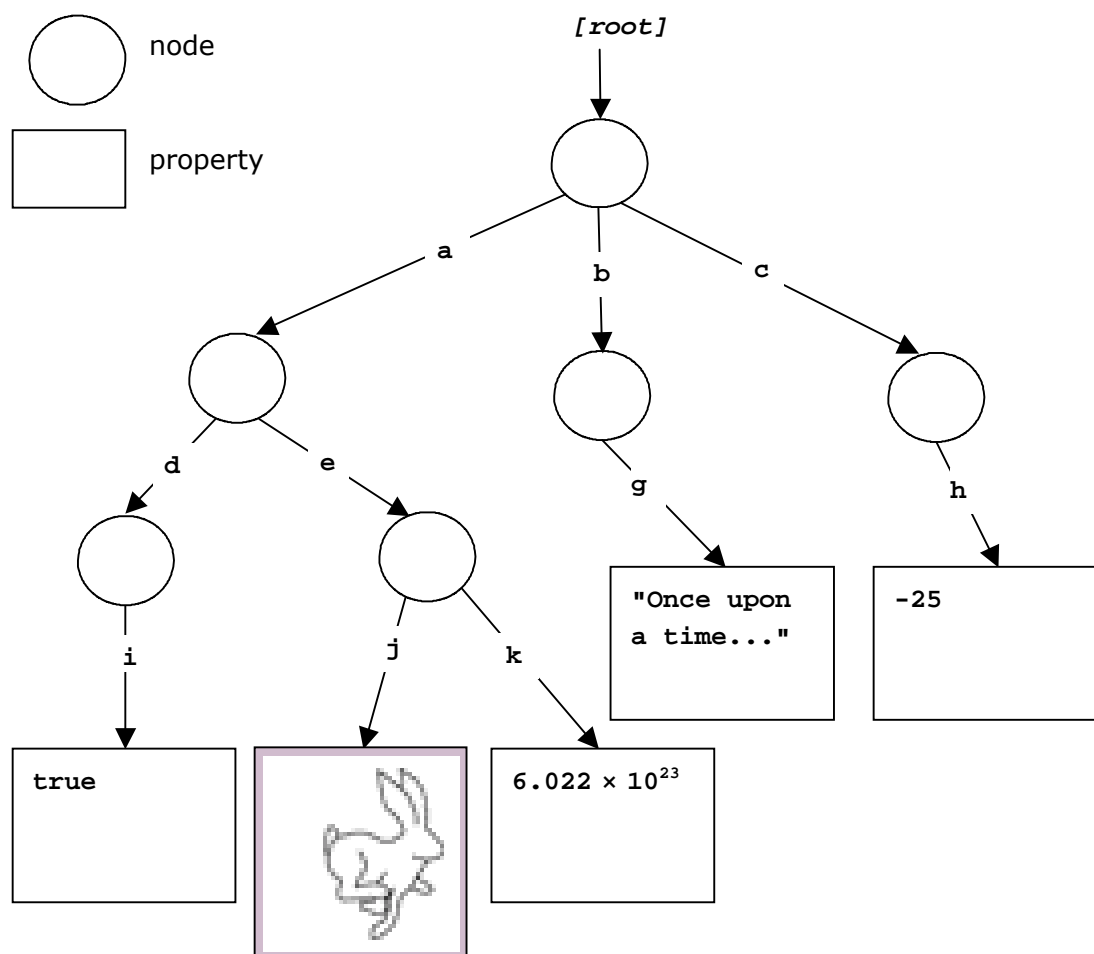


Since the adoption of the JCR specification, PersonalizeIt offers a standard interface to JCR-compliant repositories. Therefore, they no longer have to maintain dozens of proprietary integrations and can concentrate on their core competency.



4 The Repository Model

A JCR repository consists of one or more *workspaces*, each of which contains a tree of *items*. An item is either a *node* or a *property*. Each node may have zero or more child nodes and zero or more child properties. There is a single root node, which has no parent. All other nodes have one parent. Properties have one parent (a node) and cannot have children; they are the leaves of the tree. A property is the smallest unit of content, being a name-value pair. All of the actual content in the repository is stored within the values of the properties.



In the diagram above, we see the single root node of some workspace with child nodes **a**, **b** and **c**, each of which have further child nodes or properties. For example, the node **a** has two child nodes, **d** and **e**. Node **e**, in turn, has two properties, **j** and **k**. The property **j** contains an image (a picture of a rabbit) and **k** contains a floating-point number (6.022×10^{23}). Similarly, the property **i** contains a boolean value (**true**), the property **g** contains a string ("Once upon a time...") and the property **h** contains an integer (-25).

Any item in the hierarchy can be identified by an *absolute path*. For example, the path `/` refers to the root node and the path `/a/d/i` refers to the property with value `true`. Absolute paths always begin with a `/` character.

A *relative path* specifies a node or property relative to another location within the hierarchy. For example, relative to node `/a` in the above diagram, the path to the property with value `true` is `d/i`. The Unix-style path segments `."` and `.."` (meaning respectively, "this" and "parent") are also supported so that, relative to `/a`, the property containing the value `-25` would be `../c/h`. Relative paths are distinguished from absolute paths by having *no* leading `/` character.

4.1 API Basics

The repository as a whole is represented by a `Repository` object¹. A client connects to the repository by calling `Repository.login` specifying a workspace name and a `Credentials` object². The client then receives a `Session` object tied to the specified workspace.

How the `Repository` object is actually acquired is beyond the scope of this specification. However, one possibility, as shown below, is to use JNDI, though this depends entirely on the implementation. Similarly, the specifics of acquiring the `Credentials` object are also not specified. Below we show one possibility: using the class `SimpleCredentials` included in the specification. Implementations may provide their own `Credentials` classes as well.

```
// Get the Repository object
Repository repository =
    (Repository) java.rmi.Naming.lookup("MyRepo");

// Get a Credentials object
Credentials credentials =
    new SimpleCredentials("MyName",
                        "MyPassword".toCharArray());

// Get a Session
Session mySession =
    repository.login(credentials, "MyWorkspace");
```

Through the `session` object the client can access any node or property in that tree of the workspace to which the `session` is tied.

¹ Unless otherwise noted, all references to "objects" refer to instances of Java classes implementing the indicated Java interface from the `javax.jcr` package. For example, the term "`Repository` object" means an instance of a class implementing the interface `javax.jcr.Repository`.

² There are other signatures of `login` as well. See [6.1.1 Repository](#) for details.

The API provides methods for both traversing the tree step by step and for directly accessing a particular node.

4.1.1 Traversal Access

Traversal access typically begins with

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

From the returned root node, successive levels of child nodes can be accessed with

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

which takes a relative path (so it can skip down or up multiple levels as well). There is also a similar method for accessing properties,

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

The data stored in the property can be accessed either directly with methods like

```
String Property.getString(),
```

or in the form of a type-neutral wrapper, the `Value` object, using

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

The actual data can then be retrieved with, for example,

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

Here is an example code snippet:

```
// Get the root node
Node root = mySession.getRootNode();

// Traverse to the node you want
Node myNode = root.getNode("a/e");

// Retrieve a property of myNode
Property myProperty = myNode.getProperty("k");

// Get the value of the property
Value myValue = myProperty.getValue();

// Convert the value to the desired type
double myDouble = myValue.getDouble();

// The variable myDouble will contain the
// value 6.022 x 10^23
```

4.1.2 Direct Access

The most important direct access method is

```
Item Session.getItem(String abspath).
```

This method takes an absolute path and is used to jump directly to the indicated item (node or property):

```
// Directly get the property with
// the value of Avogadro's Number
// (i.e.,  $6.022 \times 10^{23}$ )
Property myProperty =
    (Property)mySession.getItem("/a/e/k");

// Directly convert to a double
double myDouble = myProperty.getDouble();

// The variable myDouble will contain the
// value  $6.022 \times 10^{23}$ 
```

Another direct access method (though for nodes only, not properties) is,

```
Node Session.getNodeByUUID(String uuid).
```

This method can be used to access those nodes that have Universally Unique Identifiers (see section [4.9 Referenceable Nodes](#)):

```
// Assuming that the node /a/e is referenceable
// and has UUID 1111 2222 3333 4444, we get it
Node myNode =
    mySession.getNodeByUUID("1111 2222 3333 4444");

// and then get the property and convert it to a double
double myDouble = myNode.getProperty("k").getDouble();

// The variable myDouble will contain the
// value  $6.022 \times 10^{23}$ 
```

4.1.3 Writing to the Repository

If the repository is level 2 compliant (see [4.2 Compliance Levels](#)) then, having acquired a session, the client can write to the repository by adding or removing nodes and properties or changing the values of properties.

For example, the client can retrieve a node, add a child node to it and add a property to that child node:

```
// Retrieve a node
Node myNode = (Node) mySession.getItem("/a/e");

// Add a child node
Node newNode = myNode.addNode("n");

// Add a property
newNode.setProperty("x", "Hello");

// Persist the changes
mySession.save();
```

The node `myNode` has the path `/a/e`, so the new node will have path `/a/e/n` and the new property will have the path `/a/e/n/x` and the value `"Hello"`.

4.1.3.1 Removing Items

To erase an item, the method `Item.remove()` is used. For example, continuing from the above code segment, the following code,

```
// Remove the node /a/e/n (and its subtree)
newNode.remove();

// Persist the changes
mySession.save();
```

would result in the node at `/a/e/n` (and its child property `/a/e/n/x`) being erased.

In the case of `Properties`, an alternative to `remove` is to set the property to `null`. This can be done in two ways, by calling `setValue` with `null` on the property itself, or by calling `setProperty` with the property name and a `null` value on the property's parent node:

```
// Assume we have node /m and two
// properties /m/p and /m/q
Node m = (Node) mySession.getItem("/m");
Property p = m.getProperty("p");

//Remove p by calling setValue on p itself
p.setValue((Value)null);

//Remove q by calling setProperty on q's parent node
m.setProperty("q", (Value)null);

// Persist the changes
mySession.save();
```

See [4.7.3 No Null Values](#).

4.1.3.2 Transient Storage in the Session

Notice the use of the method `session.save` in the above examples. This method is needed because changes made through most methods of `Session`, `Node`³ or `Property` are not immediately reflected in the persistent workspace. The changes are held in transient storage associated with the `session` object until they are

³ A few `Node` methods act immediately on the persistent workspace and do not require `save`. See [6.3 Writing to the Repository](#) for details.

either persisted (using `Session.save` or `Item.save`) or discarded (using `Session.refresh` or `Item.refresh`).

Changes not yet saved or discarded are called *pending changes*. Pending changes are immediately visible through the session that made them but are not visible through other sessions accessing the same workspace.

`Session.save` validates and, if validation succeeds, persists all pending changes currently stored in the `Session` object, making them visible to other sessions (though this only applies if the `save` is not within the scope of a transaction, see [4.1.3.3 Transactions](#), below). Conversely, `Session.refresh(false)` discards all pending changes currently stored in the `Session`.

For more fine-grained control over which changes are persisted or discarded, the methods `Item.save` and `Item.refresh` are also provided. `Item.save` saves all pending changes in the `Session` that apply to that particular item or its subtree. Analogously, `Item.refresh(false)` discards all pending changes that apply to that item or its subtree. See [7.1 Writing Repository Content](#).

4.1.3.3 Transactions

Throughout this document, any discussion of “persistence of changes upon `save`” or “immediate persistence of changes through methods that do not require `save`” refers to cases in which the `save` is performed *outside the scope of a transaction*.

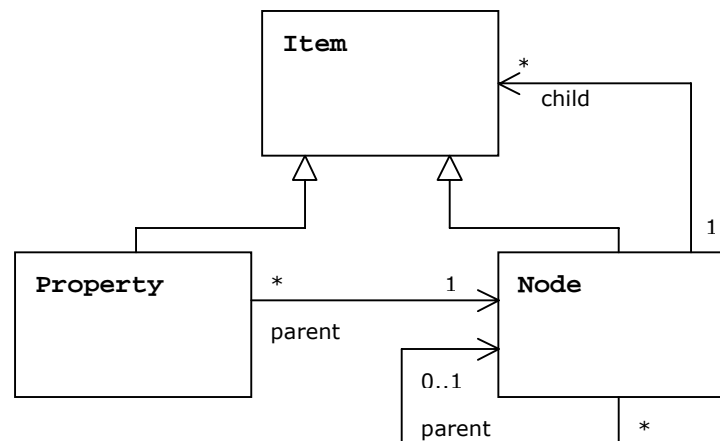
Within the scope of a transaction, `save` will not make pending changes visible to other sessions; this will only occur when the transaction is committed.

However, even within the scope of a transaction, `save` still performs validation and, if successful, clears pending changes from the `Session`. As well, `refresh(false)` still clears pending changes from the `Session`.

When a transaction commits, it persists only those changes that have been saved; it does *not* automatically `save` pending changes *and then* commit them as well. After a commit, pending changes remain in the `Session` and may be saved and committed later. Note that support for transactions is optional. See [8.1 Transactions](#).

4.1.4 Nodes, Properties and Items

Because nodes and properties have some common functionality, common methods are defined in the interface **Item**, to which the sub-interfaces **Node** and **Property** add further methods. The following diagram summarizes the basic relationships between the interfaces.



This UML diagram indicates that **Property** and **Node** are subinterfaces of **Item**. A single **Property** has one and only one parent **Node**. A **Node** can have either zero parents (only if it is the root node) or one parent. A **Node** can have any number of child **Item** objects (i.e., either **Property** or **Node** objects).

4.2 Compliance Levels

The JCR specification is divided into two compliance levels with a set of additional optional features which repositories of either level may support. Level 1 provides for read functions and level 2 adds additional write functions. The functional division is as follows:

Level 1 includes:

- Retrieval and traversal of nodes and properties
- Reading the values of properties
- Transient namespace remapping
- Export to XML/SAX
- Query facility with XPath syntax
- Discovery of available node types

Level 2 adds:

- Adding and removing nodes and properties
- Writing the values of properties
- Persistent namespace changes

- Import from XML/SAX
- Assigning node types to nodes
- Discovery of access control permissions

Optional:

Any combination of the following features may be added to an implementation of either level.

- Transactions
- Versioning
- Observation (Events)
- Access Control
- Locking
- SQL syntax for query

4.3 Same-Name Siblings

A particular node *may*, in some cases, have *same-name siblings*, that is, other nodes that share its parent and have the same name. Whether a particular node allows this depends on the *child-node definition* that applies to it (this definition is part of the node type of that node's parent, see section [6.7](#) Node Types).

The standard method for retrieving a set of such nodes is `Node.getNodes(String namePattern)` which returns an iterator over all the child nodes of the calling node that have the specified pattern (by making `namePattern` just a name, without wildcards, we can get all the nodes with that exact name, see section [6.2.3](#) Node Read Methods).

4.3.1 Index Notation

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 practice stems from the need to allow XPath-based queries on the repository (see [6.6](#) Searching Repository Content).

When a name in a path does not specify an index, an implicit index of 1 is assumed. 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()`. In nodes that support orderable child nodes this order will be

persistent and client-controlled, otherwise it is maintained by the implementation. See [4.4 Orderable Child Nodes](#).

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.

4.3.2 Support for Same Name Siblings is Optional

As mentioned, whether or not a particular node allows multiple child items with the same name is governed by the *node type* of that particular node. See [6.7 Node Types](#).

Though there is a required set of node types that every compliant repository must support, none of these required node types allow same-name siblings and any further node types available in a particular repository are implementation-specific. Therefore, it is possible for a repository to disallow same-name siblings altogether by restricting the set of available node types.

4.3.3 Properties Cannot Have Same Name Siblings

Properties cannot have sibling properties of the same name. However, they may have multiple values. See [4.7.1, Multi-Value Properties](#), below.

4.4 Orderable Child Nodes

Some nodes may support client-controlled ordering of their child nodes. Whether a particular node preserves the order of its child nodes is governed by its node type. See [6.7 Node Types](#).

If a node supports orderable child nodes, the order of its child nodes, as reflected in the iterator acquired through `Node.getNodes()` can be controlled by the client using the method `Node.orderBefore`. The order of child nodes is persisted upon `save` of the parent node.

When a child node is added to a node that supports orderable child nodes it is added to the end of the list. It can then be re-ordered using the above method.

4.4.1 Orderable Same Name Siblings

If the parent node supports orderable child nodes *and* same-name siblings then the same-name sibling child nodes will be client-orderable just like an other child nodes. For example, given the following initial ordering of child nodes,

[A, B, C, A, D]

a call to

```
orderBefore("A[1]","A[2]")
```

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].

4.4.2 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.

4.4.3 Orderable Child Node Support is Optional

Like same name siblings, support for orderable child nodes depends on the range of node types available in a particular repository. Orderable child nodes are not mandated by any required node types, and any additional node types are implementation-specific. Therefore, orderable child node support is, in effect, optional.

4.4.4 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.

4.5 Namespaces

The name of a node or property may have a *prefix*, delimited by a single ':' (colon) character that indicates the *namespace* of the item.

Namespacing in JCR is patterned after namespacing in XML. As in XML, the prefix is actually shorthand for the full namespace, which is a URI. URIs are used as namespaces in order to minimize naming collisions. Every JCR-compliant (level 1 or 2) repository has a namespace registry. The namespace registry always contains at least the following built-in namespace prefixes:

- **jcr** Reserved for items defined within built-in node types.
- **nt** Reserved for the names of built-in primary node types.
- **mix** Reserved for the names of built-in mixin node types.
- **sv** Reserved for use in XML serialization.
- **""** (the empty prefix) This indicates the default namespace.

In level 1 repositories the prefix assigned to an existing registered namespace (a URI) may be temporarily over-ridden by another prefix within the scope of a particular **session**. Level 2 repositories have, additionally, the capability to add, remove and change the set of namespaces (URIs) stored in the namespace registry (excluding the built-in namespaces). See section [6.3](#), *Namespaces*, for more details.

4.6 Path Syntax

A syntactically valid path is:

```
path ::= properpath ['/' ]
properpath ::= abspath | relpath
abspath ::= '/' relpath
relpath ::= pathelement | relpath '/' pathelement
pathelement ::= name | name '[' number ']' | '..' | '.'
number ::= /* An integer > 0 */
name ::= simplename | prefixedname
simplename ::= onecharsimplename |
              twocharsimplename |
              threeormorecharname
prefixedname ::= prefix ':' localname
localname ::= onecharlocalname |
              twocharlocalname |
              threeormorecharname
onecharsimplename ::= (* Any Unicode character except:
                       '.', '/', ':', '[', ']', '*',
                       "'", '"', '|' or any whitespace
                       character *)
twocharsimplename ::= '.' onecharsimplename |
                    onecharsimplename '.' |
                    onecharsimplename onecharsimplename
onecharlocalname ::= nonspace
twocharlocalname ::= nonspace nonspace
threeormorecharname ::= nonspace string nonspace
```

```

prefix ::= (* Any valid XML Name *)

string ::= char | string char

char ::= nonspace | ' '

nonspace ::= (* Any Unicode character except:
               '/', ':', '[', ']', '*',
               '''', '""', '|' or any whitespace
               character *)

```

4.6.1 Names vs. Paths

A “name” in JCR terminology is a path element *without any square-bracket index*. For example, `myapp:paragraph` is a *name* and a valid *relative path* (of depth 1) whereas, `myapp:paragraph[3]` is not a name, it is only a relative path.

Names and paths are not simply strings with certain syntax. They have special semantics in that they respect the namespace mappings of the current `session` (see [6.3 Namespaces](#)). The special property types `NAME` and `PATH` are provided to enable the storage of these values in the repository in a namespace-sensitive way (see [6.2.5 Property Types](#) as well as [4.7 Properties](#), immediately below).

4.7 Properties

Every property is of one of the following types:

- `PropertyType.STRING`
- `PropertyType.BINARY`
- `PropertyType.DATE`
- `PropertyType.LONG`
- `PropertyType.DOUBLE`
- `PropertyType.BOOLEAN`
- `PropertyType.NAME`
- `PropertyType.PATH`
- `PropertyType.REFERENCE`

Methods are provided to read (in level 1) and write (in level 2) the values of properties to and from the appropriate native Java types (i.e. `PropertyType.STRING` values can be read and written as `java.lang.String` objects, `PropertyType.LONG` values into Java `long` variables, and so on). In addition, a generic class `javax.jcr.Value` is provided which can wrap a value of any `PropertyType`.

4.7.1 Multi-Value Properties

In some cases, a property may have more than one value. A property that may have more than one value is referred to as a multi-valued property (regardless of whether it currently has one or more than one value).

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.

The values within a multi-valued property are ordered.

Accessing the values of such a property is done with the method `Property.getValues`, which returns an array of `Value` objects that contains the values in their prescribed order.

Accessing a multi-valued property with `Property.getValue`, or a single-value property with `Property.getValues` will throw a `ValueFormatException`.

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

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 indexes of those values with indexes greater than that of the removed value by -1.

Note that this *does* mean that a multi-value property can have *no values* (i.e., be an empty array), whereas a single-value property either has a (non-null) value or does not exist.

See [7.1.4 Adding and Writing Properties](#) for more details.

4.7.2 Reference, Path and Name Property Types

Three of the property types listed above have special semantics: **REFERENCE**, **PATH** and **NAME**.

NAME properties are used for storing strings that are namespace-qualified, such as the names of node types or the names of repository items. A **NAME** property can be thought of as a namespace-aware **STRING**. It is set like a string (for example, `setProperty("aNodeType", "nt:file")`). However, the prefix is automatically mapped to its current URI and the value is stored using that full namespace URI. When the property is later read the mapping is reversed and if the URI in question has been remapped that remapping is reflected in the returned value.

A **PATH** property represents a path in a workspace (either relative or absolute) and therefore can also be used to refer to items

elsewhere in the workspace. However, the **PATH** property does not enforce referential integrity; in other words it can point to a location where no item currently exists. Like a **NAME** property, a **PATH** is also namespace-aware in that its apparent value when read will always reflect the current prefix to URI mapping.

A **REFERENCE** property is used to provide a named reference to a node elsewhere in the workspace. The value of the property is the UUID of the node to which it refers. Consequently, only a *referenceable node* can be the target of a **REFERENCE** property (see [4.9 Referenceable Nodes](#)). **REFERENCE** properties have the additional semantic feature of maintaining referential integrity by preventing the removal of any node that is currently the target of a reference property. To remove a node that is the target of a **REFERENCE**, one must first remove the **REFERENCE**. The check for referential integrity is done when an attempt is made to save the removal of a node (it is not done immediately on **remove**). The method `Node.getReferences()` can be used to find all **REFERENCE** properties that refer to a particular node. The method `Node.setProperty(Node value)` can be used to set the value of a **REFERENCE** property to the UUID of the specified node.

In versioning repositories the version storage is exposed in the workspace tree as a protected subtree below `jcr:system/jcr:versionStorage` (see [8.2.2 Version Storage](#)). Within this subtree, the referential integrity requirement is lifted for **REFERENCE** properties stored as part of the frozen state of a version (see [8.2.2.9 Reference Properties within a Version](#)).

4.7.3 No Null Values

In a JCR repository, 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. In the case of multi-value properties, the setting of a particular value within the array to null results in the removal of that value and the compacting of the array. As a result it is possible to have a multi-value property with no values (an empty array). See [7.1.4 Adding and Writing Properties](#).

4.8 Node Types

A key feature of JCR is the support for node types. Every node in a JCR repository *must* have one and only one *primary node type*. The primary node type defines the names, types and other characteristics of the properties and child nodes that this node is allowed (or required) to have. Every node in the repository has a special property called `jcr:primaryType` that records the name of its primary node type.

In addition to its primary node type, a node *may* also have one or more *mixin types*. These are node type definitions that mandate

extra characteristics (i.e., more child nodes, properties and their respective names and types) for a particular node in addition to those enforced by its primary node type. When a node is assigned a mixin node type, it acquires a special multi-value property called `jcr:mixinTypes` that records its mixin node types.

Level 1 of the specification provides methods for discovering the node types of existing nodes, and for discovering and reading the definitions of node types available in the repository.

Level 2 of the specification provides methods for assigning primary and mixin node types to nodes.

The specification does not attempt to provide methods for defining, creating or managing primary or mixin node types.

This specification also provides a set of predefined primary and mixin node types; some required and some optional. See [6.7 Node Types](#).

4.9 Referenceable Nodes

The concept of the *referenceable node* is foundational to many features of the repository, including *multiple workspaces* and *versioning*. The following principles define the characteristics and functioning of referenceable nodes:

- A repository *may* support *referenceable nodes*. To do this the repository must support the mixin type `mix:referenceable`.
- The `mix:referenceable` type has the effect of enforcing, on any node to which it is assigned, the presence of a property called `jcr:uuid`.
- The `jcr:uuid` property is a protected, auto-created, mandatory property. This means that it is created and administered by the system and can only be read (but not changed or deleted) by the client.
- The job of the `jcr:uuid` property is to expose the universally unique identifier (UUID) of its node.
- The UUID of a referenceable node is assigned on node creation (or at least on node persistence) by the system itself.
- In a given workspace, there is never more than one node with a given UUID, though there may be nodes that are not `mix:referenceable` and so do not have UUIDs at all.
- If a node that is not `mix:referenceable` happens to have a property called `jcr:uuid`, then this property has no special significance (Note that adding such a property is not recommended: In general, the `jcr` prefix should be reserved for

items defined within the specification, though this restriction may not be enforced by the implementation).

- A repository implementation may make its workspace root nodes **mix:referenceable**. If so, then the root node of *all* workspaces must be referenceable, and all must have the *same* UUID.

Some implementations may allow or even require every node to have a UUID, and hence be **mix:referenceable**. In some cases however, especially where JCR is being implemented on top of an existing datastore, the provision of UUIDs for every node may be impractical. For this reason, an implementation is free to enforce whatever policies it wishes regarding where within a workspace tree referenceable nodes may be created or existing nodes extended with an assignment of **mix:referenceable** (in fact, this stems from the more general principle that an implementation is free to enforce such policies on the assignment of primary and mixin node types in general).

4.9.1.1 When UUIDs are Assigned

In some client-server implementations the assignment of a permanent UUID may be done on the server. In these cases it is not practical for a newly created referenceable node to be given a UUID upon creation. Rather, it makes more sense for the UUID to be assigned upon **save** of that node. In such cases a “dummy UUID” may be assigned on creation of a new node while the real UUID assignment takes place later, upon **save**. Applications should not, therefore, rely on the UUID of a node before that node is saved for the first time.

4.9.1.2 Reference Properties

Being referenceable allows a node to be the target of a property of **PropertyType.REFERENCE**. A **REFERENCE** property stores the UUID of an existing node in the same workspace and it enforces referential integrity (see [6.2.5.4 Reference](#)). An exception to the referential integrity rule is made for **REFERENCE** properties stored as part of frozen version state in the version storage (see [8.2.2.9 Reference Properties within Version Storage](#)).

4.10 Workspaces

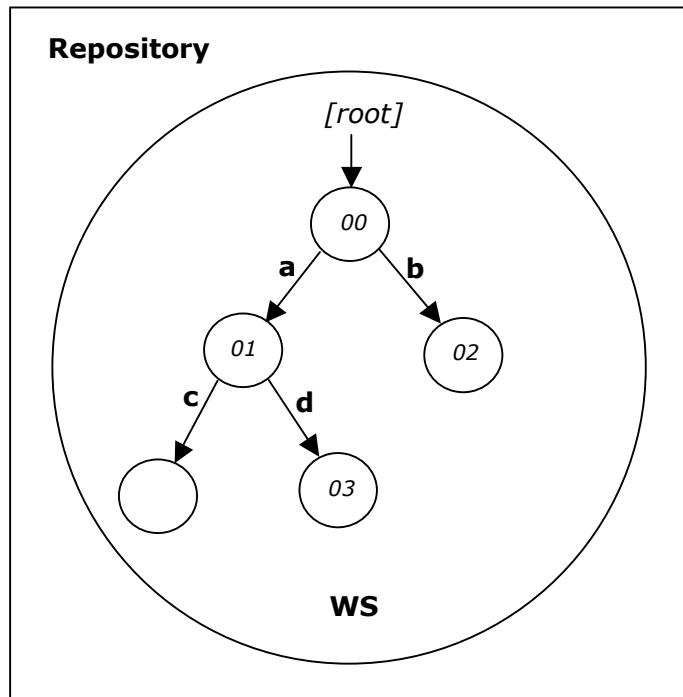
A JCR repository is composed of a number of *workspaces*. Each workspace contains a single rooted tree of items. In the simplest case a repository will consist of just one workspace. In more complex cases a repository will consist of more than one workspace.

4.10.1 Single Workspace Repositories

A repository with only a single workspace consists of a single tree of nodes and properties. The example at the beginning of this section ([4 The Repository Model](#)) essentially describes a single workspace repository.

Since a given workspace contains at most one node with a given UUID, in this case, there is at most one node with a given UUID *in the repository as a whole*.

The following diagram depicts a single workspace repository:



The small circles represent nodes. The arrows point from parent to child and are labeled with the name of the child. The name of the root node is actually the empty string though, for clarity, it is indicated here with the string "[root]". The numbers within the nodes represent the UUIDs of the nodes. For example, the UUID of the root node / is 00 and the UUID of /a/d is 03. The node /a/c is not referenceable and therefore does not have a UUID.

4.10.2 Multiple Workspaces and Corresponding Nodes

In repositories that have multiple workspaces, a node in one workspace may have *corresponding nodes* in other workspaces. A node's corresponding node in another workspace is the node in that workspace that has the same UUID. If the node is not referenceable (has no UUID) then its corresponding node is the non-referenceable node with the same path relative to the nearest ancestor that does have a UUID (or the root node, whichever comes first). Referenceable nodes only correspond to other referenceable node and non-referenceables to other non-referenceables (the relationship is symmetric).

Apart from having the same UUID (or UUID plus path, or absolute path, as the case may be), corresponding nodes need have nothing else in common. They can differ, for example, in terms of their properties and child nodes.

While a node *may* have a corresponding node in another workspace, it is not *required* to.

Note that there is still at most one node with a given UUID per workspace.

The `update` method,

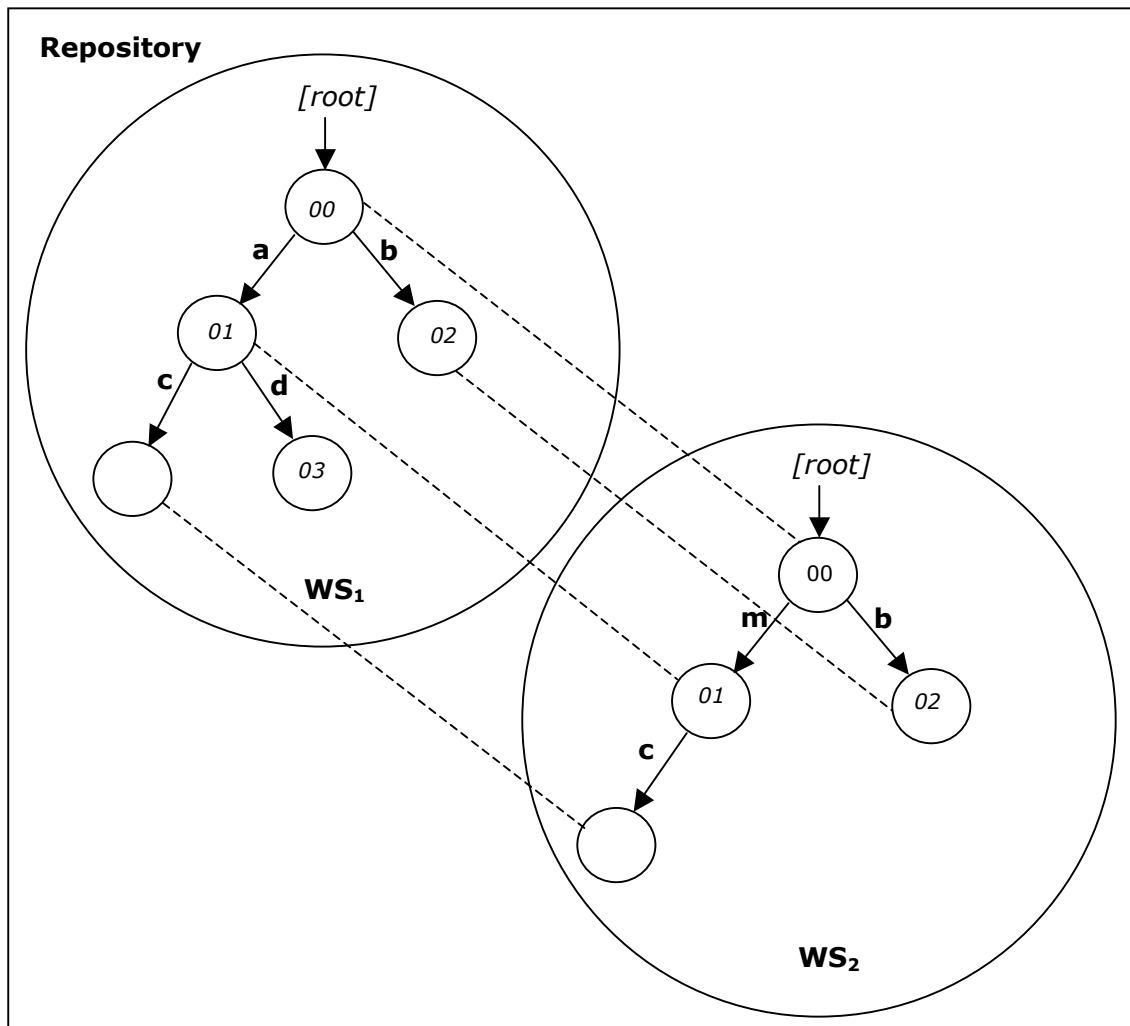
`Node.update(String srcWorkspace)`

causes `this` node and all nodes in its subtree to be updated to reflect the state of their corresponding nodes in `srcWorkspace`.

For more details on corresponding nodes and the update method see [7.1.7](#) *Updating and Cloning Nodes across Workspaces*.

4.10.2.1 Example

The following diagram shows a schematic of a two-workspace repository.



Here we see two workspaces, ws_1 and ws_2 . The dotted lines indicate corresponding nodes. For example, the node $/a$ in ws_1 corresponds to $/m$ in ws_2 because both have a UUID of 01 . Similarly, $/b$ in ws_1 corresponds with $/b$ in ws_2 . In these cases, because the nodes are referenceable, their paths and names are not relevant in determining their correspondence.

On the other hand, $/a/c$ in ws_1 corresponds with $/m/c$ in ws_2 because they have the same relative path (namely, c) from their nearest referenceable corresponding ancestors (namely, $/a$ and $/m$ in ws_1 and ws_2 respectively).

Note there can also be nodes (such as $/a/d$ in ws_1) that exist in one workspace but not in the other.

4.11 Versioning

Support for versioning is an optional feature. The versioning system is built on top of the system of workspaces and referenceable nodes described above.

In a repository that supports versioning, a workspace may contain both *versionable* and *nonversionable* nodes. A node is versionable if and only if it has been assigned the *mixin type* `mix:versionable`, otherwise it is nonversionable. Repositories that do not support versioning will simply not provide this mixin type, whereas repositories that do support versioning must provide it. The type `mix:versionable` is a subtype of `mix:referenceable`, so if a node is versionable it is automatically also referenceable and thus has a UUID.

Being versionable means that at any given time the node's state can be saved for possible future recovery. This saved state is called a *version* and the action of saving it is called *checking in*.

Versions exist as part of a *version history*. Within a version history, the versions form a *version graph* that describes the predecessor/successor relations among versions of a particular versionable node.

Version histories and their contained versions are stored in version storage. There is one version storage per repository, though it is exposed in each workspace as a special protected subtree below the node `/jcr:system/jcr:versionStorage`.

4.11.1.1 Relation Between Nodes and Version Histories

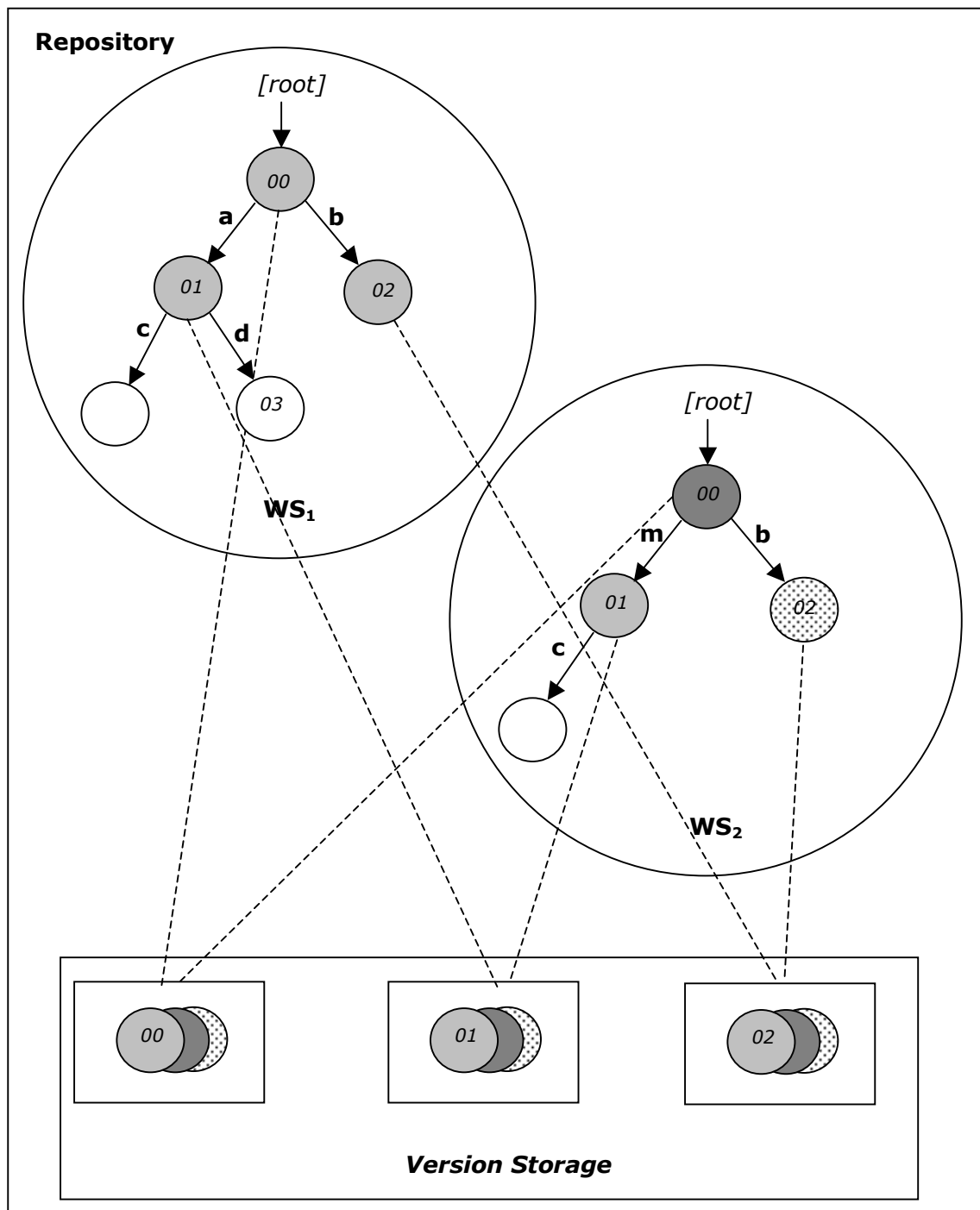
The relationship between nodes and version histories is built on the notion of correspondence via UUID. The details are as follows:

- Each set of corresponding versionable nodes (nodes with the same UUID) share the same version history.
- In a given workspace, there is at most one versionable node per version history (this follows directly from the fact that there is at most one node from each correspondence set per workspace).
- Given a particular workspace, there may be version histories for which that particular workspace does not contain a corresponding versionable node.
- A workspace may contain nonversionable nodes, which, of course, never have corresponding version histories.
- When a new versionable node is created (i.e., the first instance in the repository as whole) a version history for that node is automatically created in version storage.

- If a versionable node is cloned to another workspace, it maintains the same UUID and the new corresponding versionable node remains associated with the original's version history.
- Note that since all versionable nodes are by definition referenceable, there is no need to include the qualification involving relative paths to the nearest versionable node (or root node) as in the discussion of `update`, above.

4.11.1.2 Example

The following diagram illustrates a possible repository architecture.



This diagram shows a repository that supports versioning and contains two workspaces. The version storage is represented by the area in the bottom. It contains a version history for each versionable node in the repository. The versionable nodes in the workspaces are shown in various shadings. The nonversionable nodes are shown in white.

All versionable nodes are referenceable, though not all referenceable nodes are versionable (for example the node *03* in **ws₁** is referenceable, because it has a UUID, but it is not versionable). Both **ws₁** and **ws₂** also contain nonreferenceable nodes (the nodes *c* below *01*).

In the diagram the version histories are represented by stacked circles of differing shades. Each versionable node shares its version history with its corresponding node in the other workspace.

At any given time a particular workspace may hold nodes based on various versions stored in version storage. In the diagram, **ws₁** holds nodes based on the “light gray” version of the nodes *00*, *01* and *02*. **ws₂**, in contrast, has nodes based on the “dark gray” version of *00*, the “light gray” version of *01* and the “dotted” version of *02*.

Note that for the purposes of illustration, each version history is depicted as containing three versions. This is a simplification; in an actual system the version histories of distinct nodes may differ. Furthermore, in this picture, parent child relations within the version storage are not shown. See [8.2 Versioning](#) for a more detailed description.

4.12 Metadata

All content in the repository is ultimately accessed through properties (that is, objects that implement the **Property** interface). The API does not distinguish between “real” content and meta-content.

Such a separation would only duplicate the entire API, since one would probably want to provide the same functionality for handling both meta-content and primary content. The distinction is in any case only meaningful at the level of the application, not the repository. Any particular application built on top of a JCR compliant repository may, of course, choose which content is to be considered “meta”, and which primary.

However, the API does provide the concept of the *primary child item*. Any one of a node’s child items may be specified as its primary child item. This item can be directly accessed (without knowing its name) with the method **Node.getPrimaryItem()**. The designation of a particular node or property as primary is determined by the node type assigned to the parent node.

4.13 Hierarchical versus Direct Access

Though the JCR specification provides a hierarchical, tree-based view of content, it is also compatible with repository implementations that are not primarily hierarchy-based.

Though such implementations must still expose a hierarchical structure, the flexibility of the JCR specification ensures that this need not be a particularly restrictive requirement. The following strategies can be adopted by implementations that are not primarily hierarchical:

- The exposed hierarchy can be almost flat. A very shallow tree consisting of a root node with a large set of child nodes or properties is a valid arrangement. The names of the nodes may be identical with the UUIDs of the nodes.
- There is no requirement that a particular hierarchical view of the repository be in any way “primary”. Through the use of **REFERENCE** properties feature, many orthogonal hierarchical views of the same underlying content are supported. This does away with the notion that there is a single canonical hierarchy. See, for example, [6.7.22.7 nt:linkedFile](#).
- Hierarchical navigation is only one possible way to access the repository. The specification also supports a search query interface (see [6.6 Searching Repository Content](#)). In addition, direct access via node UUID is also provided (see [6.2.1 Session Read Methods](#)).

5 Example Implementations

5.1 JCR on top of a File System

The most straightforward implementation of JCR is as a layer on top of a conventional file system.

Consider, for example, a file system with the following layout:

```
content/
  newpaintings/
    bigredstripe.gif
    bigredstripe.desc
  oldpaintings/
    sistinechapel.gif
    sistinechapel.desc
```

Here is a possible mapping of this file structure to a JCR:

Node	
Property = "..."	
content/	/
newpaintings/	newpaintings
bigredstripe.gif	bigredstripe.gif
(creation date of the file)	jcr:created = "2001-01-01T00:00:00.000Z"
<binary data>	jcr:content
	myapp:data = <binary data>
bigredstripe.desc	bigredstripe.desc
(creation date of the file)	jcr:created = "2001-01-02T00:00:00.000Z"
"An excellent example..."	jcr:content
	myapp:data = "An excellent..."
oldpaintings/	oldpaintings
sistinechapel.gif	sistinechapel.gif
(creation date of the file)	jcr:created = "2001-01-03T00:00:00.000Z"
<binary data>	jcr:content
	myapp:data = <binary data>
sistinechapel.desc	sistinechapel.desc
(creation date of the file)	jcr:created = "2001-01-04T00:00:00.000Z"
"Not bad."	jcr:content
	myapp:data = "Not bad."

In this example, both directories and files are mapped to nodes. Nodes that represent files have a `jcr:created` property and a `jcr:content` node. The `jcr:content` node in turn has a single `myapp:data` property that holds the actual contents of the corresponding file (note that the nodes representing the files `bigredstripe.gif` and so forth, are of node type `nt:file`. See [6.9.22.6 nt:file](#)).

A variation on the above arrangement is to reflect the directory structure directly but combine the file pairs (the picture and the description text) into a single node:

Node/	
Property = "..."	
content/	/
newpaintings/	newpaintings
bigredstripe.gif	bigredstripe
(creation date of .gif or .desc whichever was first)	jcr:created = "2001-05-03T00:00:00.000Z"
<binary data>	jcr:content
	myapp:image = <binary data>

bigredstripe.desc "An excellent example of stripeism."	myapp:desc = "An excellent example of stripeism."
oldpaintings/	oldpaintings
sistinechapel.gif	sistinechapel
(creation date of .gif or .desc whichever was first)	jcr:created = "2001-06-04T00:00:00.000Z"
<binary data>	jcr:content myapp:image = <binary data>
sistinechapel.desc "Not bad."	myapp:desc = "Not bad."

In this example, there is no longer a one-to-one correspondence between file and hierarchy node (though as above, nodes of type `nt:file` are used).

5.2 JCR on top of WebDAV

Consider a repository with some arbitrary underlying implementation structure (file system, database) but which exposes its content via a WebDAV interface.

In such a case, the implementer might choose to implement this specification not directly on top of the underlying system but on top of the existing WebDAV layer.

The mapping from the exposed WebDAV structure to the repository hierarchy can be done quite directly using a mapping similar to that described above in the file system example.

In addition to this, all supplemental information of a WebDAV resource, such as properties, locking information, etc. can be mapped to JCR properties.

/newpaintings/	newpaintings
bigredstripe.gif	bigredstripe.gif
<PROPFIND /newpaintings/bigredstripe.gif>	dav:displayName = "Big Red Stripe" dav:contentLength = 4532 dav:eTag = xyz
...	...
<webdav creation date property>	jcr:created = "2001-05-03 T00:00:00.000Z"
<GET /newpaintings/bigredstripe.gif>	jcr:content myapp:data = <binary data>
bigredstripe.desc	bigredstripe.desc
<PROPFIND /newpaintings/bigredstripe.desc>	dav:displayName = "Big Red Stripe" dav:contentLength = 255 dav:eTag = xyz
...	...
<webdav creation date property>	jcr:created = "2001-05-03 T00:00:00.000Z"
<GET /newpaintings/bigredstripe.desc>	jcr:content myapp:data = "An excellent..."
...	...

(In this example the nodes representing the files are of a subtype of `nt:file`. This can be seen from the fact that they include the subitems `jcr:created` and `jcr:content` specified by `nt:file`, as well as additional WebDAV-specific properties)

5.3 JCR on top of a Database

A JCR-compliant repository can also be implemented on top of a database. Consider again the following repository structure:

```
Node/
Property = "..."/>
/
├─ newpaintings
│   └─ bigredstripe
│       ├── jcr:created = "2001-05-03T00:00:00.000Z"
│       └─ jcr:content
│           ├── myapp:image = <binary data>
│           └─ myapp:desc = "An excellent example
│                           of stripeism."
└─ oldpaintings
    └─ sistinechapel
        ├── jcr:created = "2001-06-04T00:00:00.000Z"
        └─ jcr:content
            ├── myapp:image = <binary data>
            └─ myapp:desc = "Not bad."
```

One possible implementation is to use four tables, a **NODES** table and three **XXXX_PROPERTIES** tables, one for each of the three property types used in the example:

NODES

name	id	parent_id
newpaintings	1	0
bigredstripe	2	1
jcr:content	3	2
oldpaintings	4	0
sistinechapel	5	4
jcr:content	6	5

DATE_PROPERTIES

name	value	parent_id
jcr:created	2001-05-03T00:00:00.000Z	2
jcr:created	2001-06-04T00:00:00.000Z	5

TEXT_PROPERTIES

name	value	parent_id
myapp:desc	An excellent...	3
myapp:desc	Not bad.	6

BLOB_PROPERTIES

name	value	parent_id
myapp:image	<BLOB>	3
myapp:image	<BLOB>	6

5.4 JCR on top of a File System with Structured Content

A slightly more complex implementation of a JCR is as a layer on top of a file system where that file system contains structured content in the form of XML files.

Let's say that the file system looks like this:

```
/
  products.xml
  people.xml
  services.xml
  products/
    rhombus.xml
    ...
  people/
    ...
  services/
    ...
```

And `products.xml` looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<content>
  <title>Our Products</title>
  <lead>Geometrix is proud to offer...</lead>
  <paragraph>
    <text>Geometrix is the industry leader...</text>
    <image>/9j/4AAQSkZJRgABAQ...</image>
  </paragraph>
  <paragraph>
    <text>We have recently...</text>
    <image>/9j/4AAQSkZJRgABAQ...</image>
  </paragraph>
</content>
```

And similarly, `rhombus.xml` looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<content>
  <title>Rhombus: The shape of things to come!</title>
  <price>123.00</price>
  <lead>Here at Geometrix...</lead>
  <paragraph name="_0">
    <text>The rhombus is a very special shape...</text>
    <image>/9j/4AAQSkZJRgABAQ...</image>
  </paragraph>
  <paragraph>
    <text>Some say a square is...</text>
    <image>/9j/4AAQSkZJRgABAQ...</image>
  </paragraph>
</content>
```

One way of mapping this to a JCR would be:

Node		Property = "Some value"
/	[root]	
products.xml	products	
[creation date of the file]	jcr:created = "2001-01-01T..."	
<?xml version="1.0"...?>	jcr:content	
<content>		
<title>Our Products</title>	myapp:title = "Our Products"	
<lead>Geometrixx is...</lead>	myapp:lead = "Geometrixx is..."	
<paragraph>	myapp:paragraph[1]	
<text>Geometrixx is...</text>	myapp:text = "Geometrixx is..."	
<image>/9j/4AAQ...</image>	myapp:image = [binary data]	
</paragraph>		
<paragraph>	myapp:paragraph[2]	
<text>We have...</text>	myapp:text = "We have..."	
<image>/9j/4AAQ...</image>	myapp:image = [binary data]	
</paragraph>		
</content>		
products/		
rhombus.xml	rhombus	
[creation date of the file]	jcr:created = "2002-06-01T..."	
<?xml version="1.0"...?>	jcr:content	
<content>		
<title>Rhombus:...</title>	myapp:title = "Rhombus:..."	
<price>123.00</price>	myapp:price = "123.00"	
<lead>Here at...</lead>	myapp:lead = "Here at..."	
<paragraph>	myapp:paragraph[1]	
<text>The rhombus...</text>	myapp:text = "The rhombus..."	
<image>/9j/4A...</image>	myapp:image = [binary data]	
</paragraph>		
<paragraph>	myapp:paragraph[2]	
<text>Some say...</text>	myapp:text = "Some say..."	
<image>/9j/4A...</image>	myapp:image = [binary data]	
</paragraph>		
</content>		
...	...	
people.xml	people	
[creation date of the file]	jcr:created = "2001-01-01T..."	
<?xml version="1.0"...?>	jcr:content	
<content>		
...	...	
people/		
fred.xml	fred	
[creation date of the file]	jcr:created = "2001-12-01T..."	
<?xml version="1.0"...?>	jcr:content	
...	...	

This example demonstrates the use of a *fine-grained content model* where the mapping to a node-property structure extends from the folder and file level into the internal structure of the XML document.

Note that this example is just one possible mapping; it is not meant to imply that this is the only mapping between JCR and XML (see, for example, XML serialization, for other possible mappings).

5.5 Namespace Prefixes in the Examples

In the above examples, we use colon-delimited *prefixes* for naming certain nodes and properties. The nodes and properties fall into three categories:

- **Nodes and properties common to all applications:** In the example these include `jcr:content` and `jcr:created`. This naming convention is enforced by the node types built into the repository. For example, in example [5.1](#), the node

`bigstripe.gif` is mapped to a node of type `nt:file`, and that node type requires the node to have a property named `jcr:created` and a child node named `jcr:content` (see [6.7 Node Types](#)).

- **Nodes and properties specific to the application:** In the examples we use `myapp`. These names could be enforced by application-specific node types.
- **Nodes and properties with names taken from existing resources:** In the example these include the file and directory names `products` and `rhombus` in example [5.4](#). Since the repository cannot require or predict these names, they cannot be specified in node type definitions:

For more details see [6.3 Namespaces](#).

6 Level 1 Repository Features

The following section explains level 1 of the JCR API on a functional basis. For an explanation organized on an interface-by-interface basis, see the accompanying Javadoc.

Level 1 defines a *read-only* repository. This encompasses the following functionality:

- Retrieval and traversal of nodes and properties
- Reading the values of properties
- Transient namespace remapping
- Export to XML/SAX
- Query facility with XPath syntax
- Discovery of available node types

Where a level 2 repository (or a repository supporting an optional feature, such as versioning) would differ from a purely level 1 repository, the relevant difference is noted.

Since level 2 is a superset of level 1, anything required for level 1 compliance is automatically required for level 2 (see [4.2 Compliance Levels](#)). Thus, this section applies to level 2 implementations as well.

Note that in the discussion below, reference to a “level 1 repository” means a repository that implements *only* level 1 features.

6.1 Accessing the Repository

The point of entry is the **Repository** object, which will typically be registered with a naming service based on the Java Naming and Directory (JNDI) API.

6.1.1 Repository

The naming service lookup (or whatever mechanism is used) will return an object implementing the **Repository** interface.

javax.jcr. Repository	
Session	login(Credentials credentials, String workspaceName) Authenticates the user using the supplied credentials . If workspaceName is recognized as the name of an existing workspace in the repository and

	<p>authorization to access that workspace is granted, then a new <code>Session</code> object is returned. The format of the string <code>workspaceName</code> depends upon the implementation.</p> <p>If <code>credentials</code> is <code>null</code>, it is assumed that authentication is handled by a mechanism external to the repository itself (for example, through the JAAS framework) and that the repository implementation exists within a context (for example, an application server) that allows it to handle authorization of the request for access to the specified workspace. See 8.4 Access Control for more details.</p> <p>If <code>workspaceName</code> is <code>null</code>, a default workspace is automatically selected by the repository implementation. This may, for example, be the “home workspace” of the user whose credentials were passed, though this is entirely up to the configuration and implementation of the repository. Alternatively, this may be a “null workspace” that serves only to provide the method</p> <p><code>Workspace.getAccessibleWorkspaceNames</code>, allowing the client to select from among available “real” workspaces (see 6.2.2 Workspace Read Methods).</p> <p>If authentication or authorization for the specified workspace fails, a <code>LoginException</code> is thrown.</p> <p>If <code>workspaceName</code> is not recognized, a <code>NoSuchWorkspaceException</code> is thrown.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>
<code>Session</code>	<p><code>login(Credentials credentials)</code></p> <p>Equivalent to <code>login(credentials, null)</code>.</p>
<code>Session</code>	<p><code>login(String workspaceName)</code></p> <p>Equivalent to <code>login(null, workspaceName)</code>.</p>
<code>Session</code>	<p><code>login()</code></p> <p>Equivalent to <code>login(null, null)</code>.</p>
<code>String[]</code>	<p><code>getDescriptorKeys()</code></p> <p>Returns a string array holding all descriptor keys available for this implementation. This set must contain at least the built-in keys defined by the string constants in this interface (see below). Used in conjunction with <code>Repository.getDescriptor(String</code></p>

	<code>name</code>) to query information about this repository implementation.
<code>String</code>	<code>getDescriptor(String name)</code> Returns the descriptor for the specified key. Used to query information about this repository implementation. The set of available keys can be found by calling <code>getDescriptorKeys</code> . If the specified key is not found, <code>null</code> is returned.

6.1.1.1 Repository Descriptors

The methods **`Repository.getDescriptorNames`** and **`Repository.getDescriptor`** can be used to query information about the particular repository implementation. The required names are defined as string constants of the **`Repository`** interface. They are:

Descriptor Key (String Constant)	Information Returned
<code>SPEC_VERSION_DESC</code>	The version of the specification that this repository implements.
<code>SPEC_NAME_DESC</code>	The name of the specification that this repository implements.
<code>REP_VENDOR_DESC</code>	The name of the vendor of this repository implementation.
<code>REP_VENDOR_URL_DESC</code>	The URL of the repository vendor.
<code>REP_NAME_DESC</code>	The name of this repository implementation.
<code>REP_VERSION_DESC</code>	The version of this repository implementation.
<code>LEVEL_1_SUPPORTED</code>	The presence of this key indicates that this implementation supports all level 1 features. This key will always be present.
<code>LEVEL_2_SUPPORTED</code>	The presence of this key indicates that this implementation supports all level 2 features.
<code>OPTION_TRANSACTIONS_SUPPORTED</code>	The presence of this key indicates that this implementation supports transactions.
<code>OPTION_VERSIONING_SUPPORTED</code>	The presence of this key indicates that

	this implementation supports versioning.
<code>OPTION_OBSERVATION_SUPPORTED</code>	The presence of this key indicates that this implementation supports observation.
<code>OPTION_LOCKING_SUPPORTED</code>	The presence of this key indicates that this implementation supports locking.
<code>OPTION_QUERY_SQL_SUPPORTED</code>	The presence of this key indicates that this implementation supports the SQL query language.
<code>QUERY_XPATH_POS_INDEX</code>	The presence of this key indicates that the index position notation for same-name siblings is supported within XPath queries. See 6.6.3.1 Adapting XPath for JCR::Same-Name Siblings for more details.
<code>QUERY_XPATH_DOC_ORDER</code>	The presence of this key indicates that XPath queries return results in document order. See 6.6.3.2 Adapting XPath for JCR::Document Order .
<code>QUERY_JCRPATH</code>	The presence of this key indicates that SQL queries can <code>SELECT</code> the pseudo-property <code>jcr:path</code> . See 8.6.3 Database View .
<code>QUERY_JCRSCORE</code>	The presence of this key indicates that the <code>jcr:score</code> pseudo-property is available in XPath and SQL queries that include a <code>jcrfn:contains</code> (in XPath) or <code>CONTAINS</code> (in SQL) function to do a full-text search. See 6.6.4.3 contains function and 8.6.5.5 CONTAINS .

Implementations may add additional descriptors.

6.1.2 Credentials

The credentials that are passed must implement the empty marker interface `Credentials`.

The implementer may either provide its own custom implementation or use the provided `SimpleCredentials` class. This class provides a minimal standard method for authenticating against a repository (i.e., using a user ID and password). Additional attributes may be used by the repository, for example, to set a token that can then be passed back and forth once authentication

has been completed (thus enabling later authorization without re-authentication).

javax.jcr. SimpleCredentials	
String	SimpleCredentials(String userId, char[] password) Create a new SimpleCredentials object, given a user ID and password. Note that the given password is cloned before it is stored in the new SimpleCredentials object. This should avoid the risk of having unnecessary references to password data lying around in memory.
String	getUserId() Gets the user ID.
char[]	getPassword() Returns the password. Note that this method returns a reference to the password. It is the caller's responsibility to zero out the password information after it is no longer needed.
void	setAttribute(String name, Object value) Stores an attribute in this credentials instance.
void	removeAttribute(String name) Removes an attribute from this credentials instance.
Object	getAttribute(String name) Returns the value of the named attribute as an Object , or null if no attribute of the given name exists.
String[]	getAttributeNames() Returns the names of the attributes available to this credentials instance. This method returns an empty array if the credentials instance has no attributes available to it.

6.2 Reading Repository Content

Reading repository content involves accessing nodes (either directly or by traversing the hierarchy step by step) and reading the values of properties.

The **Session** object returned by **Repository.login** encapsulates both the authorization settings of a particular user (as determined by the **Credentials** object) and a binding to the workspace specified by the **workspaceName** passed on **login**.

Each `session` object is associated one-to-one with a `Workspace` object. The `Workspace` object represents a “view” of an actual repository workspace entity as seen through the authorization settings of its associated `session`.

There is an important distinction between the `Workspace` object instance associated with a particular `session` and the actual workspace entity in the repository. If multiple `sessions` access a particular workspace (say the one called “myWorkspace”) each will have its own `Workspace` object, even though all of these `Workspace` objects may represent the same actual workspace entity in the repository. In other words, a `Workspace` object corresponds to a *view* of a particular workspace entity, and that view is determined by the `session` associated with the `Workspace` object.

On the other hand, each `session` object represents a separate session entity. Two or more `session` instances can exist for the same `Credentials` and the same `workspaceName` but still have different states.

Since `session` and `Workspace` instances are always associated one-to-one, combining them into a single object might seem logical. However, the distinction between the two objects comes into play in level 2 implementations, where writing to the repository can occur in two ways, either through transient storage (associated with the `session` object) or directly to the persistent layer (associated with the `Workspace` object). It is primarily to differentiate these two modes of writing that the distinction between the two objects is maintained. See [4.1.3.2 Transient Storage in the Session](#) and [7.1 Writing Repository Content](#) for more details.

In a level 1 repository the distinction between `session` and `Workspace` objects does not play a significant role. It exists simply for the sake of compatibility with level 2.

6.2.1 Session Read Methods

The following are the methods of `session` associated with accessing information about a repository and for accessing content from this `session`'s workspace. `session` has other methods as well. In a level 1-only implementation, these other methods will either do nothing or throw an exception. See and [7.1 Writing Repository Content](#).

The most important methods exposed by `Session` are those that provide access to the Items in the workspace tree: typically the user would begin by calling `session.getRootNode()`, which returns the root node of the workspace. From this the user can traverse the workspace tree. It is also possible to directly access a node in the workspace with `session.getNodeByUUID` or `session.getItem`.

javax.jcr. Session	
Repository	getRepository() Returns the Repository object through which this Session was acquired.
String	getUserId() Gets the user ID associated with this session . How the user ID is set is up to the implementation, it may be a string passed in as part of the credentials or it may be a string acquired in some other way. This method is free to return an "anonymous user id" or null .
String[]	getAttributeNames() Returns the names of the attributes set in this session as a result of the Credentials that were used to acquire it. Not all Credentials implementations will contain attributes (though, for example, SimpleCredentials does allow for them). This method returns an empty array if the Credentials instance did not provide attributes.
Object	getAttribute(String name) Returns the value of the named attribute as an Object , or null if no attribute of the given name exists. See Session.getAttributeNames , above.
Workspace	getWorkspace() Returns the Workspace attached to this Session .
Session	impersonate(Credentials c) Returns a new Session in accordance with the specified (new) Credentials . Allows the current user to "impersonate" another using incomplete or relaxed credentials requirements (perhaps including a user name but no password, for example), assuming that this Session gives them that permission. The new Session is tied to a new Workspace instance. In other words, Workspace instances are not re-used. However, the Workspace instance returned represents the same actual persistent workspace entity in the repository as is represented by the Workspace object tied to this Session . A LoginException is thrown if this session does not

	<p>have sufficient permissions to perform the operation.</p> <p>A RepositoryException is thrown if another error occurs.</p>
<code>void</code>	<p>logout()</p> <p>Releases all resources associated with this session. This method should be called when a session is no longer needed.</p>
<code>Node</code>	<p>getRootNode()</p> <p>Returns the root node of the workspace, <code>/</code>. This node is the main access point to the content of the workspace.</p> <p>A RepositoryException is thrown if an error occurs.</p>
<code>Item</code>	<p>getItem(String absPath)</p> <p>Returns the item at the specified absolute path in the workspace.</p> <p>A PathNotFoundException is thrown if no item at absPath exists.</p> <p>A RepositoryException is thrown if another error occurs.</p>
<code>boolean</code>	<p>itemExists(String absPath)</p> <p>Returns true if an item exists at absPath and this session has read access to it; otherwise returns false. Also returns false if the specified absPath is malformed.</p>
<code>Node</code>	<p>getNodeByUUID(String uuid)</p> <p>Returns the node specified by the given UUID. Only applies to nodes that expose a UUID, in other words, those of mixin node type mix:referenceable.</p> <p>An ItemNotFoundException is thrown if no item with the specified UUID exists.</p> <p>A RepositoryException is thrown if another error occurs.</p>

6.2.2 Workspace Read Methods

In a level 1 repository the **workspace** object serves only to encapsulate a number of methods for accessing either information about the **workspace** or classes that provide further repository functions (see [6.2 Reading Repository Content](#) for an explanation). The following are the level 1 methods of **workspace**. **workspace** has

other methods as well. In a level 1-only implementation these will either do nothing or throw an exception.

javax.jcr. Workspace	
Session	getSession() Returns the session object through which this Workspace object was acquired.
String	getName() Returns the name of the actual persistent workspace represented by this Workspace object. This is the name used in Repository.login .
QueryManager	getQueryManager() Returns the QueryManager , through which search methods are accessed. See 6.6 Searching Repository Content . A RepositoryException is thrown if an error occurs.
NamespaceRegistry	getNamespaceRegistry() Returns the NamespaceRegistry object, which can be used to access the mapping between prefixes and namespaces. See 6.3 Namespaces . A RepositoryException is thrown if an error occurs.
NodeTypeManager	getNodeTypeManager() Returns the NodeTypeManager , which is used to access information about which node types are available in the repository. There is one node type registry per repository, therefore the NodeTypeManager is not workspace-specific; it provides introspection methods for the global, repository-wide set of available node types. See 6.7 Node Types . A RepositoryException is thrown if an error occurs.
String[]	getAccessibleWorkspaceNames() Returns an string array containing the names of all workspaces in this repository that are accessible to this user, given the Credentials that were used to get the session to which this Workspace is tied. In order to access one of the listed workspaces, the user performs another Repository.login , specifying

	<p>the name of the desired workspace, and receives a new <code>Session</code> object.</p> <p>A <code>RepositoryException</code> is thrown if an error occurs.</p>
--	---

6.2.3 Node Read Methods

The `Node` interface includes the following methods for getting the child nodes and properties of a node.

javax.jcr. Node	
Node	<p>getNode(String relPath)</p> <p>Returns the node at <code>relPath</code> relative to this node.</p> <p>If <code>relPath</code> contains a path element that refers to a node with same-name sibling nodes without explicitly including an index using the array-style notation (<code>[x]</code>), then the index <code>[1]</code> is assumed (See 4.3 Same-Name Siblings).</p> <p>Within the scope of a single <code>session</code> object, if a node has been acquired with <code>getNode</code>, any subsequent call of <code>getNode</code> reacquiring the same node must return a <code>Node</code> object reflecting the same state as the earlier <code>Node</code> object. Whether this object is actually the same <code>Node</code> instance, or simply one wrapping the same state, is up to the implementation. See 7.1.2.1 Re-using Item Objects.</p> <p>If no node exists at <code>relPath</code> a <code>PathNotFoundException</code> is thrown.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>
NodeIterator	<p>getNodes()</p> <p>Returns all child nodes of this node. Does <i>not</i> include properties of this node. The same reacquisition semantics apply as with <code>getNode</code>. If this node has no child nodes, then an empty iterator is returned.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>
NodeIterator	<p>getNodes(String namePattern)</p> <p>Gets all child nodes of this node that match <code>namePattern</code>. The pattern may be a full name or a partial name with one or more wildcard characters ("<code>*</code>"), or a disjunction (using the "<code> </code>" character to</p>

	<p>represent logical OR) of these. For example,</p> <pre>N.getNodes("jcr:* myapp:report my doc")</pre> <p>would return a <code>NodeIterator</code> holding all child nodes of <code>N</code> that are either called 'myapp:report', begin with the prefix 'jcr:' or are called 'my doc'.</p> <p>Note that leading and trailing whitespace <i>around</i> a disjunct is ignored, but whitespace <i>within</i> a disjunct forms part of the pattern to be matched.</p> <p>The EBNF for <code>namePattern</code> is:</p> <pre>namePattern ::= disjunct { ' ' disjunct } disjunct ::= '*' '*' fragment fragment '*' fragment { '*' fragment } fragment ::= char { char } char ::= (* Any Unicode character other than '*' or ' ' *)</pre> <p>The pattern is matched against the names (not the paths) of the immediate child nodes of this node.</p> <p>If this node has no matching child nodes, then an empty iterator is returned.</p> <p>The same reacquisition semantics apply as with <code>getNode</code>.</p> <p>A <code>RepositoryException</code> is thrown if an error occurs.</p>
Property	<pre>getProperty(String relPath)</pre> <p>Get the property at <code>relPath</code> relative to this node. The same reacquisition semantics apply as with <code>getNode</code>.</p> <p>If no property exists at <code>relPath</code> a <code>PathNotFoundException</code> is thrown.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>
PropertyIterator	<pre>getProperties()</pre> <p>Gets all properties of this node. Does <i>not</i> include child <i>nodes</i> of this node. The same reacquisition semantics apply as with <code>getNode</code>. If this node has no properties, then an empty iterator is returned.</p> <p>A <code>RepositoryException</code> is thrown if an error occurs.</p>

PropertyIterator	<p>getProperties(String namePattern)</p> <p>Gets all properties of this node that match namePattern. The pattern may be a full name or a partial name with one or more wildcard characters ("*"), or a disjunction (using the " " character to represent logical OR) of these. For example,</p> <pre>N.getProperties("jcr:* myapp:name my doc")</pre> <p>would return a PropertyIterator holding all properties of N that are either called 'myapp:name', begin with the prefix 'jcr:' or are called 'my doc'.</p> <p>Note that leading and trailing whitespace <i>around</i> a disjunct is ignored, but whitespace <i>within</i> a disjunct forms part of the pattern to be matched.</p> <p>The EBNF for namePattern is:</p> <pre>namePattern ::= disjunct {' ' disjunct} disjunct ::= '*' '*' fragment fragment '*' fragment {'*' fragment} fragment ::= char {char} char ::= (* Any Unicode character other than '*' or ' ' *)</pre> <p>The pattern is matched against the names (not the paths) of the immediate child properties of this node.</p> <p>If this node has no matching properties, then an empty iterator is returned.</p> <p>The same reacquisition semantics apply as with getNode.</p> <p>A RepositoryException is thrown if an error occurs.</p>
Item	<p>getPrimaryItem()</p> <p>The primary node type (see 6.9 Node Types) of this node may specify one child item (child node or property) of this node as the <i>primary child item</i>. This method returns that item.</p> <p>The same reacquisition semantics apply as with getNode.</p> <p>If this node has no primary child item then this method throws an ItemNotFoundException.</p> <p>A RepositoryException is thrown if another error</p>

	occurs.
<code>String</code>	<p>getUUID()</p> <p>Returns the UUID of this node as recorded in the node's <code>jcr:uuid</code> property. This method only works on nodes of mixin node type <code>mix:referenceable</code>.</p> <p>On nonreferenceable nodes, this method throws an <code>UnsupportedRepositoryOperationException</code>. To avoid throwing an exception to determine whether a node has a UUID, a call to <code>isNodeType("mix:referenceable")</code> can be made.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>
<code>int</code>	<p>getIndex()</p> <p>This method 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., <code>/a[3]/b[4]</code>. Note that the index always starts at 1 (not 0), for compatibility with XPath. As a result, for nodes that do not have same-name-siblings, this method will always return 1.</p>
<code>PropertyIterator</code>	<p>getReferences()</p> <p>Returns all REFERENCE properties that refer to this node.</p> <p>Note that in level 2 implementations, this method returns only saved properties (in a transactional setting this includes both those properties that have been saved but not yet committed, as well as properties that have been committed). This method does not return REFERENCE properties that have been added but not yet saved.</p> <p>In implementations that support versioning, this method does it return REFERENCE properties that are part of the frozen state of a version in version storage.</p> <p>If this node has no references, an empty iterator is returned.</p> <p>A <code>RepositoryException</code> is thrown if an error occurs.</p>
<code>boolean</code>	<p>hasNode(String relPath)</p> <p>Returns <code>true</code> if a node exists at <code>relPath</code> and <code>false</code></p>

	otherwise.
<code>boolean</code>	hasNodes() Returns <code>true</code> if this node has one or more child nodes. Returns <code>false</code> otherwise.
<code>boolean</code>	hasProperty(String relPath) Returns <code>true</code> if a property exists at <code>relPath</code> and <code>false</code> otherwise.
<code>boolean</code>	hasProperties() Returns <code>true</code> if this node has one or more properties. Returns <code>false</code> otherwise.

6.2.4 Property Read Methods

The read methods of **Property** are:

javax.jcr. Property	
<code>Value</code>	getValue() Returns the value of this property as a generic <code>Value</code> object. If this property is multi-valued, this method throws a <code>ValueFormatException</code> . A <code>RepositoryException</code> is thrown if an error occurs.
<code>Value[]</code>	getValues() Returns an array of all the values of this property. This method is used to access multi-value properties. If the property is single-valued, this method throws a <code>ValueFormatException</code> . The array returned is a copy of the stored values, so changes to it are not reflected in internal storage. A <code>RepositoryException</code> is thrown if an error occurs.
<code>String</code>	getString() Returns a <code>String</code> representation of the value of this property. A shortcut for <code>Property.getValue().getString()</code> . See 6.2.7 Value . If this property is multi-valued, this method throws a

	<p>ValueFormatException.</p> <p>If the value of this property cannot be converted to a String, a ValueFormatException is thrown.</p> <p>A RepositoryException is thrown if another error occurs.</p>
InputStream	<p>getStream()</p> <p>Returns an InputStream representation of the value of this property. A shortcut for Property.getValue().getStream(). See 6.2.7 Value.</p> <p>If this property is multi-valued, this method throws a ValueFormatException.</p> <p>If the value of this property cannot be converted to a stream, a ValueFormatException is thrown.</p> <p>A RepositoryException is thrown if another error occurs.</p>
long	<p>getLong()</p> <p>Returns a long representation of the value of this property. A shortcut for Property.getValue().getLong(). See 6.2.7 Value.</p> <p>If this property is multi-valued, this method throws a ValueFormatException.</p> <p>If the value of this property cannot be converted to a long, a ValueFormatException is thrown.</p> <p>A RepositoryException is thrown if another error occurs.</p>
double	<p>getDouble()</p> <p>Returns a double representation of the value of this property. A shortcut for Property.getValue().getDouble(). See 6.2.7 Value.</p> <p>If this property is multi-valued, this method throws a ValueFormatException.</p> <p>If the value of this property cannot be converted to a double, a ValueFormatException is thrown.</p> <p>A RepositoryException is thrown if another error occurs.</p>

Calendar	<p>getDate()</p> <p>Returns a Calendar representation of the value of this property. A shortcut for Property.getValue().getDate() See 6.2.7 Value.</p> <p>If this property is multi-valued, this method throws a ValueFormatException.</p> <p>If the value of this property cannot be converted to a Calendar, a ValueFormatException is thrown.</p> <p>A RepositoryException is thrown if another error occurs.</p>
boolean	<p>getBoolean()</p> <p>Returns a boolean representation of the value of this property. A shortcut for Property.getValue().getBoolean(). See 6.2.7 Value.</p> <p>If this property is multi-valued, this method throws a ValueFormatException.</p> <p>If the value of this property cannot be converted to a boolean, a ValueFormatException is thrown.</p> <p>A RepositoryException is thrown if another error occurs.</p>
Item	<p>getNode()</p> <p>If this property is of type REFERENCE this method returns the Node to which this property refers.</p> <p>If this property is multi-valued, this method throws a ValueFormatException.</p> <p>If this property cannot be converted to a reference, then a ValueFormatException is thrown.</p> <p>If this property is a REFERENCE property but is currently part of the frozen state of a version in version storage, this method will throw a ValueFormatException.</p> <p>A RepositoryException is thrown if another error occurs.</p>
long	<p>getLength()</p> <p>Returns the length of the value of this property in bytes if the value is a PropertyType.BINARY, otherwise it returns the number of characters needed</p>

	<p>to display the value (for strings this is the string length, for numeric types it is the number of characters needed to display the number).</p> <p>Returns <code>-1</code> if the implementation cannot determine the length of the value.</p> <p>If this property is multi-valued, this method throws a ValueFormatException.</p> <p>A RepositoryException is thrown if another error occurs.</p>
<code>long[]</code>	<p>getLengths()</p> <p>Returns an array holding the lengths of the values of this (multi-value) property in bytes if the values are PropertyType.BINARY, otherwise it returns the number of characters needed to display each value (for strings this is the string length, for numeric types it is the number of characters needed to display the number). The order of the length values corresponds to the order of the values in the property.</p> <p>Returns a <code>-1</code> in the appropriate position if the implementation cannot determine the length of a value.</p> <p>If this property is single-valued, this method throws a ValueFormatException.</p> <p>A RepositoryException is thrown if another error occurs.</p>
<code>int</code>	<p>getType()</p> <p>Returns the type of this Property. The type returned is that which was set at property creation. Note that for some property <code>p</code>, the type returned by <code>p.getType()</code> may differ from the type returned by <code>p.getDefinition().getRequiredType()</code> only in the case where the latter returns UNDEFINED. The type of a property instance is never UNDEFINED (it must always have some actual type). See 6.2.5 Property Types and 6.7.18 Discovery of Constraints on Existing Items.</p>

6.2.5 Property Types

The class **PropertyType** defines integer constants for the available property types as well as for their standardized type names (used in serialization) and two methods for converting back and forth between name and integer value:

javax.jcr. PropertyType	
int	STRING The STRING property type has the same characteristics as the Java <code>String</code> class.
int	BINARY BINARY properties are used to store binary data.
int	LONG The LONG property type has the same characteristics as the Java primitive type <code>long</code> .
int	DOUBLE The DOUBLE property type has the same characteristics as the Java primitive type <code>double</code> .
int	BOOLEAN The BOOLEAN property type has the same characteristics as the Java primitive type <code>boolean</code> .
int	DATE The text format of a DATE property must conform to the ISO 8601:2000 format: <code>YYYY-MM-DDThh:mm:ss.sssTZD</code> . See 6.2.5.1 Date .
int	NAME A NAME is a pairing of a <i>namespace</i> and a <i>local name</i> . It is stored internally in fully qualified form, though when read (or written, in level 2) the namespace is mapped to the current prefix. See 6.2.5.2 Name .
int	PATH A PATH property is an ordered list of <i>path elements</i> . A path element is a NAME plus an optional index. When read (or written, in level 2), the fully qualified NAMES within the path are mapped to their current prefix and the result is returned as a string. A path may be absolute or relative. See 6.2.5.3 Path .

int	<p>REFERENCE</p> <p>A REFERENCE property stores the UUID of a referenceable node (one having type <code>mix:referenceable</code>), which must exist within the same workspace as the REFERENCE property. A REFERENCE property enforces this referential integrity by preventing the removal of its target node. See 6.2.5.4 Reference.</p>
int	<p>UNDEFINED</p> <p>This constant can be used within a property definition (see 6.7.6 Property Definitions) to specify that the property in question may be of any type. However, it cannot be the actual type of any property instance. For example it will never be returned by <code>Property.getType</code> and (in level 2 implementations) it cannot be assigned as the type when creating a new property.</p>
String	<p><code>TYPENAME_STRING == "String"</code></p> <p><code>TYPENAME_BINARY == "Binary"</code></p> <p><code>TYPENAME_LONG == "Long"</code></p> <p><code>TYPENAME_DOUBLE == "Double"</code></p> <p><code>TYPENAME_DATE == "Date"</code></p> <p><code>TYPENAME_BOOLEAN == "Boolean"</code></p> <p><code>TYPENAME_NAME == "Name"</code></p> <p><code>TYPENAME_PATH == "Path"</code></p> <p><code>TYPENAME_REFERENCE == "Reference"</code></p> <p><code>TYPENAME_UNDEFINED == "Undefined"</code></p> <p>These constants define the standard string forms of the property types. These are used, for example, when serializing content to XML. See 6.4 XML Mappings.</p>
String	<p>nameFromValue(int type)</p> <p>Returns the standard name of the given property type, specified by its integer value.</p>
int	<p>valueFromName(String name)</p> <p>Returns the integer value of the given property type, specified by its standard name.</p>

6.2.5.1 Date

The text format of dates must follow the following ISO 8601:2000-compliant format:

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

where:

sYYYY Four-digit year with optional leading positive ('+') or negative ('-') sign. A negative sign indicates a year BCE. The absence of a sign or the presence of a positive sign indicates a year CE (for example, -0055 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)

mm Two digits of minute (00 through 59)

ss.sss Seconds, to three decimal places (00.000 through 59.999)

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. See <http://www.w3.org/TR/NOTE-datetime> for more information.

6.2.5.2 Name

A **NAME** is a pairing of a *namespace* and a *local name*. It is stored internally in fully qualified form, for example:

`{http://mycorp.com/myapp}myItem`

(note that this format is only an example. The actual way in which URI and local name are stored is an implementation issue.)

When read through the API the namespace is mapped to the current prefix, for example. If at the time of reading the current prefix to URI mapping is:

`myapp -> http://mycorp.com/myapp`

then the above value would be returned as the string:

`myapp:myItem`

If the namespace were later remapped to

`yourapp -> http://mycorp.com/myapp`

then the value returned would be the string

yourapp:myItem

NAME properties are used for recording values such as node type names (see [6.7.5 Special Properties *jcr:primaryType* and *jcr:mixinTypes*](#)) that must respect namespace mappings.

Upon **save**, a **NAME** property is validated according to two criteria:

- The prefix specified (if any) must be currently mapped to a registered namespace (see [6.3 Namespaces](#)).
- The syntax of the string specified must conform to the following EBNF:

```
name ::= simplename | prefixedname

simplename ::= onecharsimplename |
              twocharsimplename |
              threeormorecharname

prefixedname ::= prefix ':' localname

localname ::= onecharlocalname |
              twocharlocalname |
              threeormorecharname

onecharsimplename ::= (* Any Unicode character except:
                        '.', '/', ':', '[', ']', '*',
                        '''', '""', '|' or any whitespace
                        character *)

twocharsimplename ::= '.' onecharsimplename |
                      onecharsimplename '.' |
                      onecharsimplename onecharsimplename

onecharlocalname ::= nonspace

twocharlocalname ::= nonspace nonspace

threeormorecharname ::= nonspace string nonspace

prefix ::= (* Any valid XML Name *)

string ::= char | string char

char ::= nonspace | ' '

nonspace ::= (* Any Unicode character except:
              '/', ':', '[', ']', '*',
              '''', '""', '|' or any whitespace
              character *)
```

6.2.5.3 Path

A **PATH** property is an ordered list of *path elements*. A path element is a **NAME** plus an with optional index. When read, the fully qualified **NAMES** within the path are mapped to their current prefix and the

result is returned as a string. A path may be absolute or relative. For example, give the namespace mapping

```
myapp -> http://mycorp.com/myapp
```

a **PATH** property value stored internally as:

```
/ {http://mycorp.com/myapp}document[1]  
/ {http://mycorp.com/myapp}paragraph[3]
```

would, when read, return

```
/myapp:document/myapp:paragraph[3]
```

If the namespace were later remapped to

```
yourapp -> http://mycorp.com/myapp
```

then the value returned would be the string

```
/yourapp:document/yourapp:paragraph[3]
```

A common use for **PATH** properties is likely to be the storage of paths to other items in the workspace. However the repository does not enforce referential integrity (unlike in the case of **REFERENCE** properties, see [6.2.5.4 Reference](#)); a **PATH** property may specify a location where no item exists.

Upon **save**, a **PATH** property is validated according to two criteria:

- All prefixes specified must be currently mapped to registered namespaces (see [6.3 Namespaces](#)).
- The syntax of the string specified must conform to the following EBNF:

```
path ::= properpath ['/' ]  
properpath ::= abspath | relpath  
abspath ::= '/' relpath  
relpath ::= pathelement | relpath '/' pathelement  
pathelement ::= name | name '[' number ']' | '..' | '.'  
number ::= /* An integer > 0 */  
name ::= simplename | prefixedname  
simplename ::= onecharsimplename |  
               twocharsimplename |  
               threeormorecharname  
prefixedname ::= prefix ':' localname  
localname ::= onecharlocalname |  
              twocharlocalname |  
              threeormorecharname
```

```

onecharsimplename ::= (* Any Unicode character except:
                        '.', '/', ':', '[', ']', '*',
                        '''', '""', '|' or any whitespace
                        character *)

twocharsimplename ::= '.' onecharsimplename |
                      onecharsimplename '.' |
                      onecharsimplename onecharsimplename

onecharlocalname ::= nonspace

twocharlocalname ::= nonspace nonspace

threeormorecharname ::= nonspace string nonspace

prefix ::= (* Any valid XML Name *)

string ::= char | string char

char ::= nonspace | ' '

nonspace ::= (* Any Unicode character except:
              '/', ':', '[', ']', '*',
              '''', '""', '|' or any whitespace
              character *)

```

Note that the method `Property.getNode()` which resolves a **REFERENCE** property and returns the referenced node *does not work* with **PATH** properties (see [6.2.5.4 Reference](#)). **PATH** properties may point to other properties (not just referenceable nodes) or to nothing at all. In order to use a **PATH** to retrieve an item, the **PATH**'s value must be retrieved and then used in a regular `getNode` or `getProperty` call.

6.2.5.4 Reference

A **REFERENCE** property stores the UUID of a referenceable node (one having type `mix:referenceable`), which must exist within the same workspace as the **REFERENCE** property. A **REFERENCE** property enforces this referential integrity by preventing the removal of its target node.

When a **REFERENCE** property is created or set, an immediate (that is, not on **save**) check is done to guarantee that the target of the reference exists within the workspace. If it does not exist an exception is thrown.

When a **remove** is done on a referenceable node, a check is done on when an attempt is made to **save** the removal to ensure that no reference properties in the workspace still refer to this node. If any do, a `ConstraintViolationException` is thrown.

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 UUID of a node that is no longer in the workspace (see [8.2.2.9 Reference Properties within a Version](#)).

6.2.6 Property Type Conversion

When a read or write of a property is performed with an access method or value of a different type than the property, an attempt will be made to automatically convert between types using the principles described in the following table:

From	To								
	String	Binary	Date	Double	Long	Boolean	Name	Path	Reference
String		UTF-8	ISO 8601:2000. Throw on format error.	java.lang. Double. valueOf(String) (base 10 conversion)	java.lang. Long. valueOf(String) (base 10 conversion)	java.lang. Boolean. valueOf(String)	Throw on format error.	Throw on format error.	Throw on format error.
Binary	UTF-8. If binary is not UTF-8 behavior is implementation-specific		via String	via String	via String	via String	via String	via String	via String
Date	ISO 8601:2000	via String		Milliseconds since 1970-01-01T00:00:00Z	Milliseconds since 1970-01-01T00:00:00Z	Throws ValueFormat Exception	Throw ValueFormat Exception	Throw ValueFormat Exception	Throw ValueFormat Exception
Double	java.lang. Double. toString() (base 10 conversion)	via String	Milliseconds since 1970-01-01T00:00:00Z		Standard Java conversion	Throw ValueFormat Exception	Throw ValueFormat Exception	Throw ValueFormat Exception	Throw ValueFormat Exception
Long	java.lang. Long.toString() (base 10 conversion)	via String	Milliseconds since 1970-01-01T00:00:00Z	Standard Java conversion		Throw ValueFormat Exception	Throw ValueFormat Exception	Throw ValueFormat Exception	Throw ValueFormat Exception
Boolean	java.lang. Boolean. toString()	via String	Throw ValueFormat Exception	Throw ValueFormat Exception	Throw ValueFormat Exception		Throw ValueFormat Exception	Throw ValueFormat Exception	Throw ValueFormat Exception
Name	Direct	via String	Throw ValueFormat Exception	Throw ValueFormat Exception	Throw ValueFormat Exception	Throw ValueFormat Exception		Name becomes relative path	Throw ValueFormat Exception
Path	Direct	via String	Throw ValueFormat Exception	Throw ValueFormat Exception	Throw ValueFormat Exception	Throw ValueFormat Exception	If Path is relative and one element long and has no index, convert directly, otherwise throw ValueFormat Exception		Throw ValueFormat Exception
Reference	Direct	via String	Throw ValueFormat Exception	Throw ValueFormat Exception	Throw ValueFormat Exception	Throw ValueFormat Exception	Throw ValueFormat Exception	Throw ValueFormat Exception	

6.2.7 Value

The `Value` interface provides a type-neutral wrapper for a property's content, allowing it to be passed to other methods without knowledge of the underlying type of the property. The methods of the `Value` interface are:

javax.jcr. Value	
string	getString() Returns a string representation of this value. If this value cannot be converted to a string , a ValueFormatException is thrown.

	<p>If <code>getStream</code> has previously been called on this <code>Value</code> instance, an <code>IllegalStateException</code> is thrown. In this case, a new <code>Value</code> instance must be acquired in order to successfully call <code>getString</code>.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>
<code>InputStream</code>	<p><code>getStream()</code></p> <p>Returns an <code>InputStream</code> representation of this value. Uses the standard conversion to binary.</p> <p>If this value cannot be converted to an <code>InputStream</code>, a <code>ValueFormatException</code> is thrown.</p> <p>If a non-stream <code>get</code> method has previously been called on this <code>Value</code> instance, an <code>IllegalStateException</code> is thrown. In this case, a new <code>Value</code> instance must be acquired in order to successfully call <code>getStream</code>.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>
<code>long</code>	<p><code>getLong()</code></p> <p>Returns a <code>long</code> representation of this value.</p> <p>If this value cannot be converted to a <code>long</code>, a <code>ValueFormatException</code> is thrown.</p> <p>If <code>getStream</code> has previously been called on this <code>Value</code> instance, an <code>IllegalStateException</code> is thrown. In this case a new <code>Value</code> instance must be acquired in order to successfully call <code>getLong</code>.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>
<code>double</code>	<p><code>getDouble()</code></p> <p>Returns a <code>double</code> representation of this value.</p> <p>If this value cannot be converted to a <code>double</code>, a <code>ValueFormatException</code> is thrown.</p> <p>If <code>getStream</code> has previously been called on this <code>Value</code> instance, an <code>IllegalStateException</code> is thrown. In this case a new <code>Value</code> instance must be acquired in order to successfully call <code>getDouble</code>.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>

Calendar	<p>getDate()</p> <p>Returns a Calendar representation of this value.</p> <p>If this value cannot be converted to a Calendar, a ValueFormatException is thrown.</p> <p>If getStream has previously been called on this Value instance, an IllegalStateException is thrown. In this case a new value instance must be acquired in order to successfully call getDate.</p> <p>A RepositoryException is thrown if another error occurs.</p>
boolean	<p>getBoolean()</p> <p>Returns a boolean representation of this value.</p> <p>If this value cannot be converted to a boolean, a ValueFormatException is thrown.</p> <p>If getStream has previously been called on this Value instance, an IllegalStateException is thrown. In this case a new value instance must be acquired in order to successfully call getBoolean.</p> <p>A RepositoryException is thrown if another error occurs.</p>
int	<p>getType()</p> <p>Returns the type of this value. See 6.2.5 Property Types. The type returned is that which was set at property creation.</p>

The API includes a set of classes implementing the **value** interface. **BaseValue** is an abstract class that serves as the superclass of the type-specific **value** implementations, which are:

- **StringValue**
- **LongValue**
- **DoubleValue**
- **BooleanValue**
- **DateValue**
- **BinaryValue**
- **ReferenceValue**
- **NameValue**

- `PathValue`

A repository implementation may use these classes, or may implement its own `Value` classes. If the repository uses its own classes, they must match the behavior of the JCR-supplied classes. In particular, they must observe the following restrictions:

- A `Value` object returned by `Property.getValue()` can be read using type-specific `get` methods. These methods are divided into two groups:
 - The non-stream `get` methods `getString()`, `getDate()`, `getLong()`, `getDouble()` and `getBoolean()`.
 - `getStream()`.
- 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()`.
- Once a `Value` object has been read once using `getStream()`, any subsequent call to any of the non-stream `get` methods will throw an `IllegalStateException`. In order to successfully invoke a non-stream `get` method the `Value` must be reacquired via `Property.getValue()`.
- Once a `Value` object has been read once using a non-stream `get` method, any subsequent call to `getStream()` will throw an `IllegalStateException`. In order to successfully invoke `getStream()` the `Value` must be reacquired via `Property.getValue()`.

`BaseValue` and its subclasses obey these restrictions.

6.2.7.1 Equality Conditions

Two `Value` instances, `v1` and `v2`, are considered equal if and only if

```
v1.getType() == v2.getType() and
v1.getString().equals(v2.getString()).
```

Actually comparing two `Value` instances by converting them to string form may not be practical in some cases (for example, if the values are very large binaries). Consequently, the above is intended as a normative definition of `Value` equality, but not as a procedural test. It is assumed that implementations will have an efficient means of determining equality that conforms with the above definition.

6.2.7.2 Value Length

Determining the length of a `Value` can be done through the `Property` interface by calling `Property.getLength` or `getLengths` (the former for single value properties, the latter for multi-value properties). These length-reporting methods are found on `Property` and not on `Value` because determining the length of a value is typically more useful if done *before* loading the value into local memory as a `Value` object (of course, whether to do such late-loading is an implementation-level issue, but it is likely to be a common approach). As well, in many implementations determining the length of some values may require access to the `Workspace` object and in many cases `Value` objects will not hold such reference (whereas `Property` objects will). See [6.2.4 Property Read Methods](#).

6.2.8 Item Read Methods

The `Item` interface also contains a number of other methods, inherited by both `Node` and `Property`. The following methods provide access to and information about nodes and properties (`Item` also has other methods applicable to level 2).

javax.jcr. Item	
String	getPath() Returns the absolute path to this item. If the path includes items that are same name sibling nodes or multi-value properties then those elements in the path will include the appropriate "square bracket" index notation (for example, <code>/a/b[3]/c</code>). A <code>RepositoryException</code> is thrown if an error occurs.
String	getName() Returns the name of this item. The name is the last item in the path, minus any square-bracket index that may exist. If this item is the root node of the workspace (i.e., if <code>this.getDepth() == 0</code>), an empty string will be returned. A <code>RepositoryException</code> is thrown if an error occurs.
Item	getAncestor(int depth) Returns the ancestor of the specified depth below the root. An ancestor of depth <code>x</code> is the <code>Item</code> that is <code>x</code> levels down along the path from the root node to <i>this Item</i> . <ul style="list-style-type: none">• <code>depth = 0</code> returns the root node.• <code>depth = 1</code> returns the child of the root node

	<p>along the path to <i>this Item</i>.</p> <ul style="list-style-type: none"> • depth = 2 returns the grandchild of the root node along the path to <i>this Item</i>. • And so on to depth = <i>n</i>, where <i>n</i> is the depth of <i>this Item</i>, which returns <i>this Item</i> itself. If depth > <i>n</i> is specified then an ItemNotFoundException is thrown. <p>An ItemNotFoundException will be thrown if depth < 0 or depth > <i>n</i> where <i>n</i> is the is the depth of this item along the path returned by getPath().</p> <p>An AccessDeniedException is thrown if the current session does not have sufficient access permissions to retrieve the specified node.</p> <p>A RepositoryException is thrown if another error occurs.</p>
Node	<p>getParent()</p> <p>Returns the parent of this <i>Item</i>.</p> <p>An ItemNotFoundException is thrown if there is no parent node. This only happens if this item is the root node of a workspace.</p> <p>An AccessDeniedException is thrown if the current session does not have sufficient access permissions to retrieve the parent of this item.</p> <p>A RepositoryException is thrown if another error occurs.</p>
int	<p>getDepth()</p> <p>Returns the depth below the root node of <i>this Item</i> (counting <i>this Item</i> itself):</p> <ul style="list-style-type: none"> • The root node returns 0. • A property or child node of the root node returns 1. • A property or child node of a child node of the root returns 2. • And so on to <i>this Item</i>. <p>A RepositoryException is thrown if an error occurs.</p>
Session	<p>getSession()</p> <p>Returns the session through which this <i>Item</i> was</p>

	<p>acquired.</p> <p>A RepositoryException is thrown if an error occurs.</p>
boolean	<p>isNode()</p> <p>Returns true if this Item is a Node; returns false if this Item is a Property.</p>
boolean	<p>isSame(Item otherItem)</p> <p>Returns true if this Item object (the Java object instance) represents the same actual repository item as the object otherItem.</p> <p>This method does not compare the <i>states</i> 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 when comparing these two objects. Note that if two Item objects representing the same repository item are retrieved through the <i>same</i> session they will always reflect the same state (see 7.1.2 Reflecting Item State) so comparing state is not an issue.</p>
void	<p>accept(ItemVisitor visitor)</p> <p>Accepts an ItemVisitor and calls the appropriate visit method according to whether <i>this</i> Item is a Node or a Property.</p> <p>This method provides support for the <i>visitor</i> design pattern. It takes an ItemVisitor object that must implement two methods: visit(Node node) and visit(Property property). Depending on whether this Item is a Node or a Property one of the visit methods is called with this Item as the parameter.</p> <p>The API also provides the abstract class TraversingItemVisitor implementing ItemVisitor, which automatically traverses the hierarchy calling accept at each node and property. It provides the methods entering and leaving that can be overridden in a subclass to perform custom operations.</p> <p>Throws a RepositoryException if an error occurs.</p>

6.2.9 Example

The following section gives some examples of how node access and property read operations are performed. In order to provide some

context for the examples, we return to our earlier example of a repository structured like this:

```
Node
Property = "Some Value"

[root]
└─ products
    ├── jcr:created = "2001-01-01T..."
    └─ jcr:content
        ├── myapp:title = "Our Products..."
        ├── myapp:lead = "Geometrix is proud..."
        ├── myapp:paragraph[1]
        │   ├── myapp:text = "Geometrix is..."
        │   └─ myapp:image = [binary data]
        └─ myapp:paragraph[2]
            ├── myapp:text = "We have..."
            └─ myapp:image = [binary data]
└─ rhombus
    ├── jcr:created = "2002-06-01T"
    └─ jcr:content
        ├── myapp:title = "Rhombus:..."
        ├── myapp:price = "123.00"
        ├── myapp:lead = "Here at..."
        ├── myapp:paragraph[1]
        │   ├── myapp:text = "The rhombus..."
        │   └─ myapp:image = [binary data]
        └─ myapp:paragraph[2]
            ├── myapp:text = "Some say..."
            └─ myapp:image = [binary data]
```

Now, assume that the programmer has called:

```
Session session = ...
Node root = session.getRootNode();
```

From the root node, one can access any node or property in the workspace. For example,

```
Node n1 = root.getNode("products");
Node n2 = n1.getNode("rhombus");
Node n3 = n2.getNode("jcr:content");
Node n4 = n3.getNode("myapp:paragraph[2]");
Property p = n4.getProperty("myapp:text");
Value v = p.getValue();
String s = v.getString();
System.out.println(s);
```

would print, "Some say..." to standard output. Alternatively, more convenient direct access is also possible,

```
Property p = root.getProperty("products/rhombus/
                               jcr:content/myapp:paragraph[2]/myapp:text");
System.out.println(p.getString());
```

Here we use a relative path from the root to access a property deep in the hierarchy.

As well, traversal of the hierarchy is easily done. For example, given the following method,

```
public void traverse(Node n, int level)
    throws ValueFormatException {
    String name = (n.getDepth() == 0) ? "/" : n.getName();
    System.out.println(makeIndent(level) + name);
    for (PropertyIterator i = n.getProperties();
        i.hasNext();) {
        Property p = i.nextProperty();
        System.out.println(makeIndent(level + 1) +
            p.getName() + " = \"" +
            p.getString() + "\"");
    }
    for (NodeIterator i = n.getNodes(); i.hasNext();) {
        Node nn = i.nextNode();
        traverse(nn, level + 1);
    }
}
```

the call,

```
traverse(root, 0);
```

would print out something like the following:

```
/
products
  jcr:created = "2001-01-01T..."
  jcr:content
    myapp:title = "Our Products..."
    myapp:lead = "Geometrix is proud..."
    myapp:paragraph[1]
      myapp:text = "Geometrix is..."
      myapp:image = ""
    myapp:paragraph[2]
      myapp:text = "We have..."
      myapp:image = ""
rhombus
  jcr:created = "2002-06-01T..."
  jcr:content
    myapp:title = "Rhombus:..."
    myapp:price = "123.00"
    myapp:lead = "Here at..."
    myapp:paragraph[1]
      myapp:text = "The rhombus..."
      myapp:image = ""
    myapp:paragraph[2]
      myapp:text = "Some say..."
      myapp:image = ""
```

6.3 Namespaces

JCR provides support for the namespacing of item and node type names. Namespacing serves to prevent naming collisions among items and node types that come from different sources or application domains. The JCR namespace system is modelled after *XML Namespaces*.

6.3.1 Namespace Registry

Each JCR repository has a single, persistent namespace registry represented by the `NamespaceRegistry` object, accessed via `Workspace.getNamespaceRegistry()`. The following describes the methods of `NamespaceRegistry` supported in level 1.

`NamespaceRegistry` also has other methods that are supported in level 2. In level 1 repositories these methods either do nothing or throw an exception. See [7.2 Adding and Deleting Namespaces](#) for more details.

javax.jcr. NamespaceRegistry	
String[]	getPrefixes() Returns an array holding all currently registered prefixes. A <code>RepositoryException</code> is thrown if an error occurs.
String[]	getURIs() Returns an array holding all currently registered URIs. A <code>RepositoryException</code> is thrown if an error occurs.
String	getURI(String prefix) Returns the URI to which the given <code>prefix</code> is mapped. If a mapping with the specified <code>prefix</code> does not exist, a <code>NamespaceException</code> is thrown. A <code>RepositoryException</code> is thrown if another error occurs.
String	getPrefix(String uri) Returns the prefix which is mapped to the given <code>uri</code> . If a mapping with the specified <code>uri</code> does not exist, a <code>NamespaceException</code> is thrown. A <code>RepositoryException</code> is thrown if another error occurs.

A registered prefix can be used in the name of any node or property in the repository. The prefix serves as shorthand for the URI to which it is mapped. Because the space of URIs is universally managed, the combination of the per-repository namespace and the larger URI namespace can be used to provide universal uniqueness of node or property names. Of course, just as in the case of XML namespaces, ensuring this universal uniqueness requires applications to map their application-specific prefixes to URIs that are uniquely identified with that particular application.

The namespace registry always contains at least the following built-in mappings:

- **jcr** -> <http://www.jcp.org/jcr/1.0>
Reserved for items defined within built-in node types. For example **jcr:content**.
- **nt** -> <http://www.jcp.org/jcr/nt/1.0>
Reserved for the names of built-in primary node types.
- **mix** -> <http://www.jcp.org/jcr/mix/1.0>
Reserved for the names of built-in mixin node types.
- **sv** -> <http://www.jcp.org/jcr/sv/1.0>
Reserved for use in the system view XML serialization format.
- **""** (the empty prefix) -> **""**(the empty URI)
This makes the *default namespace* the *empty URI*. In effect this means that a name without a prefix is identical in both its prefixed form and in its fully qualified form (i.e. when it is stored internally as *URI plus local name*). See [6.6.1 Internal Storage of Names and Values](#).

In a level 1 repository there is no provision for adding new namespaces to the registry (or deleting namespaces from it), this functionality is part of level 2 (see [7.2 Adding and Deleting Namespaces](#)). However, a level 1 implementation may provide any number of built-in namespaces, in addition to the five required ones listed above. As well, level 1 supports the temporary assignment of new prefixes to existing namespaces within the scope of a particular **session** (see immediately below).

6.3.2 Prefix Syntax

A JCR prefix can be any valid XML name. Note that the local name for a JCR item (the part after the colon) might not be a valid XML name (the space of JCR local names is a superset of the space of XML names), however the set of possible JCR prefixes is identical to with the set of possible XML prefixes.

6.3.3 Session Namespace Remapping

Any registered namespace can be temporarily remapped to a new prefix within the scope of a particular **session**.

javax.jcr. Session	
void	setNamespacePrefix(String prefix, String uri) Within the scope of this session , remaps a persistently registered namespace URI to the new prefix . The remapping only affects operations done through this session. To clear all remappings, the client must acquire a new Session .

	<p>Note that a prefix that is currently mapped in the global namespace registry to some URI cannot be remapped to a new URI using this method, since this would make any content stored using the old URI unreadable. An attempt to do this will throw a NamespaceException.</p> <p>A NamespaceException will be thrown if an attempt is made to re-map a URI to a prefix beginning with "xml". These prefixes are reserved by the XML specification.</p> <p>A NamespaceException will also be thrown if the specified uri is not among those registered in the NamespaceRegistry.</p> <p>A RepositoryException is thrown if another error occurs.</p>
String[]	<p>getNamespacePrefixes()</p> <p>Returns all prefixes currently set for this session. This includes all those registered in the NamespaceRegistry but <i>not temporarily over-ridden</i> by a Session.setPrefix, plus those currently set locally by Session.setPrefix.</p> <p>A RepositoryException is thrown if an error occurs.</p>
String	<p>getNamespaceURI(String prefix)</p> <p>Returns the URI to which the given prefix is mapped as currently set in this session.</p> <p>A NamespaceException is thrown if the specified prefix is unknown.</p> <p>A RepositoryException is thrown if another error occurs.</p>
String	<p>getNamespacePrefix(String uri)</p> <p>Returns the prefix to which the given URI is mapped as currently set in this session.</p> <p>A NamespaceException is thrown if the specified uri is unknown.</p> <p>A RepositoryException is thrown if another error occurs.</p>

6.3.3.1 Using Session Namespace Remapping

One use case for session-based namespace remapping occurs in the context of an XPath query (see [6.6 Searching Repository Content](#)).

XPath queries often include literal names that have namespace prefixes. When attempting to use a stored XPath query (or one obtained from some external source) whose prefixes do not match those currently used in the repository, dynamic remapping of

namespaces in the session allows the temporary session mapping to be adapted to whatever prefixes are used in the XPath statement.

6.3.3.2 Scope of Session Namespace Remapping

All methods that take paths or names as arguments use the current `session` namespace mappings to interpret those paths and names. This includes not just methods of `session`, `Item`, `Node` and `Property` but also methods of the `Workspace` object. Since each `Workspace` object is associated one-to-one with a particular `session` object, the object has access to the namespace mapping currently in effect on that `session`.

6.3.4 Internal Storage of Names and Paths

Note that the names of nodes and properties should be stored internally using their fully qualified names, not their prefixed names. For example, the implementation should not store the literal property name

`myapp:myProperty`

but rather (something like),

`{http://mycorp.com/myapp}myProperty`

The syntax of the internal name is, of course, implementation specific; this is just an example.

The fully-qualified name, however, should never be directly exposed through the API (though of course a client may re-construct it by finding the URI to which a particular prefix is mapped).

Similarly, all properties of type `NAME` or `PATH` should store their values in fully-qualified form. All accesses to the values of these properties should shield the client from the raw fully-qualified name and translate the value using the currently mapped prefix (see [6.2.5.2 Name](#) and [6.2.5.3 Path](#)).

6.4 XML Mappings

Level 1 of JCR supports two mappings of the JCR data model to XML. The mappings are called the *system view* and the *document view*.

6.4.1 System View XML Mapping

The system view mapping provides a complete serialization of workspace content to XML without loss of information. In level 1 this allows the complete content of a workspace to be exported (see [6.5 Exporting Repository Content](#)). In level 2, this also allows

for roundtripping of content to XML and back again through export and import (see [7.3 Importing Repository Content](#)).

Given a subtree of a JCR workspace, the resulting system view is determined as follows:

1. Each JCR node becomes an XML element `<sv:node>`.
2. Each JCR property becomes an XML element `<sv:property>`.
3. 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.
4. If the root node of a workspace is included in the serialized subtree, it receives the special name `jcr:root` (instead of the empty string).
5. The property type of each JCR 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` (i.e., "String", "Binary", "Date", "Boolean", "Double", "Long", "Name", "Path" and "Reference").
6. The value of each JCR property is converted to string form (according to [6.2.6 Property Type Conversion](#)) and included as text within an `<sv:value>` element within the `<sv:property>` element. Entity references are used to escape characters (such as `&` and `<`) which should not be included as literals within XML text (see [6.4.4 Escaping of JCR Values](#)).
7. A multi-value property is converted to an `<sv:property>` element containing multiple `<sv:value>` elements.
8. The hierarchy of the JCR nodes and properties is reflected in the hierarchy of the corresponding XML elements.
9. The namespace mapping from the repository `NamespaceRegistry` is included at the beginning of each XML view in the form of XML namespace declarations.
10. The first two `<sv:property>` elements within an `<sv:node>` elements must be the `jcr:primaryType` and `jcr:mixinTypes` (in that order) properties of the node in question.
11. In the case of referenceable nodes, the third `<sv:property>` element in the `<sv:node>` element must be `jcr:uuid`.

6.4.1.1 Example

A subtree with the following structure:

Node

Property = "value"

myapp:document

- └ jcr:primaryType = "nt:unstructured"
- └ myapp:title = "JSR 170"
- └ myapp:lead = "Content Repository"
- └ myapp:body
 - └ jcr:primaryType = "nt:unstructured"
 - └ myapp:paragraph
 - └ jcr:primaryType = "nt:unstructured"
 - └ myapp:title = "Node Types"
 - └ myapp:text = "An important feature..."

where the source repository's namespace registry holds the mappings (in addition to the built-in ones):

myapp -> <http://mycorp.com/myapp>

would appear in the system view as:

```
<?xml version="1.0" encoding="UTF-8"?>
<sv:node xmlns:jcr="http://www.jcp.org/jcr/1.0"
  xmlns:nt="http://www.jcp.org/jcr/nt/1.0"
  xmlns:mix="http://www.jcp.org/jcr/mix/1.0"
  xmlns:sv="http://www.jcp.org/jcr/sv/1.0"
  xmlns:myapp="http://mycorp.com/myapp"
  sv:name="myapp:document">
  <sv:property sv:name="jcr:primaryType" sv:type="Name">
    <sv:value>nt:unstructured</sv:value>
  </sv:property>
  <sv:property sv:name="myapp:title" sv:type="String">
    <sv:value>JSR 170</sv:value>
  </sv:property>
  <sv:property sv:name="myapp:lead" sv:type="String">
    <sv:value>Content Repository</sv:value>
  </sv:property>
  <sv:node sv:name="myapp:body">
    <sv:property sv:name="jcr:primaryType" sv:type="String">
      <sv:value>nt:unstructured</sv:value>
    </sv:property>
    <sv:node sv:name="myapp:paragraph">
      <sv:property sv:name="jcr:primaryType" sv:type="String">
        <sv:value>nt:unstructured</sv:value>
      </sv:property>
      <sv:property sv:name="myapp:title" sv:type="String">
        <sv:value>Node Types</sv:value>
      </sv:property>
      <sv:property sv:name="myapp:text" sv:type="String">
        <sv:value>An important feature...<sv:value>
      </sv:property>
    </sv:node>
  </sv:node>
</sv:node>
```

Note that in the above, the XML has been formatted for readability.

The actual XML stream would not have any extraneous whitespace between elements or attributes.

6.4.2 Document View XML Mapping

The document view is designed to be more human-readable than the system view, though it achieves this at the expense of completeness.

In level 1 the document view is used as the format for the virtual XML stream against which an XPath query is run (see [6.6 Searching Repository Content](#)). As well, in level 1, export to document view format is supported (see [6.5 Exporting Repository Content](#)). In level 2, document view also allows for the import of arbitrary XML (see [7.3.2 Import from Document View](#)).

The document view mapping in fact consists of a family of related mappings whose precise features vary according to the context in which the mapping is employed (export, import or XPath query) and which optional features are supported by the particular implementation in question.

We begin by describing the general structure of the document view mapping, and then move on to explain the special cases, context-related differences and optional features. With respect to context-related differences, the description below addresses XPath and export. A discussion of document view in the context of import can be found in the above-mentioned section [7.3.2 Import from Document View](#).

6.4.2.1 General Structure

Given a subtree of a JCR workspace, the general form of the document view is determined as follows:

1. The content of the repository namespace registry is rendered as a set of namespace declarations in the top-most element of the XML document.
2. Each JCR node \mathfrak{n} becomes an XML element of the same name, \mathfrak{n} .
3. Each child JCR node \mathfrak{c} of \mathfrak{n} becomes a subelement \mathfrak{c} of XML element \mathfrak{n} .
4. Each JCR property \mathfrak{p} of JCR node \mathfrak{n} becomes an XML attribute \mathfrak{p} of XML element \mathfrak{n} .
5. The value of each JCR property \mathfrak{p} is converted to string form according to the standard conversion (see [6.2.6 Property Type Conversion](#)) and becomes the value of the XML attribute \mathfrak{p} . The attribute value is enclosed in double quotes (e.g., $\mathfrak{p}=\text{"v"}$). Entity references are used to escape

characters (such as & and <) which should not be included as literals within attribute values (see [6.4.4 Escaping of JCR Values](#)).

6. The following sections describe the exceptions to the above general rules.

6.4.2.2 Workspace Root

If the root node of a workspace is included within the scope of the serialization, then that node is mapped to an XML element with the name `jcr:root`. This convention is required because XML elements cannot have empty-string names, whereas a JCR root node, by definition, has the empty string as its name.

6.4.2.3 XML Text

If a child node of `N` called `jcr:xmltext` is encountered that has one (and only one) property and that property is a single-value string property called `jcr:xmlcharacters`, then the treatment of that `jcr:xmltext` depends on the context within which the document view is being used:

- **Export:** In the context of export, 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`. If there exists a JCR node `jcr:xmltext` which does not conform to the above description (for example, it has more than one property, or it has one property but that property is not a single valued string property called `jcr:xmlcharacters`) then that `jcr:xmltext` node is ignored. Entity references are used to escape characters (such as & and <) which should not be included as literals within XML text (see [6.4.4 Escaping of JCR Values](#)).
- **XPath:** In the context of an XPath query `jcr:xmltext` nodes and `jcr:xmlcharacters` properties are treated just like any other nodes and properties, appearing as elements and attributes, respectively, within the virtual document view stream against which XPath queries are run.

6.4.2.4 Invalid JCR Item Names

If the name of a JCR item `I` is not a valid XML element or attribute name (as the case may be) then how it is handled depends upon the context in which the document view is being used:

- **Export:** In the context of export, the repository may either ignore the item in question or employ the escaping scheme described below (see [6.4.3 Escaping JCR Names](#)). Which approach taken is up to the implementation.

- **XPath:** In the context of an XPath query, the escaping scheme described below (see [6.4.3 Escaping JCR Names](#)) *must* be used in the virtual document view XML stream against which the query is run. Consequently, the same escaping scheme must be used within any XPath statement that refers to the item `Ⓜ`.

6.4.2.5 Multi-value Properties

If a multi-value JCR property `Ⓜ` is encountered, then its treatment depends on the context within which the document view is being used:

- **Export:** In the context of export, the repository may either ignore the multi-value property or serialize it as an attribute whose value is an XML Schema list type⁴ (i.e., a whitespace-delimited list of strings). If the latter approach is taken then:
 - Each value in the property is converted to a string according to standard conversion, see [6.2.6 Property Type Conversion](#).
 - All literal whitespace within each string is escaped, as well as any characters (such as `&` and `<`) that should not be included as literals in any case, see [6.4.4 Escaping of JCR 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 the escaping of space literals must be applied to the value of all single-value properties as well, see [6.4.4 Escaping of JCR Values](#).
- **XPath:** In the context of an XPath query, the value array of property `Ⓜ` is mapped to a pseudo list type attribute value. We call it a *pseudo* list type because space delimiters are not used and consequently space literals within individual values are not escaped, nor are other characters such as `&` and `<` that would normally be escaped. This is possible because the XML stream in the XPath context is virtual and therefore it need never be actually serialized. However,

⁴ See <http://www.w3.org/TR/xmlschema-0/#ListDt> for more information about the XML Schema list type.

tests against multi-value properties in XPath *using general comparison* operators act as they would if the multi-value property were a list-type attribute, except that spaces and characters like & and < within individual value literals used within the XPath statement are not escaped (see [6.6.3.10 Searching Multi-value Properties](#)).

6.4.2.6 Example

Given the same subtree and namespace settings as used in the system view example (see 6.4.1.1 Example), the document view would look like this (note that the following example assumes that multi-value property serialization is not supported and therefore escaping of space literals is not done (see [6.4.4 Escaping of JCR Values](#)):

```
<?xml version="1.0" encoding="UTF-8"?>
<myapp:document xmlns:jcr="http://www.jcp.org/jcr/1.0"
  xmlns:nt="http://www.jcp.org/jcr/nt/1.0"
  xmlns:mix="http://www.jcp.org/jcr/mix/1.0"
  xmlns:sv="http://www.jcp.org/jcr/sv/1.0"
  xmlns:myapp="http://mycorp.com/myapp"
  jcr:primaryType="nt:unstructured">
  myapp:title="JSR 170"
  myapp:lead="Content Repository">
  <myapp:body jcr:primaryType="nt:unstructured">
    <myapp:paragraph jcr:primaryType="nt:unstructured"
      myapp:title="Node Types"
      myapp:text="An important feature..." />
  </myapp:body>
</myapp:document>
```

Note that in the above, the XML has been formatted for readability. The actual XML stream would not have any extraneous whitespace between elements or attributes.

6.4.3 Escaping of JCR Names

Not every JCR name is a valid XML name. In particular, even though a JCR prefix is always a valid XML prefix, the JCR local name (the part after the colon, or the whole name, if there is no prefix) may not be a valid XML name. For example, a JCR name may contain spaces, whereas XML names cannot.

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.

The escape character is "_". Any invalid character is escaped as `_xHHHH_`, where `HHHH` is the four-digit hexadecimal UTF-16 code for the character. If the character corresponds to a code point outside the BMP (Unicode's *Basic Multilingual Plane*) then a UTF-32 encoding is used and the character is escaped as `_xHHHHHHHHH_`.

where `HHHHHHHH` is the eight-digit hexadecimal UTF-32 code for the character (note that UTF-16 surrogates are *not* used in this case, only full UTF-32 encodings).

The underscore character itself, if appearing as a literal within a JCR name, is only encoded if it is the first character in a character sequence that could be misinterpreted as an escape sequence.

So, for example,

- `"My Documents"` is converted to `"My_x0020_Documents"`,
- `"My_Documents"` is not encoded,
- `"My_x0020Documents"` is not encoded either,
- but `"My_x0020_Documents"` is encoded as `"My_x005f_x0020_Documents"`.

6.4.4 Escaping of JCR Values

When a JCR value is serialized during either system view or document view export, it is first converted to string form using standard JCR conversion, see [6.2.6 Property Type Conversion](#).

Within the resulting string, any occurrence of one of the five characters corresponding to the five *predefined entity references* in XML, ampersand (&), less-than symbol (<), greater-than symbol (>), apostrophe ('), and quotation mark (") must be escaped as `&`, `<`, `>`, `'` and `"`, respectively.

Furthermore, if multi-value property serialization is supported (see [6.4.2.5 Multi-value Properties](#)) then, in both the case of a single-value and the case of a multi-value property, 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 [6.4.3 Escaping of JCR 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_`.

If multi-value property serialization is not supported, then this second step of encoding whitespace is not required.

6.5 Exporting Repository Content

Level 1 supports the export of repository content to both system view XML and document view XML. The XML can be output either in as a stream or as SAX events.

The export methods are found in the `Session` object:

javax.jcr. Session	
void	<pre>exportSysView(String absPath, ContentHandler contentHandler, boolean skipBinary, boolean noRecurse)</pre> <p>Serializes the node (and if <code>noRecurse</code> is <code>false</code>, the whole subtree) at <code>absPath</code> into a series of SAX events by calling the methods of the supplied <code>org.xml.sax.ContentHandler</code>. The resulting XML is in the <i>system view</i> form. Note that <code>absPath</code> must be the path of a node, not a property.</p> <p>If <code>skipBinary</code> is <code>true</code> then any properties of <code>PropertyType.BINARY</code> will be ignored and will not appear in the serialized output. If <code>skipBinary</code> is <code>false</code> then the actual value of each <code>BINARY</code> property is recorded using Base64 encoding.</p> <p>If <code>noRecurse</code> is <code>true</code> then only the node at <code>absPath</code> and its properties, but not its child nodes, are serialized. If <code>noRecurse</code> is <code>false</code> then the entire subtree rooted at <code>absPath</code> is serialized.</p> <p>If the user lacks read access to some subsection of the specified tree that section simply does not get serialized, since, from the user's point of view it is not there.</p> <p>The serialized output will reflect the state of the current workspace as modified by the state of this <code>Session</code>. This means that pending changes (regardless of whether they are valid according to node type constraints) and the current session-mapping of namespaces are reflected in the output.</p> <p>A <code>PathNotFoundException</code> is thrown if no node exists at <code>absPath</code>.</p> <p>A <code>SAXException</code> is thrown if an error occurs while feeding events to the <code>ContentHandler</code>.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>
void	<pre>exportSysView(String absPath, OutputStream out, boolean skipBinary,</pre>

	<p>boolean noRecurse)</p> <p>Serializes the node (and if noRecurse is false, the whole subtree) at absPath into an XML stream and outputs it through the supplied outputStream. The resulting XML is in the <i>system view</i> form. Note that absPath must be the path of a node, not a property.</p> <p>If skipBinary is true then any properties of PropertyType.BINARY will be ignored and will not appear in the serialized output. If skipBinary is false then the actual value of each BINARY property is recorded using Base64 encoding.</p> <p>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 subtree rooted at absPath is serialized.</p> <p>If the user lacks read access to some subsection of the specified tree that section simply does not get serialized, since, from the user's point of view it is not there.</p> <p>The serialized output will reflect the state of the current workspace as modified by the state of this session. This means that pending changes (regardless of whether they are valid according to node type constraints) and the current session-mapping of namespaces are reflected in the output.</p> <p>The output XML will be encoded in UTF-8.</p> <p>An IOException is thrown if an I/O error occurs.</p> <p>A PathNotFoundException is thrown if no node exists at absPath.</p> <p>A RepositoryException is thrown if another error occurs.</p>
void	<p>exportDocView(String absPath, ContentHandler contentHandler, boolean skipBinary, boolean noRecurse)</p> <p>Serializes the node (and if noRecurse is false, the whole subtree) at absPath into a series of SAX events by calling the methods of the supplied org.xml.sax.ContentHandler. The resulting XML is in the <i>document view</i> form. Note that absPath must be the path of a node, not a property.</p> <p>If skipBinary is true then any properties of</p>

	<p>PropertyType.BINARY will be ignored and will not appear in the serialized output. If skipBinary is false then the actual value of each BINARY property is recorded using Base64 encoding.</p> <p>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 subtree rooted at absPath is serialized.</p> <p>If the user lacks read access to some subsection of the specified tree that section simply does not get serialized, since, from the user's point of view it is not there.</p> <p>The serialized output will reflect the state of the current workspace as modified by the state of this Session. This means that pending changes (regardless of whether they are valid according to node type constraints) and the current session-mapping of namespaces are reflected in the output.</p> <p>A PathNotFoundException is thrown if no node exists at absPath.</p> <p>A SAXException is thrown if an error occurs while feeding events to the ContentHandler.</p> <p>A RepositoryException is thrown if another error occurs.</p>
void	<pre>exportDocView(String absPath, OutputStream out, boolean skipBinary, boolean noRecurse)</pre> <p>Serializes the node (and if noRecurse is false, the whole subtree) at absPath into an XML stream and outputs it through the supplied OutputStream. The resulting XML is in the <i>document view</i> form. Note that absPath must be the path of a node, not a property.</p> <p>If skipBinary is true then any properties of PropertyType.BINARY will be ignored and will not appear in the serialized output. If skipBinary is false then the actual value of each BINARY property is recorded using Base64 encoding.</p> <p>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 subtree rooted at absPath is serialized.</p>

	<p>If the user lacks read access to some subsection of the specified tree that section simply does not get serialized, since, from the user's point of view it is not there.</p> <p>The serialized output will reflect the state of the current workspace as modified by the state of this <code>session</code>. This means that pending changes (regardless of whether they are valid according to node type constraints) and the current session-mapping of namespaces are reflected in the output.</p> <p>The output XML will be encoded in UTF-8.</p> <p>An <code>IOException</code> is thrown if an I/O error occurs.</p> <p>A <code>PathNotFoundException</code> is thrown if no node exists at <code>absPath</code>.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>
--	--

6.5.1 Encoding

XML streams produced by JCR serialization must be encoded in UTF-8.

6.6 Searching Repository Content

JCR supports the searching of repository content. In level 1 support for the XPath syntax is required. Optionally, a repository may support the SQL syntax (see [8.6 Searching Repository Content with SQL](#)). Implementations may also support additional languages.

6.6.1 XPath in JCR

XPath is a search language originally designed for selecting elements from an XML document. Since a JCR workspace, like an XML document, is essentially a tree of elements, XPath provides a convenient syntax for searching JCR content. The main prerequisite for providing XPath querying over JCR content is to establish an XML mapping of the JCR workspace tree. Having already established two such mappings (system and document view) for purposes of serialization and deserialization, we simply re-use one of them, the document view, as the basis against which an XPath query is run (see [6.4.2 Document View XML Mapping](#)).

6.6.2 XPath over Document View

When an XPath query is executed, the XPath expression specified is applied to the document view of the workspace being searched.

For example, consider a workspace with the following structure (based on the earlier example in Section [6.4.1.1](#), with the addition of a top-most root node):

```
<?xml version="1.0" encoding="UTF-8"?>
<jcr:root xmlns:jcr="http://www.jcp.org/jcr/1.0"
          xmlns:nt="http://www.jcp.org/jcr/nt/1.0"
          xmlns:mix="http://www.jcp.org/jcr/mix/1.0"
          xmlns:sv="http://www.jcp.org/jcr/sv/1.0"
          xmlns:myapp="http://mycorp.com/myapp"
          jcr:primaryType="nt:unstructured">

  <myapp:document jcr:primaryType="nt:unstructured"
                 myapp:title="JSR 170"
                 myapp:lead="Content Repository">
    <myapp:body jcr:primaryType="nt:unstructured">
      <myapp:paragraph jcr:primaryType="nt:unstructured"
                     myapp:title="Node Types"
                     myapp:text="An important feature..." />
    </myapp:body>
  </myapp:document>
</jcr:root>
```

(Note that in the above, the XML has been formatted for readability. The actual XML stream would not have any extraneous whitespace between elements or attributes.)

In this case, to find the node called `myapp:paragraph`, the following XPath expression would be used:

```
//myapp:paragraph
```

This query will return the node at JCR path
`/myapp:document/myapp:body/myapp:paragraph`

6.6.3 Adapting XPath to JCR

XPath 2.0 forms the basis of the level 1 querying in JCR. However, many features of XPath 2.0 are either not applicable to JCR or may impose unwarranted burdens on some JCR implementations (depending on their underlying design). As a result, we define a minimal subset of features that must be supported by all repositories, and which is chosen with an eye to broad adoption. Implementations may, of course, support additional features of XPath as well.

The following presents a feature-by-feature discussion of the minimal requirements. As well, some of the more common possible extensions are mentioned. For those common extensions, a discovery mechanism is provided in the form of additional descriptor keys whose presence in the array returned by `Repository.getDescriptorKeys()` indicates that feature in question is supported (see [6.1.1.1 Repository Descriptors](#)).

6.6.3.1 Same-Name Siblings

The syntax used to address same-name sibling nodes in a JCR path is purposely similar to the XPath abbreviated syntax for addressing sibling XML elements with the same name, i.e., the "square bracket index notation" where the first sibling is indicated by [1] (not [0]) the second by [2] and so forth.

However, because some implementations (those built on an underlying relational model, for example) may find it difficult to support querying on the basis of node position, this feature is optional. The presence of the descriptor key `QUERY_XPATH_POS_INDEX` in the array returned by `Repository.getDescriptorKeys()` indicates that the index position notation for same-name siblings is supported.

6.6.3.2 Document Order

The results returned by an XPath query *may* reflect the document order of the elements returned (see [6.6.12 Query Results](#)). Support for this is optional.

Note that the reason for this is that in some implementations, a query may be performed using the SQL syntax. In that case there is no concept of document order, since this concept applies only to XML views, not database views (see [8.6 Searching Repository Content with SQL](#)).

If document order searching is supported, then the context functions related to document order, `last()` and `position()`, must also be supported.

The presence of the descriptor key `QUERY_XPATH_DOC_ORDER` in the array returned by `Repository.getDescriptorKeys()` indicates that document order searching is supported.

6.6.3.3 Context Node

The context node of an XPath query is the XML node relative to which the query expression is evaluated.

A relative XPath statement (one that does not have a leading `/`) will be interpreted relative to the root node of the workspace, which, in the XML document view is the top-most XML element, `<jcr:root>`. This means that one should not include `jcr:root` as the first segment in a relative XPath statement, since that element is already the default context node.

An absolute XPath (one with a leading `/`), in contrast, will be interpreted relative to a position one level above `<jcr:root>`. This means that an absolute XPath must either begin with `//` or with `/jcr:root` in order to match anything at all.

6.6.3.4 Mapping JCR property types to XML types

The following table outlines the mapping between JCR property types to XML Schema types.

JCR Property Type	XML Schema Type
String	xs:string
Binary	xs:base64Binary
Double	xs:double
Long	xs:long
Boolean	xs:boolean
Date	xs:dateTime
Path	xs:string
Name	xs:string
Reference	xs:IDREF

6.6.3.5 Abbreviated Syntax

Only support for the abbreviated syntax of XPath is required.

6.6.3.6 Axes

As part of a location path, the only axes for which support is required are:

child (in abbreviated syntax this is the default axis, represented simply by /, the location path separator).

descendant-or-self (abbreviated syntax: //).

attribute (abbreviated syntax: @).

Support for the other axes is not required.

6.6.3.7 Predicates

Support for predicates in the location path is required. For example, the following query would be supported:

Find all employees who have a secretary and an assistant property:

```
//employee[@secretary and @assistant]
```

Find Reports of year 2000 that was authored by Randy or David:

```
//Reports/2000/*[@Author = 'Randy' or @Author = 'David']
```

6.6.3.8 Boolean Functions

The boolean functions `not()`, `true()` and `false()` are required.

6.6.3.9 Escaping

Within an XPath statement, the same escaping rules apply as do to the document view XML serialization. See [6.4.2.3 Escaping of JCR Names](#) and [6.4.3 Escaping Quotation Marks](#).

6.6.3.10 Searching Multi-value Properties

XML Schema supports attributes having types. Among those is one called the list type⁵. This type specifies that white space within an attribute value serve as the delimiters between individual list items.

Additionally, XPath already distinguishes between two types of comparison operators: general comparisons (`=`, `!=`, `<`, `>`, `<=` and `>=`) and value comparisons (`eq`, `ne`, `lt`, `le`, `gt`, and `ge`). General comparisons, when applied to a list type attribute value, test for the existence (or non-existence, or other relationship etc.) of the specified argument among the elements in the list. Value comparisons test the entire attribute value as a single unit. In cases where an attribute only has one value the general and value comparisons are identical.

In the virtual XML document against which XPath queries are run, multi-value properties will be mapped to XML attributes with values in a form *similar* to the XML list type.

However, since the XML document is virtual, it need never be actually serialized. As a result, we do not need to specify white space as the delimiter. Doing so would require that white space that occurs within a value of a multi-value property be escaped when converted to document view, which would in turn require use of awkward escaping characters within XPath queries that tested for such values.

Instead, we simply specify that tests against multi-value properties *using general comparison* operators act as they would if the multi-value property *were* a list type attribute, except that spaces within individual values used within the test are not escaped. For example:

```
//x/y[@p = "hello again"]
```

would return all nodes `y` that are children of a node `x` and that have a property `p` which has at least one value `"hello again"`.

⁵ See <http://www.w3.org/TR/xmlschema-0/#ListDt> for more information about the XML Schema list type.

Additionally, tests against multi-value properties *using value comparison* operators will only return true if the multi-value property actually has only one value (and that value meets the test).

Note that multi-value properties in document view will be handled differently for purposes of export (see [6.4.2 Document View XML Mapping](#)).

In order to ensure mutual translatability between XPath and SQL (see [8.6 Searching Repository Content with SQL](#)) we mandate support only for the XPath general comparison operators = and !=. The other XPath general comparison operators are not translatable to SQL and therefore are optional. The value comparison operators must all be supported. The mapping to SQL is:

XPath	SQL
=	IN
!=	NOT IN
eq	=
ne	<>
lt	<
le	<=
gt	>
ge	>=

Note that in JCR XPath support is only required for comparisons of the form `<property><op><literalvalue>` and `<literalvalue><op><property>`. For example, support for `[@p = "hello"]` and `["hello" = @p]` (and so forth for each operator) is required. However, support for `[@p eq @q]` and `[@p = @q]` (and so forth for each operator) is not required.

6.6.4 XPath Extensions

XPath in JCR also defines a small number of functions and one syntactic addition (the order-by clause). JCR-related functions are prefixed with `jcrfn:`.

Note that the function signatures below are expressed in XPath terminology. In particular, the reference to the type `element()` means an XML element, which corresponds to a JCR node. Similarly, the type `attribute()` refers to an XML attribute, which corresponds to a JCR property.

6.6.4.1 *like* function

This function is based on the **LIKE** predicate found in SQL. Support for this function is required.

```
jcrfn:like($property as attribute(),
           $pattern as xs:string) as xs:boolean
```

For example, the query “Find all paragraphs whose title property has the word ‘Java’”, is expressed as:

```
//para[jcrfn:like(title,'%Java%')]
```

As in SQL, the character ‘%’ represents any string of zero or more characters, and the character ‘_’ (underscore) represents any single character. Any literal use of these two characters must be escaped with a backslash (“\”). Consequently, any literal instance of a backslash must also be escaped, resulting in a double backslash (“\\”).

6.6.4.2 *contains* function

This function is required for content repositories that support text search.

```
jcrfn:contains($searchexp as xs:string) as xs:boolean
```

This function is used to embed a statement in a full-text search language. It is functionally equivalent to the SQL **CONTAINS** function (for level 2 implementations) described in [8.6.4.5 CONTAINS](#).

The relevant EBNF for the required search expression syntax is:

```
searchexp ::= '"' [-]term {[OR][-]term} '"'
term ::= word | '"' word {whitespace word} '"'
word ::= /* A string containing no whitespace */
whitespace ::= /* A string of only whitespace*/
```

At minimum, all implementations must support the *simple search-engine syntax* defined by *searchexp* in the EBNF above. This syntax is based on the syntax of search engines like Google.

The semantics of the simple search expression are as follows:

- Terms separated by whitespace are implicitly **ANDed** together.
- Terms may also be **ORed** with explicit use of the **OR** keyword.
- Terms may be excluded by prefixing with a – (minus sign) character. This means that the result set *must not* contain the excluded term.

- A term may be either a single word or a phrase delimited by single quotes ("'") or double quotes (").
- The entire text search expression (*searchexp* in the EBNF, above) *must be* delimited by double quotes ("").
- Within the *searchexp* literal instances of single quote ("'"), double quote (""), and hyphen ("-") must be escaped with a backslash ("\"). Backslash itself must therefore also be escaped, ending up as double backslash ("\\").

The scope of the `jcrfn:contains` clause is the intersection of two sets. These two sets are:

- The values of those properties that are the immediate children of the context node-set at the position of the `jcrfn:contains` function within the XPath statement.
- The contents of the full-text index of the repository. A repository may, for example, index only the values of `STRING` properties. Additionally, it may index some binary properties according to some application-specific encoding. The scope of full-text indexing is implementation specific.

For example, the query "Find all title properties that contain the text 'JCR' " is expressed as:

```
//myapp:title[jcrfn:contains("JCR")]
```

The relevance score for each matching node is returned as the pseudo-property `jcr:score` (if supported, see [6.1.1.1 Repository Descriptors](#)). The specification does not define the range of the `jcr:score` value, it is implementation specific.

An implementation may choose to support other embedded full-text search languages other than the simple search engine style shown here.

6.6.4.3 *deref* function

This function is used follow a **REFERENCE** property into the target reference. Support for this function is optional.

```
jcrfn:deref($source as element()*,
           $node-test as xs:string) as element()*
```

The first argument is a node sequence in which each (XML) node represents a JCR reference property.

The second argument is a node test which is a string to match the target node name. The function returns a node sequence of all target nodes matching the node-test. An error is raised if the reference cannot be resolved.

For example, suppose there is a property of type **REFERENCE** called **myapp:author** which refers to a node representing the author of this document. A query expression to find the person's last name property would be:

```
//myapp:myDoc/
    deref(@myapp:author, 'myapp:person')/lastName
```

The dereference expression above evaluates to a node which has the name **myapp:person**. Subsequently a child node or property of person's last name is selected.

The target reference can be '*' if it is not known, for example, to find a descendant node address under a target reference described above,

```
//myapp:myDoc/deref(@myapp:author, '*')//myapp:Address
```

6.6.4.4 order by clause

It may be desirable to sort the result of an XPath query into either ascending or descending order based on the value of a property. This task can be achieved by adding the order by clause to a location path expression. The implementation of this feature is optional. The additions to the syntax of the XPath location clause are as follows:

```
JCRXPathExpr      ::=  (XPath OrderByClause)?
OrderByClause      ::=  <"order" "by"> OrderSpecList
OrderSpecList      ::=  OrderSpec ("," OrderSpec)*
OrderSpec          ::=  SimpleAddExpr OrderModifier
OrderModifier       ::=  ("ascending" | "descending")?
SimpleAddExpr       ::=  SimpleMultiExpr ( "+" | "-" )
                      SimpleMultiExpr *
SimpleMultiExpr     ::=  SimpleUnaryExpr ( "*" | "div" | "idiv"
                      | "mod" ) SimpleUnaryExpr *
SimpleUnaryExpr     ::=  ("-" | "+")* SimpleExpr
SimpleExpr          ::=  "@" AttributeName | Literal | "text-
                      score" "(" "." " " "," TextSearchString ")"
```

The order-by clause sorts the result of the location path expression according to the order specification. The relative location path expression specifies the sort key element as a relative path from each node in the result. This feature is available in XQuery 1.0.

For example, the query “Find all paragraphs with the type property with value ‘warning’ under ‘section’ and sort them by title property in ascending order” can be expressed as:

```
//section/para[@type='warning'] order by title ascending
```

6.6.5 XPath Grammar

The following grammar defines the required subset of XPath in JCR. Text in *Courier New* indicates parts of standard XPath 2.0 that are required in JCR. Text in ~~*Courier New strikethrough*~~ indicates parts of standard XPath 2.0 that are not required in JCR.

6.6.5.1 Named Terminals

- | | | | |
|------|---------------------------------|----------------|---|
| [1] | ExprComment | ::= | "(: " (ExprCommentContent ExprComment)* " :) " |
| [2] | ExprCommentContent | ::= | Char |
| [3] | IntegerLiteral | ::= | Digits |
| [4] | DecimalLiteral | ::= | ("." Digits) (Digits "." [0-9]*) |
| [5] | DoubleLiteral | ::= | ((("." Digits) (Digits "." [0-9]*)?)) ("e" "E") ("+" "-")? Digits |
| [6] | StringLiteral | ::= | ('' ' ((' ' ' ' ' ') [^"]) * ' ' ' ') (' ' ' ' ((' ' ' ' ' ') [^']) * ' ' ' ') |
| [7] | SchemaGlobalTypeName | ::= | "type" - "(" QName - ")" " |
| [8] | SchemaGlobalContext | ::= | QName SchemaGlobalTypeName |
| [9] | SchemaContextStep | ::= | QName |
| [10] | Digits | ::= | [0-9]+ |
| [11] | NCName | ::= | [http://www.w3.org/TR/REC-xml-names/#NT-NCName] |
| [12] | VarName | ::= | QName |
| [13] | QName | ::= | [http://www.w3.org/TR/REC-xml-names/#NT-QName] |
| [14] | Char | ::= | [http://www.w3.org/TR/REC-xml#NT-Char] |

6.6.5.2 Non-Terminals

[15]	XPath	::=	Expr?
[16]	Expr	::=	ExprSingle ("," ExprSingle)*
[17]	ExprSingle	::=	ForExpr QuantifiedExpr IfExpr OrExpr
[18]	ForExpr	::=	SimpleForClause "return" ExprSingle
[19]	SimpleForClause	::=	<"for" "\$"> VarName "in" ExprSingle ("," "\$" VarName "in" ExprSingle)*
[20]	QuantifiedExpr	::=	(<"some" "\$"> <"every" "\$">) VarName "in" ExprSingle ("," "\$" VarName "in" ExprSingle)* "satisfies" ExprSingle
[21]	IfExpr	::=	<"if" "(" Expr ")" "then" ExprSingle "else" ExprSingle
[22]	OrExpr	::=	AndExpr ("or" AndExpr)*
[23]	AndExpr	::=	InstanceofExpr ("and" InstanceofExpr)*
[24]	InstanceofExpr	::=	TreatExpr (<"instance" "of"> SequenceType)?
[25]	TreatExpr	::=	CastableExpr (<"treat" "as"> SequenceType)?
[26]	CastableExpr	::=	CastExpr (<"castable" "as"> SingleType)?
[27]	CastExpr	::=	ComparisonExpr (<"cast" "as"> SingleType)?
[28]	ComparisonExpr	::=	RangeExpr ((ValueComp GeneralComp NodeComp) RangeExpr)?
[29]	RangeExpr	::=	AdditiveExpr ("to" AdditiveExpr)?

[30]	AdditiveExpr	::=	MultiplicativeExpr (("+" "-") MultiplicativeExpr)*
[31]	MultiplicativeExpr	::=	UnaryExpr (("*" "div" "idiv" "mod") UnaryExpr)*
[32]	UnaryExpr	::=	("-" "+")* UnionExpr
[33]	UnionExpr	::=	IntersectExceptExpr (("union" " ") IntersectExceptExpr)*
[34]	IntersectExceptExpr	::=	ValueExpr (("intersect" "except") ValueExpr)*
[35]	ValueExpr	::=	PathExpr
[36]	PathExpr	::=	("/" RelativePathExpr?) ("//" RelativePathExpr) RelativePathExpr
[37]	RelativePathExpr	::=	StepExpr (("/" "//") StepExpr)*
[38]	StepExpr	::=	AxisStep FilterStep
[39]	AxisStep	::=	(ForwardStep ReverseStep) Predicates
[40]	FilterStep	::=	PrimaryExpr Predicates
[41]	ContextItemExpr	::=	","
[42]	PrimaryExpr	::=	Literal VarRef ParenthesizedExpr ContextItemExpr FunctionCall
[43]	VarRef	::=	"\$" VarName
[44]	Predicates	::=	("[" Expr "]")*
[45]	GeneralComp	::=	"=" "!=" "<" "<="

		">" ">="
[46]	ValueComp	::= "eq" "ne" "lt" "le" "gt" "ge"
[47]	NodeComp	::= "is" "<<" ">>"
[48]	ForwardStep	::= (ForwardAxis NodeTest) AbbrevForwardStep
[49]	ReverseStep	::= (ReverseAxis NodeTest) AbbrevReverseStep
[50]	AbbrevForwardStep	::= "@"? NodeTest
[51]	AbbrevReverseStep	::= ".."
[52]	ForwardAxis	::= <"child" "::"> <"descendant" "::"> <"attribute" "::"> <"self" "::"> <"descendant or self" "::"> <"following sibling" "::"> <"following" "::"> <"namespace" "::">
[53]	ReverseAxis	::= <"parent" "::"> <"ancestor" "::"> <"preceding sibling" "::"> <"preceding" "::"> <"ancestor or self" "::">
[54]	NodeTest	::= KindTest NameTest
[55]	NameTest	::= QName Wildcard
[56]	Wildcard	::= "*" <NCName ":" "*"> <"*" ":" NCName>
[57]	Literal	::= NumericLiteral StringLiteral
[58]	NumericLiteral	::= IntegerLiteral DecimalLiteral

DoubleLiteral

[59]	ParenthesizedExpr	::=	"(" Expr? ")"
[60]	FunctionCall	::=	<QName "> (ExprSingle ("," ExprSingle)*)? ">
[61]	SingleType	::=	AtomicType "?"
[62]	SequenceType	::=	{ItemTest OccurrenceIndicator?} <"empty" "(" ">">
[63]	AtomicType	::=	QName
[64]	ItemTest	::=	AtomicType KindTest <"item" "(" ">">
[65]	KindTest	::=	DocumentTest ElementTest AttributeTest PITest CommentTest TextTest AnyKindTest
[66]	ElementTest	::=	<"element" "> {((SchemaContextPath ElementName) (ElementNameOrWildcard ("," TypeNameOrWildcard "nillable"?))?) }"
[67]	AttributeTest	::=	<"attribute" "> {((SchemaContextPath AttributeName) (AttributeNameOrWildcard ("," TypeNameOrWildcard)?)?)? ">
[68]	ElementName	::=	QName
[69]	AttributeName	::=	QName
[70]	TypeName	::=	QName
[71]	ElementNameOrWildca	::=	ElementName "*"

rd

- [72] ~~AttribNameOrWildcard ::= AttributeName | "*" | &~~
- [73] ~~TypeNameOrWildcard ::= TypeName | "*" | &~~
- [74] ~~PITest ::= <"processing-instruction" "("> (NCName | StringLiteral)? ")"~~
- [75] ~~DocumentTest ::= <"document-node" "("> ElementTest? ")"~~
- [76] ~~CommentTest ::= <"comment" "("> ")"~~
- [77] ~~TextTest ::= <"text" "("> ")"~~
- [78] ~~AnyKindTest ::= <"node" "("> ")"~~
- [79] ~~SchemaContextPath ::= <SchemaGlobalContext "/"> <SchemaContextStep "/">*~~
- [80] ~~OccurrenceIndicator ::= "?" | "*" | "+"~~

6.6.5.3 Notes on the Grammar

Required function support ([42] and [60] above) is limited to the functions described in [6.6.4 XPath Extensions](#).

6.6.6 Search Scope

A query searches the persistent workspace associated with the current session. It does not search any pending changes that may be recorded on the session but not yet saved.

6.6.7 Query API

The query facility in JCR is accessed through the **QueryManager** object. The **Workspace** interface provides access to the **QueryManager** object:

javax.jcr. Workspace	
QueryManager	getQueryManager()

	<p>Returns the <code>QueryManager</code>, through which search methods are accessed.</p> <p>A <code>RepositoryException</code> is thrown if an error occurs.</p>
--	--

6.6.8 QueryManager

The `QueryManager` object provides methods for creating queries, retrieving saved queries and for discovering supported query languages:

<code>javax.jcr.query.</code> QueryManager	
Query	<p>createQuery(String statement, String language)</p> <p>Creates a new query by specifying the query <code>statement</code> itself and the <code>language</code> in which the query is stated. If the query <code>statement</code> is syntactically invalid, given the language specified, an <code>InvalidQueryException</code> is thrown. The <code>language</code> parameter must be a string from among those returned by <code>QueryManager.getSupportedQueryLanguages()</code>; if it is not, then an <code>InvalidQueryException</code> is thrown.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>
Query	<p>getQuery(Node node)</p> <p>Retrieves an existing persistent query. If <code>node</code> is not a valid persisted query (that is, a node of type <code>nt:query</code>), an <code>InvalidQueryException</code> is thrown.</p> <p>Persistent queries are created by first using <code>QueryManager.createQuery</code> to create a <code>Query</code> object and then calling <code>Query.save</code> to persist the query to a location in the workspace.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>
String[]	<p>getSupportedQueryLanguages()</p> <p>Returns an array of strings identifying the supported query languages. In level 1 this set must include the string represented by the constant <code>Query.XPATH</code>. If SQL is supported it must additionally include <code>Query.SQL</code>. An implementation of either level may also support other languages.</p>

6.6.9 The Query Object

A new query is created by calling `QueryManager.createQuery`. The returned `Query` object has the following methods:

javax.jcr.query. Query	
QueryResult	execute() Executes this query and returns a QueryResult object. Throws a RepositoryException if an error occurs.
String	getStatement() Returns the statement defined for this query.
String	getLanguage() Returns the language set for this query. This will be one of the QueryLanguage constants returned by QueryManager.getSupportedQueryLanguages() .
String	getPersistentQueryPath() If this is a persistent query (that is, a Query object that has been saved using Query.save or retrieved using QueryManager.getQuery()), then this method returns the path of the nt:query node that stores the persistent query. If this is a transient query (that is, a Query object created with QueryManager.createQuery and not yet saved), then this method throws an ItemNotFoundException . Throws a RepositoryException if another error occurs.
void	save(String absPath) Persists this Query as a node in content. In a level 1 repository this method throws an UnsupportedRepositoryOperationException . In a level 2 repository it persists this Query as a node of type nt:query at absPath . If this method succeeds, it persists the new nt:query node immediately; there is no need to call Item.save or Session.save . If an item already exists at absPath , an ItemExistsException is thrown. A VersionException is thrown if the parent node of absPath is versionable and checked-in or non-versionable and its nearest versionable ancestor is

	<p>checked-in.</p> <p>A <code>PathNotFoundException</code> is thrown if the path cannot be found.</p> <p>A <code>ConstraintViolationException</code> if creating the node would violate a node type (or other implementation specific) constraint.</p> <p>A <code>LockException</code> is thrown if a lock prevents the save.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>
<code>String</code>	<p>XPATH</p> <p>A string constant representing the XPath query language applied to the <i>document view</i> XML mapping of the workspace.</p>
<code>String</code>	<p>SQL</p> <p>A string constant representing the SQL query language applied to the <i>database view</i> of the workspace. Support for this language is optional. See 8.6 Searching Repository Content with SQL.</p>

6.6.10 Persistent vs. Transient Queries

When a new `Query` object is first created with `QueryManager.createQuery` it is a *transient query*. If the repository supports the node type `nt:query`, then a transient query can be saved as content in the repository by calling `Query.save(String absPath)`. This creates an `nt:query` node at the specified path. Retrieving a saved query is done by passing the `nt:query` node to `QueryManager.getQuery(Node node)`.

6.6.11 Query Results

Once a query has been defined, it can be executed. The method `Query.execute()` returns a `QueryResult`.

The results returned always respect the access restrictions of the current session. In other words if the current session does not have read permissions to a particular item, then that item will not be included in the result set even if it would otherwise constitute a match.

As mentioned, all queries are run against the persistent state of a workspace, pending changes stored in the `session` are not searched. However, 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`, in other words the item state will reflect any pending changes currently stored in the session. As a result, it is possible that a property returned as a match will not reflect the value that caused it to *be* a match (i.e., its persistent state). Applications using a JCR implementation should therefore clear the `session` (either through `save` or `refresh(false)`) before running a query.

The `QueryResult` is returned in two formats: as a table with property names as the column names and a set of rows of values and as a list of nodes. The relationship between the two format is explained below.

Javax.jcr.query. QueryResult	
<code>String[]</code>	getPropertyNames() Returns an array of all the property names (column names) in this result set. Throws a <code>RepositoryException</code> if an error occurs.
<code>RowIterator</code>	getRows() Returns an iterator over the <code>Rows</code> of the query result table. If an <code>ORDER BY</code> clause was specified in the query, then the order of the returned rows in the iterator will reflect the order specified in that clause. If no items match, an empty iterator is returned. Throws a <code>RepositoryException</code> if an error occurs.
<code>NodeIterator</code>	getNodes() Returns an iterator over all nodes that match the query. If an <code>ORDER BY</code> clause was specified in the query, then the order of the returned nodes in the iterator will reflect the order specified in that clause. If no nodes match, an empty iterator is returned. Throws a <code>RepositoryException</code> if an error occurs.

javax.jcr.query. Row	
<code>Value[]</code>	getValues() Returns an array of all the values in the same order as the property names (column names) returned by <code>QueryResult.getPropertyNames()</code> .

	Throws a <code>RepositoryException</code> if an error occurs.
Value	<code>getValue(String propertyName)</code> Returns the value of the indicated property in this Row. If <code>propertyName</code> is not among the column names of the query result table, an <code>ItemNotFoundException</code> is thrown. Throws a <code>RepositoryException</code> if another error occurs.

6.6.11.1 Table vs. Node List

The results of a search are returned in two parallel forms: a table view and a node list view. The content of the two forms and their relationship to one another is defined below.

(Note that this section refers to the optional SQL syntax in addition to the required XPath syntax. See [8.6 Searching Repository Content with SQL](#) for more details about SQL.)

For a node `n` to be included in the iterator returned by `QueryResult.getNodes`, at least one of the following must be true:

- The query is in XPath and `n` corresponds to a document view XML element within the XPath return set.
- The query is in XPath and `n` is the parent of a property `p` where `p` corresponds to a document view XML attribute within the XPath return set.
- The query is in SQL and `n` corresponds to a row returned by `QueryResult.getRows`.

The `QueryResult.getRows` returns a list of rows comprising a table. For a row `r` to be in the table at least one of the following must be true:

- The query is in SQL and `r` is a row produced by the operation of the query on the database view of the repository workspace (see [8.6.2 Database View](#)).
- The query is in XPath and `r` corresponds to a node returned by `QueryResult.getNodes()`.

The column (property) names of the table are returned by `QueryResult.getPropertyNames`. For a column `c` to be in the table it must be a single-value property or pseudo-property (`jcr:score` or `jcr:path`) and not a multi-value property (regardless of how many values it actually holds) and at least one of the following must be true:

- The query is in SQL and `c` is named in the `SELECT` clause but `c` is not `jcr:score` or `jcr:path`.
- The query is in SQL, `c` is `jcr:path`, `jcr:path` is named in the `SELECT` clause (either explicitly or through a wildcard) and `QUERY_JCRPATH` is among the descriptor keys returned by `Repository.getDescriptorKeys()`.
- The query is in SQL, `c` is `jcr:score`, `jcr:score` is named in the `SELECT` clause (either explicitly or through a wildcard), `QUERY_JCRSCORE` is among the descriptor keys returned by `Repository.getDescriptorKeys()` and the query includes at least one `CONTAINS` function.
- The query is in SQL, the `SELECT` clause specifies `*` and `c` is the name of a property that is explicitly declared (non-residual) in a node type specified in the `FROM` clause.
- The query is in XPath and `c` is the name of a property that is explicitly declared (non-residual) in the node type specified in a comparison with the property `jcr:primaryType` (if no such comparison exists, then node type `nt:base` is assumed).
- The query is in XPath, `c` is `jcr:path` and `QUERY_JCRPATH` is among the descriptor keys returned by `Repository.getDescriptorKeys()`.
- The query is in XPath, `c` is `jcr:score`, `QUERY_JCRSCORE` is among the descriptor keys returned by `Repository.getDescriptorKeys()` and the query includes a `jcrfn:contains` function.

The value located at a row `R` and column `c` is the value of the property (or pseudo-property in the case of `jcr:path` and `jcr:score`) with name `c` in the node corresponding to row `R`.

The `jcr:path` column (if present) at each row `R` holds the absolute path of the node to which the Row `R` corresponds.

The `jcr:score` column (if present. i.e., if a full-text search was done) at each row `R` holds a relevance score for the node to which the Row `R` corresponds (see [6.6.5.3 contains function](#) and [8.6.5.5 CONTAINS](#)).

6.6.12 Permissions

The results returned by a search are, of course, subject to the same restrictions as any access to the repository via a particular session. In other words, a search will only return results from those sections of the repository for which the initiating session has the appropriate permissions.

6.7 Node Types

An important feature of many repositories is the ability to distinguish the entities stored in the repository by *type*. In a JCR repository, this is done by assigning *node types* to nodes.

Level 1 specifies methods for the following node type-related functions:

- Discovering the primary and mixin node types of an existing node.
- Discovering which node types are supported in a particular repository.
- Discovering the definition of a supported node type.
- Discovering the constraints placed on an existing node or property due to the node type of its parent.

Level 2 additionally specifies methods for:

- Assigning a primary node type to a node on node creation.
- Assigning additional (optional) mixin node types.

In this section we explain the level 1 node type functionality, see [7.4 Assigning Node Types](#) for level 2 node type functions. In some cases node type-related information accessible through the discovery methods will only be relevant to a level 2 implementation. Where this is the case, it is mentioned in the discussion below.

6.7.1 Node Type Configuration

This specification does not attempt to define methods for defining, creating or managing node types. The wide range of approaches used to type entities in existing repositories makes it very difficult to define a single mechanism for node type configuration. Therefore, this aspect of node type functionality is left up to the individual implementation. This specification limits itself to defining node type assignment and discovery.

6.7.2 What Constitutes a Node Type

In a JCR-compliant repository, a node type defines which child nodes and properties a node may (or must) have. In order to provide a set of discovery methods for node type information, the range of that information must be defined. To this end, this specification stipulates that every node type has the following attributes:

- **Name:** Every node type registered with the repository has a unique name. The naming conventions for node types are the same as for items (i.e., they may have a colon

delimited prefix). All predefined primary node types are, for example, prefixed with `nt`. Mixin types are prefixed with `mix`. See [6.7.19 Predefined Node Types](#).

- **Supertypes:** A node type may *extend* another node type (or more than one node type, if the implementation supports multiple inheritance). [6.7.8 Inheritance Among Node Types](#).
- **Mixin status:** A node type may be either primary or mixin. This status is part of the node type's definition. See [6.7.4 Primary and Mixin Node Types](#).
- **Orderable child nodes status:** A primary node type may specify that child nodes are *client-orderable*. If this status is set to `true`, then all nodes of that node type *must* support the method `Node.orderBefore`. If this status is set to `false`, then nodes of that node type *may* support this method. Only primary node types control a node's status in this regard. This setting on a mixin node type will not have any effect on the node. See [7.1.10 Ordering Child Nodes](#).
- **Property definitions:** A node type contains a set of definitions specifying the properties that nodes of this node type are allowed (or required) to have and the characteristics of those properties.
- **Child node definitions:** A node type contains a set of definitions specifying the child nodes that nodes of this node type are allowed (or required) to have and the characteristics of those child nodes (including, in turn, *their* node types).
- **Primary Item Name:** A node type may specify one child item (property or node) as the primary item. This indicator is used by the method `Node.getPrimaryItem()`. See [6.2.3 Node Read Methods](#).

6.7.3 Node Type Discovery in Level 1

Note that in a level 1 implementation clients will not be able to re-order, add or remove nodes or change properties in any case. However, the orderable-status, property definitions and child node definitions may still provide information related to write-capabilities that a level 1 implementation cannot in practice perform.

This might be the case, for example, if a particular node-type happens to be shared with a level 2 repository. In general, the node type discovery methods will reflect the definition of the node type, regardless of the level of repository in which the node type happens to be found.

For this reason, the descriptions in this section often refer to write-related issues that will only be applicable in a level 2 repository.

6.7.4 Primary and Mixin Node Types

In a JCR repository, every node has one and only one primary node type. This node type defines, as mentioned, a set of restrictions on the child items of the node.

In addition to its single primary node type, a node *may* also have any number of mixin node types assigned to it. A mixin type is similar to a primary type in that its definition has the same parameters. It differs, though, in that it provides *additional features* to a node, beyond those defined in the node type proper.

Furthermore, while a primary node type can be “instantiated” as a node (i.e., that node’s structure is fully defined by its primary node type) this is not the case with mixin types. A mixin type cannot serve, by itself, to define the structure of a node; it just adds properties and child node requirements to a node that already has a primary node type.

A particular supported node type is either a primary type or a mixin type; it cannot be both.

6.7.5 Special Properties *jcr:primaryType* and *jcr:mixinTypes*

A node's primary node type must be stored in content as a **NAME** property of that node called `jcr:primaryType`. Similarly, any mixin node types assigned to it must be recorded in the multi-value **NAME** property `jcr:mixinTypes`. These properties are used to persist node type information across serialization/deserialization cycles. See [7.4.5](#) *Serialization and Node Types*. Both of these properties are protected; they cannot be removed or changed by the client of the API. The `jcr:primaryType` and `jcr:mixinTypes` properties are specified in the predefined primary node type `nt:base`, which is the supertype of all other primary node types (be they JCR-defined or implementation/application specific).

6.7.6 Property Definitions

Each property definition contains the following information:

- The **name** of the property to which this definition applies.
- The required **type** of the property, if any.
- The **value constraints** on the property. That is, what range of possible values may be assigned to this property.
- The **default value** that the property will have if it is auto-created.

- Whether this property will be **auto-created** when its parent node is created.
- Whether the property is **mandatory**. A mandatory property is one that *must have a value*. If a mandatory property has no value from a node then **save** on that node will fail. Note that for single value properties having no property is equivalent to the property not existing; for multi-value properties having no value may mean either that the property does not exist or that it holds an empty array. Mandatory properties cannot be removed short of removing their parent node, nor can they be set to the empty array.
- The **onParentVersion** status of the property. This specifies what to do with this property if a new version of its parent node is checked-in.
- Whether the property is **protected**. A protected property is one which cannot be modified or removed (except by removing its parent) by the client of this API but which may be modified or removed by the repository implementation itself. The protected status of a property is only relevant if its parent node is *not protected* (see [6.7.7 Child Node Definitions](#)). If the parent node is protected, all its properties are automatically also protected.
- Whether this property can have **multiple values**, meaning that it stores an array of values, not just one. Note that this “multiple values” flag is special in that a given node type may have two property definitions that are identical in every respect except for the their “multiple values” status. For example, a node type can specify two string properties both called **x**, one of which is multi-valued and the other that is not. An example of such a node type is **nt:unstructured** (see [6.7.22.4 nt:unstructured](#)).

6.7.7 Child Node Definitions

Similarly, each child node definition contains the following information:

- The **name** of the child node to which this definition applies.
- The **required primary node types** for this child node. That is, the primary node types that this child node must have. This attribute may list more than one node type in those implementations that support multiple inheritance of primary node types.
- The **default primary node type** for this child node. This is the primary node type automatically assigned if no node type information is specified when the node is created.

- Whether this child node will be **auto-created** when its parent node is created.
- Whether the child node is **mandatory**. A mandatory child node is one that *must exist*. If a mandatory child node is missing from a parent node then **save** on the parent node will fail. Mandatory child nodes cannot be removed, short of removing their parent node.
- The **onParentVersion** status of the child node. This specifies what to do with the child node if its parent node is versioned.
- Whether the child node is **protected**. A protected node is one which cannot be modified (have child items added to it or removed from it) or be removed (except by removing its parent) by the client of this API but which may be modified or removed by the repository implementation itself. When a node is protected that status propagates to its properties, making them protected as well (regardless of their actual protected setting). This means that properties cannot be added, removed *or changed in value*.
- Whether this child node can have **same-name siblings**, meaning that the parent node can have more than one child node of this name.

6.7.8 Inheritance Among Node Types

A node type may have one (or in some implementations, more than one) *supertype*. A subtype inherits the property and child node definitions of its supertype(s) and may declare further property or child node definitions.

Some repositories may support multiple inheritance of node types. As a result, the methods for discovering node type information must allow for the possibility that a node type has more than one supertype. See [6.7.10](#) *Discovering the Definition of a Node Type*.

Configuring the inheritance hierarchy of node types available within a particular repository is outside the scope of this specification. For this reason the specification does not define how conflicts between multiple super types are resolved. See also, [6.7.22.2](#) *Additions to the Hierarchy*.

6.7.9 Discovering available Node Types

Discovering which node types are available in a particular implementation begins with the **NodeTypeManager** object, which is acquired via the **workspace**. Recall from earlier in the specification:

javax.jcr. Workspace	
NodeTypeManager	getNodeTypeManager() Returns the NodeTypeManager object through which available node types are discovered. There is one node type registry per repository, therefore the NodeTypeManager is not workspace-specific; it provides introspection methods for the global, repository-wide set of available node types. A RepositoryException is thrown if an error occurs.

The **NodeTypeManager** provides the following methods:

javax.jcr.nodetype. NodeTypeManager	
NodeType	getNodeType(String nodeName) Returns the NodeType specified by nodeName . If no node type by that name is registered, a NoSuchNodeTypeException is thrown. A RepositoryException is thrown if another error occurs.
NodeType Iterator	getAllNodeTypes() Returns all available node types, primary and mixin. A RepositoryException is thrown if an error occurs.
NodeType Iterator	getPrimaryNodeTypes() Returns all available primary node types. A RepositoryException is thrown if an error occurs.
NodeType Iterator	getMixinNodeTypes() Returns all available mixin types. If none are available, an empty iterator is returned. A RepositoryException is thrown if an error occurs.

6.7.10 Discovering the Node Types of a Node

Methods are provided for determining the node types of existing nodes:

javax.jcr. Node

<code>NodeType</code>	<p><code>getPrimaryNodeType()</code></p> <p>Returns the primary node type assigned to this node. Which <code>NodeType</code> is returned when this method is called on the root node of a workspace is up to the implementation, though the returned type must, of course, be consistent with the child nodes and properties of the root node.</p> <p>A <code>RepositoryException</code> is thrown if an error occurs.</p>
<code>NodeType[]</code>	<p><code>getMixinNodeTypes()</code></p> <p>Returns an array of <code>NodeType</code> objects representing the mixin node types assigned to this node. This includes only those mixin types explicitly assigned to this node, and therefore listed in the property <code>jcr:mixinTypes</code>. It does not include mixin types inherited through the addition of supertypes to the primary type hierarchy. See 6.7.22.2 Additions to the Hierarchy.</p> <p>A <code>RepositoryException</code> is thrown if an error occurs.</p>
<code>boolean</code>	<p><code>isNodeType(String nodeName)</code></p> <p>Returns true if this node is of the specified primary node type or mixin type, or a subtype thereof. Returns <code>false</code> otherwise.</p> <p>A <code>RepositoryException</code> is thrown if an error occurs.</p>

6.7.11 Discovering the Definition of a Node Type

The `NodeType` object represents a primary or mixin node type available in the repository.

<code>javax.jcr.nodetype.</code> <code>NodeType</code>	
<code>String</code>	<p><code>getName()</code></p> <p>Returns the name of the node type.</p>
<code>boolean</code>	<p><code>isMixin()</code></p> <p>Returns <code>true</code> if this is a mixin type; returns <code>false</code> if it is primary.</p>
<code>boolean</code>	<p><code>hasOrderableChildNodes()</code></p> <p>Returns <code>true</code> if nodes of this type must support orderable child nodes; returns <code>false</code> otherwise. If a node type returns <code>true</code> on a call to this method, then all nodes of that node type <i>must</i> support the method <code>Node.orderBefore</code>. If a node type returns <code>false</code> on a</p>

	<p>call to this method, then nodes of that node type <i>may</i> support these ordering methods. Only the primary node type of a node controls that node's status in this regard. This setting on a mixin node type will not have any effect on the node. See 7.1.10 Ordering Child Nodes.</p>
String	<p>getPrimaryItemName()</p> <p>Returns the name of the primary item (one of the child items of the node's of this node type). If this node has no primary item, then this method returns <code>null</code>. This indicator is used by the method <code>Node.getPrimaryItem()</code>. See 6.2.3 Node Read Methods.</p>
NodeType[]	<p>getSupertypes()</p> <p>Returns all supertypes of this node type in the node type inheritance hierarchy. For primary types apart from <code>nt:base</code>, this list will always include at least <code>nt:base</code>. For mixin types, there is no required base type.</p>
NodeType[]	<p>getDeclaredSupertypes()</p> <p>Returns the <i>direct</i> supertypes of this node type in the node type inheritance hierarchy, that is, those actually declared in this node type. In single-inheritance systems, this will always be an array of size 0 or 1. In systems that support multiple inheritance of node types this array may be of size greater than 1.</p>
boolean	<p>isNodeType(String nodeName)</p> <p>Returns <code>true</code> if this node type is <code>nodeName</code> or a subtype of <code>nodeName</code>, otherwise returns <code>false</code>.</p>
PropertyDef[]	<p>getPropertyDefs()</p> <p>Returns an array containing the property definitions of this node type. This includes both those property definitions actually declared in this node type and those inherited from the supertypes of this type.</p>
PropertyDef[]	<p>getDeclaredPropertyDefs()</p> <p>Returns an array containing the property definitions actually declared in this node type.</p>
NodeDef[]	<p>getChildNodeDefs()</p> <p>Returns an array containing the child node definitions of this node type. This includes both those child node definitions actually declared in this node type and those</p>

	inherited from the supertypes of this node type.
<code>NodeDef[]</code>	<code>getDeclaredChildNodeDefs()</code> Returns an array containing the child node definitions actually declared in this node type.
<code>boolean</code>	<code>canSetProperty(String propertyName, Value value)</code> Returns true if setting propertyName to value is allowed by this node type. Otherwise returns false .
<code>boolean</code>	<code>canSetProperty(String propertyName, Value[] values)</code> Returns true if setting propertyName to values is allowed by this node type. Otherwise returns false .
<code>boolean</code>	<code>canAddChildNode(String childNodeName)</code> Returns true if adding a child node called childNodeName is allowed by this node type. Returns false if it is not allowed or if allowing it depends on the node type of the child node and the system cannot unambiguously determine whether it is allowable without explicit node type information.
<code>boolean</code>	<code>canAddChildNode(String childNodeName, String nodeName)</code> Returns true if adding a child node called childNodeName of node type nodeName is allowed by this node type. The same as <code>canAddChildNode(String childNodeName)</code> except that the type of the child node is explicitly specified. If it is not allowed, false is returned.
<code>boolean</code>	<code>canRemoveItem(String itemName)</code> Returns true if removing the child item called itemName is allowed by this node type. Returns false otherwise.

6.7.12 ItemDef

The **ItemDef** is the super-interface of **PropertyDef** and **NodeDef**. It encapsulates the methods common to both.

```
javax.jcr.nodetype.  
ItemDef
```

NodeType	<p>getDeclaringNodeType()</p> <p>Gets the node type that contains the declaration of this ItemDef.</p>
String	<p>getName()</p> <p>Gets the name of the item to which this definition applies. If "*", then this ItemDef defines a <i>residual set</i> of child items. That is, it defines the characteristics of all those child items with names <i>apart from the names explicitly used in other item definitions</i>. See 6.7.15 Residual Definitions.</p>
boolean	<p>isAutoCreate()</p> <p>Reports whether the item is to be automatically created when its parent node is created. If true then this ItemDef will necessarily <i>not</i> be a residual set definition but will specify an actual item name (in other words getName() will not return "*"). See 6.7.15 Residual Definitions.</p>
boolean	<p>isMandatory()</p> <p>Reports whether the item is mandatory.</p> <p>A mandatory child node is one that, if its parent node exists, must also exist.</p> <p>A mandatory property is one that must have a value. In the case of single-value properties this means that it must exist (since there is no such thing a null value). In the case of multi-value properties this means that the property must exist and must have at least one value (it cannot hold an empty array).</p> <p>A mandatory item cannot be removed, short of removing its parent. Nor can it be set to the empty array (if it is a multi-value property).</p> <p>An attempt (in a level 2 implementation) to save a node that has a mandatory child item without first creating that child item and, if it is a property, giving it a value, will throw a ConstraintViolationException on save.</p>
int	<p>getOnParentVersion()</p> <p>Gets the OnParentVersion status of the property. This governs what occurs (in implementations that support versioning) when the parent node of this item is checked-in. See 8.2 Versioning.</p>

boolean	isProtected() Reports whether the child item is protected. In level 2 implementations, a protected item is one that cannot be removed (except by removing its parent) or modified through the standard write methods of this API (that is, Item.remove , Node.addNode , Node.setProperty and Property.setValue). A protected node may be removed or modified (in a level 2 implementation), however, through some mechanism not defined by this specification or as a side-effect of operations other than the standard write methods of the API. For example, in those repositories that support versioning, the Node.checkin method has the side-effect of changing a node's jcr:isCheckedOut property, even though that property is protected. Note that when a node is protected this means that all its properties are also protected (regardless of their protected setting). The protected status of a property only becomes relevant if its parent node is not protected.
---------	--

6.7.13 PropertyDef

The **PropertyDef** represents a property definition. It inherits all the method of **ItemDef** and adds the following:

javax.jcr.nodetype. PropertyDef extends ItemDef	
int	getRequiredType() Gets the required type of the property. One of STRING , BINARY , DATE , LONG , DOUBLE , NAME , PATH , REFERENCE , BOOLEAN or UNDEFINED . See 6.2.5 Property Types . If UNDEFINED , then this property may be of any type.
String[]	getValueConstraints() Gets the array of constraint strings. This array of strings describes the constraints that exist on values of the property. Reporting of value constraints is <i>optional</i> . An implementation may return null , indicating that value constraint information is unavailable (though a constraint may still exist). Note that to indicate a null value for this attribute in a node type definition that is stored in content, the jcr:valueConstraints property is simply removed (since null values for properties are not allowed, see 6.7.20 Storage of Node Type Definitions in

	<p><i>Content</i>).</p> <p>Returning an empty array, on the other hand, indicates that constraint information is available and that no constraints are placed on the value of the property.</p> <p>If a non-empty array is returned then it is interpreted as a disjunctive set of constraints (i.e. the value must meet at least one of the constraints). The interpretation of the constraint strings themselves differs according to the type of the property. See 6.7.16 Value Constraints for details.</p>
Value[]	<p>getDefaultValues()</p> <p>Gets the default value(s) of the property. These are the values (or value) that the property defined by this PropertyDef will be assigned if it is automatically created (that is, if isAutoCreate() returns true).</p> <p>This method returns an array of Value objects. If the property is multi-valued, then this array represents the full set of values that the property will be assigned upon being auto-created. Note that this could be the empty array. If the property is single-valued, then the array returned will be of size 1.</p> <p>If null is returned, then the property has no fixed default value. This does not exclude the possibility that the property still assumes some value automatically, but that value may be parameterized (for example, "the current date") and hence not expressible as a single fixed value. In particular, this <i>must</i> be the case if isAutoCreate returns true and this method returns null.</p> <p>Note that to indicate a null value for this attribute in a node type definition that is stored in content, the jcr:defaultValues property is simply removed (since null values for properties are not allowed, see 6.7.20 Storage of Node Type Definitions in Content).</p>
boolean	<p>isMultiple()</p> <p>Reports whether this property can have multiple values. Note that the isMultiple flag is special in that a given node type may have two property definitions that are identical in every respect except for their isMultiple status. For example, a node type can specify two string properties both called x, one of which is multi-valued and the other not. An example of such a node type is nt:unstructured (see 6.7.22.4 nt:unstructured).</p>

6.7.14 NodeDef

The **NodeDef** represents a child node definition. It inherits all the methods of **ItemDef** and adds the following:

javax.jcr.nodetype. NodeDef extends ItemDef	
NodeType[]	getRequiredPrimaryTypes() Gets the minimum set of primary node types that the child node must have. Returns an array to support those implementations with multiple inheritance. This method never returns an empty array. If this node definition places no requirements on the primary node type, then this method will return an array containing only the NodeType object representing nt:base , which is the base of all primary node types and therefore constitutes the least restrictive node type requirement. Note that a node must still have only one assigned node type, though this attribute can restrict that node type by taking advantage of any inheritance hierarchy that the implementation may support.
NodeType	getDefaultPrimaryType() Gets the default node type that will be assigned to the child node if it is created without an explicitly specified node type. This node type must be a subclass of (or the same class as) the node type(s) returned by getRequiredPrimaryTypes . 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 throw a ConstraintViolationException . Note that to indicate a null value for this attribute in a node type definition that is stored in content, the jcr:defaultPrimaryType property is simply removed (since null values for properties are not allowed, see 6.7.20 Storage of Node Type Definitions in Content).
boolean	allowSameNameSibs() Reports whether this child node can have same-name siblings. In other words, whether the parent node can have more than one child node of this name.

6.7.15 Residual Definitions

When the **name** attribute (i.e., that returned by **getName()**) of a **PropertyDef** or **NodeDef** is "*", this indicates that the definition is a *residual definition*.

A residual definition defines the characteristics of all properties (if it is a `PropertyDef`) or child nodes (if it is a `NodeDef`) *apart than those explicitly named in other property or node definitions*.

It is possible for a node type to have more than one residual definition. This means that all properties and child nodes other than those explicitly named must conform to *at least one* of the residual definitions.

6.7.16 Value Constraints

Each string in the array returned by `PropertyDef.getValueConstraints()` specifies a constraint on the value(s) of the property. The constraints are OR-ed together, meaning that in order to be valid, the value (each of the values, in the case of multi-value properties) must meet *at least one* of the constraints. For example, a constraint array of `["constraint1", "constraint2", "constraint3"]` has the interpretation: "the value of this property must meet either constraint1, constraint2 or constraint3".

Reporting constraint information is optional. Therefore, the return of an empty array indicates that there are no *discoverable* constraints, meaning that either there are constraints but they are inexpressible in the constraint-string syntax, or constraint discovery is simply not supported.

In the case of multi-value properties, the constraint array returned applies independently to each of the values of the property. For example, if one value meets one constraint in the constraint array while the other meets another, the constraint set is considered met for the property as a whole.

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 by default since, by definition, no values have failed to meet the constraints.

The constraint strings themselves have different formats and interpretations depending on the type of the property in question. The following describes the value constraint syntax for each property type:

- **STRING:** The constraint string is a regular expression pattern. For example the regular expression `".*"` means "any string, including the empty string". Whereas a simple literal string (without any RE-specific meta-characters) like `"banana"` matches only the string `"banana"` (see [6.7.16.1 Choice Lists](#), below).

- **PATH:** The constraint string is a *JCR path* with an optional "*" character after the last "/" character. For example, possible constraint strings for a property of type **PATH** include:

1. `"/myapp:products/myapp:televisions"`
2. `"/myapp:products/myapp:televisions/"`
3. `"/myapp:products/*"`
4. `"myapp:products/myapp:televisions"`
5. `"../myapp:televisions"`
6. `"../myapp:televisions/*"`

The following principles apply:

- The constraint must match the *normalized* path. For example, the "*" means "matches descendants" not "matches any subsequent path", so that `/a/*` does not match `/a/..c`. Similarly, a trailing "/" has no effect (hence, 1 and 2, above, are equivalent).
- Relative path constraints only match relative path values and absolute path constraints only match absolute path values.
- The trailing "*" character means that the value of the **PATH** property is restricted to the indicated subtree (in other words any additional relative path can replace the "*"). For example, 3, above would allow `/myapp:products/myapp:radios,`
`/myapp:products/myapp:microwaves/X900,` and so forth.
- A constraint without a "*" means that the **PATH** property is restricted to that precise path. For example, 1, above would allow only the value `/myapp:products/myapp:televisions.`
- The constraint can indicate either a relative path or an absolute path depending on whether it includes a leading "/" character. 1 and 4 above, for example, are distinct.
- The string returned must reflect the namespace mapping in the current **session** (i.e., the current state of the namespace registry overlaid with any session-specific mappings). Constraint strings for **PATH** properties should be stored in fully-qualified form (using the actual URI instead of the prefix) and then be converted to prefix form according to the current mapping. Note however that these constraint strings are not themselves valid **PATH** values, since they may contain a "*" character,

which is not allowed in the value of an actual **PATH** property.

- **NAME:** The constraint string is a *JCR name* in prefix form. For example, `"myapp:products"`. No wildcards or other pattern matching are supported. As with **PATH** properties, the string returned must reflect the namespace mapping in the current **session**. Constraint strings for **NAME** properties should be stored in fully-qualified form (using the actual URI instead of the prefix) and then be converted to prefix form according to the current mapping.
- **REFERENCE:** The constraint string is a *JCR name* in prefix form. This name is interpreted as a node type name and the **REFERENCE** property is restricted to referring only to nodes that have at least the indicated node type⁶. For example, a constraint of `"mytype:document"` would indicate that the **REFERENCE** property in question can only refer to nodes that have at least the node type `mytype:document` (assuming this was the only constraint returned in the array, recall that the array of constraints are to be "OR-ed" together). No wildcards or other pattern matching are supported. As with **PATH** properties, the string returned must reflect the namespace mapping in the current **session**. Constraint strings for **REFERENCE** properties should be stored in fully-qualified form (using the actual URI instead of the prefix) and then be converted to prefix form according to the current mapping.
- **BOOLEAN:** Either `"true"` or `"false"`.

The remaining types all have value constraints in the form of inclusive or exclusive ranges: i.e., `"[min, max]"`, `"(min, max)"`, `"(min, max]"` or `"[min, max)"`. Where `"["` and `"]"` indicate "inclusive", while `"("` and `")"` indicate "exclusive". A missing **min** or **max** value indicates no bound in that direction. For example `[, 5]` means no minimum but a maximum of 5 (inclusive) while `[,]` means simply that any value will suffice. 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.

⁶ This is a minimal requirement. The referenced node may have additional mixin node types other than that indicated (and in fact, by definition, it must have at least `mix:referenceable`, for example). In addition it may be of a node type that is a subtype of the type indicated by the constraint.

- **DATE:** *min* and *max* are dates specifying the allowed date range. The date strings must be in the ISO 8601:2000-compliant format: *sYYYY-MM-DDThh:mm:ss.sssTZD*. See [6.2.5.1 Date](#).
- **LONG, DOUBLE:** *min* and *max* are numbers.

6.7.16.1 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 indication the set of permitted values.

6.7.17 Automatic Item Creation

The ability to specify the automatic creation of child nodes and properties has a number of interesting repercussions. Consider a situation where we have three node types, **C**, **B** and **A**:

- **C** specifies an auto-created **STRING** property called **z** with default value “hello”.
- **B** specifies an auto-created child node **y** of node type **C**.
- **A** specifies an auto-created child node called **x** of node type **B**.

Therefore, when a node **N** of node type **A** is added, this triggers a chain of automatic node creation resulting in a structure like this:

N--> X--> Y--> Z="hello"

It is perfectly possible for a repository to have node types that may result in a cascade of item creation. However, *it must never be the case that a repository has a set of node types that may result in an infinite loop of automatic item creation.*

6.7.18 Discovery of Constraints on Existing Items

The **Node** and **Property** interfaces also have methods that allow direct access to the **NodeDef** or **PropertyDef** applicable to a particular node or property.

javax.jcr. Node	
NodeDef	getDefinition() Returns the node definition that applies to this Node . In some cases there may appear to be more than one definition that could apply to this node. However, it is assumed that upon creation of this node, a single particular definition was used and it is <i>that</i> definition that

	<p>this method returns. How this governing definition is selected upon node creation from among others which may have been applicable is an implementation issue and is not covered by this specification. The NodeDef returned when this method is called on the root node of a workspace is also up to the implementation.</p> <p>Throws a RepositoryException if an error occurs.</p>
--	--

javax.jcr. Property	
PropertyDef	getDefinition() <p>Returns the property definition that applies to this Property. In some cases there may appear to be more than one definition that could apply to this property. However, it is assumed that upon creation of this property, a single particular definition was used and it is <i>that</i> definition that this method returns. How this governing definition is selected upon property creation from among others which may have been applicable is an implementation issue and is not covered by this specification.</p> <p>Throws a RepositoryException if an error occurs.</p>

6.7.19 Predefined Node Types

Every repository must support at least the primary node type **nt:base**. All other primary node types must be subtypes of **nt:base**. A number of predefined primary node types are defined for common application domains.

In general, support for these additional primary node types is optional. However, the type **nt:unstructured** is required for XML import (see [7.3.2 Import from Document View](#)), which is a feature of level 2. As a result, support for **nt:unstructured** is, in effect, a requirement for level 2 compliance.

Three mixin node types **mix:referenceable**, **mix:versionable** and **mix:lockable** are defined. In general support for these types is also optional. However:

- **mix:referenceable** is required in order to support UUID-bearing nodes, which in turn support **REFERENCE** property types and versioning.
- Additionally, versioning requires the mixin node type **mix:versionable** and the primary node types **nt:version**,

`nt:versionHistory`, `nt:versionLabels`,
`nt:versionedChild` and `nt:frozenNode`. See [8.2](#)
Versioning.

- Locking requires the mixin node type `mix:lockable`. See [8.5](#) *Locking*.

6.7.19.1 Node Type Definition Notation

The following sections give the definition of each predefined node type and a short description and explanation for each. The node type definitions are in the following format:

```

NodeTypeName
...
Supertypes
...
IsMixin
...
HasOrderableChildNodes
...
PrimaryItemName
...
ChildNodeDef
  Name ...
  RequiredPrimaryTypes ...
  DefaultPrimaryType ...
  AutoCreate ...
  Mandatory ...
  OnParentVersion ...
  Protected ...
  SameNameSibs ...
.
. (more ChildNodeDefs)
.
PropertyDef
  Name ...
  RequiredType ...
  ValueConstraints ...
  DefaultValues ...
  AutoCreate ...
  Mandatory ...
  OnParentVersion ...
  Protected ...
  Multiple ...
.
. (more PropertyDefs)
.
```

6.7.20 Storage of Node Type Definitions in Content

It is *optional* for the repository to store the definitions of its available node types in content itself. However, if it does store these definitions then it should store them using the built-in node type `nt:nodeType` (and its associated node types `nt:propertyDef` and `nt:childNodeDef`). These node types are defined to store node type definitions themselves. For example, to store a *PropertyDef* a node of type `nt:propertyDef` is used. It has properties for each of

the attributes: the *Name* is stored in the property `jcr:name`, the *RequiredType* in `jcr:requiredType` and so on.

The attributes that make up a node type definition may in some cases have no set value. For example, some *ChildNodeDefs* may not define a *DefaultPrimaryType* (this amounts to stating that when such a child node is created by the client the client *must* provide a valid node type, otherwise an exception will be thrown; no node type will automatically be assumed).

In order to store this information (i.e., the lack of a value) in a `nt:nodeType`, `nt:childNodeDef` or `nt:propertyDef` node the property representing that attribute must simply be not present, since null values for properties are not allowed in JCR (see [4.7.3 No Null Values](#)).

However, to indicate this state in the node type definitions that follow we do use the value `null`, even though in an in-content representation of the node type this would be represented by the absence of the property in question. For example, in the definition of the node type `nt:file`,

```
NodeTypeDefinition
  nt:file
Supertypes
  nt:hierarchyNode
IsMixin
  false
HasOrderableChildNodes
  false
PrimaryItemName
  jcr:content
ChildNodeDefinition
  Name jcr:content
  RequiredPrimaryTypes [nt:base]
  DefaultPrimaryType null
  AutoCreate false
  Mandatory true
  OnParentVersion COPY
  Protected false
  SameNameSibs false
```

the child node `jcr:content` must exist (hence `Mandatory` is `true`, but it must be added by the client, not automatically (hence `AutoCreate` is `false`) and, when created, the client must provide the node type (hence *DefaultPrimaryType* is `null`). In an in-content storage of this node type however, the `null` status of *DefaultPrimaryType* would be represented by the absence of the property `jcr:defaultPrimaryType`.

Note that the *PrimaryItemName* indicator in `nt:nodeType` works similarly, if there is no primary item specified then the `jcr:primaryItemName` property of the `nt:nodeType` node is simply missing. But in the notation used here, this is indicated by specifying a `null`.

Again, similarly, to indicate that a property or child node definition is residual, the value returned by `ItemDef.getName()` is `"*"`. However, `"*"` is not a valid value for the property `jcr:name` in a `nt:propertyDef` or `nt:nodeDef` 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. In the notation below, however, the indicator `"*"` is still used.

6.7.21 Predefined Mixin Node Types

The three predefined mixin types are `mix:referenceable`, `mix:versionable` and `mix:lockable`. `mix:versionable` is a subtype of `mix:referenceable`. `mix:referenceable` and `mix:lockable` have no supertypes. There is no required base supertype among mixin types.

6.7.21.1 `mix:lockable`

```

NodeTypeName
  mix:lockable
Supertypes
  []
IsMixin
  true
HasOrderableChildNodes
  false
PrimaryItemName
  null
PropertyDef
  Name jcr:lockOwner
  RequiredType STRING
  ValueConstraints []
  DefaultValues null
  AutoCreate false
  Mandatory false
  OnParentVersion IGNORE
  Protected true
  Multiple false
PropertyDef
  Name jcr:lockIsDeep
  RequiredType BOOLEAN
  ValueConstraints []
  DefaultValues null
  AutoCreate false
  Mandatory false
  OnParentVersion IGNORE
  Protected true
  Multiple false

```

This node type is optional.

Only nodes with mixin node type `mix:lockable` may hold locks. See [8.5 Locking](#).

6.7.21.2 `mix:referenceable`

```

NodeTypeName

```

```

mix:referenceable
Supertypes
[]
IsMixin
true
HasOrderableChildNodes
false
PrimaryItemName
null
PropertyDef
  Name jcr:uuid
  RequiredType STRING
  ValueConstraints []
  DefaultValues null
  AutoCreate true
  Mandatory true
  OnParentVersion INITIALIZE
  Protected true
  Multiple false

```

This node type is optional.

This node type specifies an auto-created, mandatory, **STRING** property called **jcr:uuid**. This property is set automatically by the implementation when the **mix:referenceable** node is created or when this mixin type is added to an existing node.

A node must be **mix:referenceable** in order to be the target of a **REFERENCE** property.

6.7.21.3 **mix:versionable**

```

NodeTypeName
mix:versionable
Supertypes
mix:referenceable
IsMixin
true
HasOrderableChildNodes
false
PrimaryItemName
null
PropertyDef
  Name jcr:versionHistory
  RequiredType REFERENCE
  ValueConstraints ["nt:versionHistory"]
  DefaultValues null
  AutoCreate false
  Mandatory true
  OnParentVersion COPY
  Protected true
  Multiple false
PropertyDef
  Name jcr:baseVersion
  RequiredType REFERENCE
  ValueConstraints ["nt:version"]
  DefaultValues null
  AutoCreate false
  Mandatory true
  OnParentVersion IGNORE

```



```

    Protected true
    Multiple false
PropertyDef
    Name jcr:isCheckedOut
    RequiredType BOOLEAN
    ValueConstraints []
    DefaultValues [true]
    AutoCreate true
    Mandatory true
    OnParentVersion IGNORE
    Protected true
    Multiple false
PropertyDef
    Name jcr:predecessors
    RequiredType REFERENCE
    ValueConstraints [nt:version]
    DefaultValues null
    AutoCreate false
    Mandatory true
    OnParentVersion COPY
    Protected true
    Multiple true
PropertyDef
    Name jcr:mergeFailed
    RequiredType REFERENCE
    ValueConstraints []
    DefaultValues null
    AutoCreate false
    Mandatory false
    OnParentVersion ABORT
    Protected true
    Multiple true

```

This node type is optional.

This mixin node type supports the versioning system. For a node to be versionable, it must be of this mixin node type. See [8.2 Versioning](#) for details.

6.7.22 Predefined Primary Node Types

The primary node types described here are optional, except for **nt:base**, which is required. Every node in the repository must be of at least this type. Any custom, implementation-specific primary node types must be subtypes of **nt:base**.

nt:version and **nt:versionHistory** are required for versioning support.

nt:nodeType, **nt:propertyDef** and **nt:childNodeDef** are required if storage of node type definitions in the repository content itself is supported.

6.7.22.1 Node Type Inheritance Hierarchy

The node type names below are arranged in a hierarchy showing their inheritance structure.

```
nt:base
|
|--nt:unstructured
|
|--nt:hierarchyNode
|   |
|   |--nt:file
|   |
|   |--nt:linkedFile
|   |
|   /
|   |--nt:folder
|
|--nt:nodeType
|
|--nt:propertyDef
|
|--nt:childNodeDef
|
|--nt:versionHistory*
|
|--nt:versionLabels
|
|--nt:version*
|
|--nt:frozenNode
|
|--nt:versionedChild
|
|--nt:query
|
|--nt:resource*
```

* these node types also have **mix:referenceable** as a supertype.

6.7.22.2 Additions to the Hierarchy

An implementation may extend the definition of any predefined node type by adding supertypes to those defined in this specification. These additional supertypes may be either predefined 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**.

The hierarchy above and the definitions below, therefore, reflect the *minimal* set of supertypes for each predefined node type.

Note that this extension mechanism is distinct from the automatic addition of mixin types that may be done on node creation in level

2 (see [7.4.4 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.

6.7.22.3 nt:base

```
NodeTypeName
  nt:base
Supertypes
  []
IsMixin
  false
HasOrderableChildNodes
  false
PrimaryItemName
  null
PropertyDef
  Name jcr:primaryType
  RequiredType NAME
  ValueConstraints []
  DefaultValues null
  AutoCreate true
  Mandatory true
  OnParentVersion COMPUTE
  Protected true
  Multiple false
PropertyDef
  Name jcr:mixinTypes
  RequiredType NAME
  ValueConstraints []
  DefaultValues null
  AutoCreate false
  Mandatory false
  OnParentVersion COMPUTE
  Protected true
  Multiple true
```

This node type is required.

All node types inherit from **nt:base**. As its name suggests it is the base type for all other types, and hence has no supertypes.

This node type specifies the special properties **jcr:primaryType**, and **jcr:mixinTypes**.

The **jcr:primaryType** property is a **NAME** property holding the name of the primary node type of its node. This property is mandatory.

The **jcr:mixinTypes** is a multi-value **NAME** property that holds the names of the node's assigned mixin node types, if any. This property may not exist if the node in question has no mixin types assigned.

Since this information is itself stored as content, it will be serialized and deserialized along with all other content. This allows the

preservation of node type information across serialization/deserialization cycles. See [7.4.5 Serialization and Node Types](#).

These properties are protected and are therefore maintained entirely by the repository itself. A client of the API can read the properties but cannot alter them.

6.7.22.4 nt:unstructured

```
NodeTypeName
  nt:unstructured
Supertypes
  nt:base
IsMixin
  false
HasOrderableChildNodes
  true
PrimaryItemName
  null
ChildNodeDef
  Name "*"
  RequiredPrimaryTypes [nt:base]
  DefaultPrimaryType nt:unstructured
  AutoCreate false
  Mandatory false
  OnParentVersion VERSION
  Protected false
  SameNameSibs true
PropertyDef
  Name "*"
  RequiredType UNDEFINED
  ValueConstraints []
  DefaultValues null
  AutoCreate false
  Mandatory false
  OnParentVersion COPY
  Protected false
  Multiple true
PropertyDef
  Name "*"
  RequiredType UNDEFINED
  ValueConstraints []
  DefaultValues null
  AutoCreate false
  Mandatory false
  OnParentVersion COPY
  Protected false
  Multiple false
```

This node type is required for level 2 compliance. In level 1 it is optional.

This is the most flexible node type. 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 supports client-orderable child nodes.

Like all node types, it inherits the special `jcr:primaryType` and `jcr:mixinTypes` properties from `nt:base`.

6.7.22.5 `nt:hierarchyNode`

```
NodeType Name
  nt:hierarchyNode
Supertypes
  nt:base
IsMixin
  false
HasOrderableChildNodes
  false
PrimaryItemName
  null
PropertyDef
  Name jcr:created
  RequiredType DATE
  ValueConstraints []
  DefaultValues null
  AutoCreate true
  Mandatory true
  OnParentVersion INITIALIZE
  Protected true
  Multiple false
```

This node type is optional.

This node type serves primarily as the supertype of `nt:file` and `nt:folder`. It defines one property inherited by these node types: `jcr:created`.

6.7.22.6 `nt:file`

```
NodeType Name
  nt:file
Supertypes
  nt:hierarchyNode
IsMixin
  false
HasOrderableChildNodes
  false
PrimaryItemName
  jcr:content
ChildNodeDef
  Name jcr:content
  RequiredPrimaryTypes [nt:base]
  DefaultPrimaryType null
  AutoCreate false
  Mandatory true
  OnParentVersion COPY
  Protected false
  SameNameSibs false
```

This node type is optional.

Nodes of this node type may be used to represent files. This node type inherits the child nodes and properties 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 client. A common approach is to make the `jcr:content` a node of type `nt:resource`.

The strategy in separating the `nt:file` node from its `jcr:content` child node is to divide hierarchy from content. The idea is to provide a common indicator that indicates a cut off point below which the nodes and properties have a different semantic interpretation than they do above. This type of division is common to many hierarchical information structures, such as file systems.

The `jcr:content` child node is also designated as the primary child item of its parent.

6.7.22.7 `nt:linkedFile`

```

NodeTypeName
  nt:linkedFile
Supertypes
  nt:hierarchyNode
IsMixin
  false
HasOrderableChildNodes
  false
PrimaryItemName
  jcr:content
PropertyDef
  Name jcr:content
  RequiredType REFERENCE
  ValueConstraints []
  DefaultValues null
  AutoCreate false
  Mandatory true
  OnParentVersion COPY
  Protected false
  Multiple false

```

This node type is optional.

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. In other words the content node can reside anywhere in the repository. In addition, because of the extra level of indirection, the same content node can be referenced by multiple `nt:linkedFile` nodes. This feature can be used, for example, to present multiple orthogonal hierarchical views of the same content.

6.7.22.8 `nt:folder`

```

NodeTypeName

```

```

nt:folder
Supertypes
nt:hierarchyNode
IsMixin
false
HasOrderableChildNodes
false
PrimaryItemName
null
ChildNodeDef
Name ""
RequiredPrimaryTypes [nt:hierarchyNode]
DefaultPrimaryType null
AutoCreate false
Mandatory false
OnParentVersion VERSION
Protected false
SameNameSibs false

```

This node type is optional.

Nodes of this node type can be used to represent folders. This node type inherits the child nodes and properties of **nt:hierarchyNode** and adds the ability to have any number of other **nt:hierarchyNode** child nodes. This means, in particular, that it can have child nodes of **nt:file** or **nt:folder**. In this way, it is analogous to a folder in a conventional file system.

6.7.22.9 **nt:resource**

```

NodeTypeName
nt:resource
Supertypes
nt:base
mix:referenceable
IsMixin
false
HasOrderableChildNodes
false
PrimaryItemName
jcr:data
PropertyDef
Name jcr:encoding
RequiredType STRING
ValueConstraints []
DefaultValues null
AutoCreate false
Mandatory true
OnParentVersion COPY
Protected false
Multiple false
PropertyDef
Name jcr:mimeType
RequiredType STRING
ValueConstraints []
DefaultValues null
AutoCreate false
Mandatory true
OnParentVersion COPY
Protected false

```

```

    Multiple false
PropertyDef
  Name jcr:data
  RequiredType BINARY
  ValueConstraints []
  DefaultValues null
  AutoCreate false
  Mandatory true
  OnParentVersion COPY
  Protected false
  Multiple false
PropertyDef
  Name jcr:lastModified
  RequiredType DATE
  ValueConstraints []
  DefaultValues null
  AutoCreate false
  Mandatory true
  OnParentVersion IGNORE
  Protected false
  Multiple false

```

This node type is optional.

This node type can 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**.

6.7.22.10 **nt:nodeType**

```

NodeType
  Name nt:nodeType
Supertypes
  nt:base
IsMixin
  false
HasOrderableChildNodes
  false
PrimaryItemName
  null
PropertyDef
  Name jcr:nodeTypeName
  RequiredType NAME
  ValueConstraints []
  DefaultValues null
  AutoCreate false
  Mandatory true
  OnParentVersion COPY
  Protected false
  Multiple false
PropertyDef
  Name jcr:supertypes
  RequiredType NAME
  ValueConstraints []
  DefaultValues null
  AutoCreate false
  Mandatory true
  OnParentVersion COPY
  Protected false
  Multiple true

```



```

PropertyDef
  Name jcr:isMixin
  RequiredType BOOLEAN
  ValueConstraints []
  DefaultValues null
  AutoCreate false
  Mandatory true
  OnParentVersion COPY
  Protected false
  Multiple false
PropertyDef
  Name jcr:hasOrderableChildNodes
  RequiredType BOOLEAN
  ValueConstraints []
  DefaultValues null
  AutoCreate false
  Mandatory true
  OnParentVersion COPY
  Protected false
  Multiple false
PropertyDef
  Name jcr:primaryItemName
  RequiredType NAME
  ValueConstraints []
  DefaultValues null
  AutoCreate false
  Mandatory false
  OnParentVersion COPY
  Protected false
  Multiple false
ChildNodeDef
  Name jcr:propertyDef
  RequiredPrimaryTypes [nt:propertyDef]
  DefaultPrimaryType nt:propertyDef
  AutoCreate false
  Mandatory false
  OnParentVersion VERSION
  Protected false
  SameNameSibs true
ChildNodeDef
  Name jcr:childNodeDef
  RequiredPrimaryTypes [nt:childNodeDef]
  DefaultPrimaryType nt:childNodeDef
  AutoCreate false
  Mandatory false
  OnParentVersion VERSION
  Protected false
  SameNameSibs true

```

This node type is optional.

This is the node type for the nodes that store node type definitions themselves (see [6.7.20 Storage of Node Type Definitions](#)).

6.7.22.11 **nt:propertyDef**

```

NodeTypeNames
  nt:propertyDef
Supertypes
  nt:base
IsMixin

```

```

    false
HasOrderableChildNodes
    false
PrimaryItemName
    null
PropertyDef
    Name jcr:name
    RequiredType NAME
    ValueConstraints []
    DefaultValues null
    AutoCreate false
    Mandatory false
    OnParentVersion COPY
    Protected false
    Multiple false
PropertyDef
    Name jcr:autoCreate
    RequiredType BOOLEAN
    ValueConstraints []
    DefaultValues null
    AutoCreate false
    Mandatory true
    OnParentVersion COPY
    Protected false
    Multiple false
PropertyDef
    Name jcr:mandatory
    RequiredType BOOLEAN
    ValueConstraints []
    DefaultValues null
    AutoCreate false
    Mandatory true
    OnParentVersion COPY
    Protected false
    Multiple false
PropertyDef
    Name jcr:onParentVersion
    RequiredType STRING
    ValueConstraints []
    DefaultValues null
    AutoCreate false
    Mandatory true
    OnParentVersion COPY
    Protected false
    Multiple false
PropertyDef
    Name jcr:protected
    RequiredType BOOLEAN
    ValueConstraints []
    DefaultValues null
    AutoCreate false
    Mandatory true
    OnParentVersion COPY
    Protected false
    Multiple false
PropertyDef
    Name jcr:requiredType
    RequiredType STRING
    ValueConstraints []
    DefaultValues null
    AutoCreate false

```

```

Mandatory true
OnParentVersion COPY
Protected false
Multiple false
PropertyDef
Name jcr:valueConstraints
RequiredType STRING
ValueConstraints []
DefaultValues null
AutoCreate false
Mandatory false
OnParentVersion COPY
Protected false
Multiple true
PropertyDef
Name jcr:defaultValues
RequiredType UNDEFINED
ValueConstraints []
DefaultValues null
AutoCreate false
Mandatory false
OnParentVersion COPY
Protected false
Multiple true
PropertyDef
Name jcr:multiple
RequiredType BOOLEAN
ValueConstraints []
DefaultValues null
AutoCreate false
Mandatory true
OnParentVersion COPY
Protected false
Multiple false

```

This node type is optional.

A node type used in conjunction with **nt:nodeType** for storing node type definitions themselves. See also **nt:childNodeDef**.

Note that in order to represent a residual property definition (see [6.7.15 Residual Definitions](#)) the property **jcr:name** must not be present in the **nt:propertyDef** node.

6.7.22.12 **nt:childNodeDef**

```

NodeTypeNames
nt:childNodeDef
Supertypes
nt:base
IsMixin
false
HasOrderableChildNodes
false
PrimaryItemName
null
PropertyDef
Name jcr:name
RequiredType NAME
ValueConstraints []

```

```

    DefaultValues null
    AutoCreate false
    Mandatory false
    OnParentVersion COPY
    Protected false
    Multiple false
PropertyDef
    Name jcr:autoCreate
    RequiredType BOOLEAN
    ValueConstraints []
    DefaultValues null
    AutoCreate false
    Mandatory true
    OnParentVersion COPY
    Protected false
    Multiple false
PropertyDef
    Name jcr:mandatory
    RequiredType BOOLEAN
    ValueConstraints []
    DefaultValues null
    AutoCreate false
    Mandatory true
    OnParentVersion COPY
    Protected false
    Multiple false
PropertyDef
    Name jcr:onParentVersion
    RequiredType STRING
    ValueConstraints []
    DefaultValues null
    AutoCreate false
    Mandatory true
    OnParentVersion COPY
    Protected false
    Multiple false
PropertyDef
    Name jcr:protected
    RequiredType BOOLEAN
    ValueConstraints []
    DefaultValues null
    AutoCreate false
    Mandatory true
    OnParentVersion COPY
    Protected false
    Multiple false
PropertyDef
    Name jcr:requiredPrimaryTypes
    RequiredType NAME
    ValueConstraints []
    DefaultValues [nt:base]
    AutoCreate false
    Mandatory true
    OnParentVersion COPY
    Protected false
    Multiple true
PropertyDef
    Name jcr:defaultPrimaryType
    RequiredType NAME
    ValueConstraints []
    DefaultValues null

```

```

    AutoCreate false
    Mandatory false
    OnParentVersion COPY
    Protected false
    Multiple false
PropertyDef
    Name jcr:sameNameSibs
    RequiredType BOOLEAN
    ValueConstraints []
    DefaultValues null
    AutoCreate false
    Mandatory true
    OnParentVersion COPY
    Protected false
    Multiple false

```

This node type is optional.

A node type used in conjunction with `nt:nodeType` for storing node type definitions themselves. See also `nt:propertyDef`.

6.7.22.13 `nt:versionHistory`

```

NodeTypeName
    nt:versionHistory
Supertypes
    nt:base
    mix:referenceable
IsMixin
    false
HasOrderableChildNodes
    false
PrimaryItemName
    null
PropertyDef
    Name jcr:versionableUuid
    RequiredType STRING
    ValueConstraints []
    DefaultValues null
    AutoCreate true
    Mandatory true
    OnParentVersion ABORT
    Protected true
    Multiple false
ChildNodeDef
    Name jcr:rootVersion
    RequiredPrimaryTypes [nt:version]
    DefaultPrimaryType nt:version
    AutoCreate true
    Mandatory true
    OnParentVersion ABORT
    Protected true
    SameNameSibs false
ChildNodeDef
    Name jcr:versionLabels
    RequiredPrimaryTypes [nt:versionLabels]
    DefaultPrimaryType nt:versionLabels
    AutoCreate true
    Mandatory true
    OnParentVersion ABORT
    Protected true

```

```

    SameNameSibs false
ChildNodeDef
    Name ""
    RequiredPrimaryTypes [nt:version]
    DefaultPrimaryType nt:version
    AutoCreate false
    Mandatory false
    OnParentVersion ABORT
    Protected true
    SameNameSibs false

```

This node type is optional.

This node type is used in the versioning system. It is required in those implementations that support versioning. See [8.2 Versioning](#), for more details.

6.7.22.14 **nt:versionLabels**

```

NodeTypeName
    nt:versionLabels
Supertypes
    nt:base
IsMixin
    false
HasOrderableChildNodes
    false
PrimaryItemName
    null
PropertyDef
    Name *
    RequiredType REFERENCE
    ValueConstraints ["nt:version"]
    DefaultValues null
    AutoCreate false
    Mandatory false
    OnParentVersion ABORT
    Protected true
    Multiple false

```

This node type is optional.

This node type is used in the versioning system. It is required in those implementations that support versioning. See [8.2 Versioning](#), for more details.

6.7.22.15 **nt:version**

```

NodeTypeName
    nt:version
Supertypes
    nt:base
    mix:referenceable
IsMixin
    false
HasOrderableChildNodes
    false
PrimaryItemName
    null

```

```

PropertyDef
  Name jcr:created
  RequiredType DATE
  ValueConstraints []
  DefaultValues null
  AutoCreate true
  Mandatory true
  OnParentVersion ABORT
  Protected true
  Multiple false
PropertyDef
  Name jcr:predecessors
  RequiredType REFERENCE
  ValueConstraints ["nt:version"]
  DefaultValues null
  AutoCreate false
  Mandatory false
  OnParentVersion ABORT
  Protected true
  Multiple true
PropertyDef
  Name jcr:successors
  RequiredType REFERENCE
  ValueConstraints ["nt:version"]
  DefaultValues null
  AutoCreate false
  Mandatory false
  OnParentVersion ABORT
  Protected true
  Multiple true
ChildNodeDef
  Name jcr:frozenNode
  RequiredPrimaryTypes [nt:frozenNode]
  DefaultPrimaryType null
  AutoCreate false
  Mandatory false
  OnParentVersion ABORT
  Protected true
  SameNameSibs false

```

This node type is optional.

This node type is used in the versioning system. It is required in those implementations that support versioning. See [8.2 Versioning](#), for more details.

6.7.22.16 nt:frozenNode

```

NodeTypeName
  nt:frozenNode
Supertypes
  nt:base
  mix:referenceable
IsMixin
  false
HasOrderableChildNodes
  true
PrimaryItemName
  null
PropertyDef

```

```

    Name jcr:frozenPrimaryType
    RequiredType NAME
    ValueConstraints []
    DefaultValues null
    AutoCreate true
    Mandatory true
    OnParentVersion ABORT
    Protected true
    Multiple false
PropertyDef
    Name jcr:frozenMixinTypes
    RequiredType NAME
    ValueConstraints []
    DefaultValues null
    AutoCreate false
    Mandatory false
    OnParentVersion ABORT
    Protected true
    Multiple true
PropertyDef
    Name jcr:frozenUuid
    RequiredType STRING
    ValueConstraints []
    DefaultValues null
    AutoCreate true
    Mandatory true
    OnParentVersion ABORT
    Protected true
    Multiple false
PropertyDef
    Name ""
    RequiredType UNDEFINED
    ValueConstraints []
    DefaultValues null
    AutoCreate false
    Mandatory false
    OnParentVersion ABORT
    Protected true
    Multiple false
PropertyDef
    Name ""
    RequiredType UNDEFINED
    ValueConstraints []
    DefaultValues null
    AutoCreate false
    Mandatory false
    OnParentVersion ABORT
    Protected true
    Multiple true
ChildNodeDef
    Name ""
    RequiredPrimaryTypes [nt:base]
    DefaultPrimaryType null
    AutoCreate false
    Mandatory false
    OnParentVersion ABORT
    Protected true
    SameNameSibs true

```

This node type is optional.

This node type is used in the versioning system. It is required in those implementations that support versioning. See [8.2 Versioning](#), for more details.

6.7.22.17 nt:versionedChild

```

NodeTypeName
    nt:versionedChild
Supertypes
    nt:base
IsMixin
    false
HasOrderableChildNodes
    false
PrimaryItemName
    null
PropertyDef
    Name jcr:childVersionHistory
    RequiredType REFERENCE
    ValueConstraints ["nt:versionHistory"]
    DefaultValues null
    AutoCreate true
    Mandatory true
    OnParentVersion ABORT
    Protected true
    Multiple false

```

This node type is optional.

This node type is used in the versioning system. It is required in those implementations that support versioning. See [8.2 Versioning](#), for more details.

6.7.22.18 nt:query

```

NodeTypeName
    nt:query
Supertypes
    nt:base
IsMixin
    false
HasOrderableChildNodes
    false
PrimaryItemName
    null
PropertyDef
    Name jcr:statement
    RequiredType STRING
    ValueConstraints []
    DefaultValues null
    AutoCreate false
    Mandatory false
    OnParentVersion COPY
    Protected false
    Multiple false
PropertyDef
    Name jcr:language
    RequiredType STRING
    ValueConstraints []

```

DefaultValues **null**
AutoCreate **false**
Mandatory **false**
OnParentVersion **COPY**
Protected **false**
Multiple **false**

This node type is optional.

This node type *may* be used to store a persistent query. See [6.6 Searching Repository Content](#), or more details

6.8 System Node

The location `/jcr:system` is reserved for use as a “system folder”. Some implementations may use this location for storing or exposing repository-internal metadata in content. For example, if a repository exposes node type definitions in content, then those node type definitions should be located at `/jcr:system/jcr:nodeTypes`.

If a repository supports versioning, then it must expose the version storage at `/jcr:system/jcr:versionStorage` (see [8.2.2.1 jcr:versionStorage](#)).

If `/jcr:system` is supported, then it should be a node of node type `nt:unstructured`.

7 Level 2 Repository Features

The following section explains level 2 of JCR API on a functional basis. For an explanation organized on an interface-by-interface basis, see the accompanying Javadoc.

Level 2 defines a *read/write* repository. This includes all features of level 1, as well as the following:

- Adding and removing nodes and properties
- Writing the values of properties
- Persistent namespace changes
- Import from XML/SAX
- Assigning node types to nodes
- Discovery of access control permissions

Where a level 1-only repository would differ in behavior from a level 2 repository, this is indicated.

7.1 Writing Repository Content

Changes can be made to a workspace through methods of:

- **Workspace** (for example, `move`, `copy` and `clone`),
- **Session** (for example, `move`),
- **Node** (for example, `addNode` and `setProperty`),
- **Item** (for example, `remove`) or
- **Property** (for example, `setValue`).

When changes are made through most methods of **Session**, **Node** or **Property** (as opposed to through **Workspace**) they are not immediately reflected in the repository workspace. The changes are held in transient storage associated with the **session** object until they are either persisted by a `save` or discarded by a `refresh`.

(Note that there are exceptions to this rule. The following **Node** methods act immediately on the persistent workspace and do not require `save`: `checkout`, `checkin`, `update`, `merge`, `doneMerge`, `cancelMerge`, `restore`, `restoreByLabel`, `lock` and `unlock`.)

session.save persists all pending changes currently stored in the **session** object. Conversely, **session.refresh** discards all pending changes currently stored in the **session**.

For more fine-grained control over which changes are persisted or discarded, the methods **Item.save** and **Item.refresh** are also

provided. `Item.save` saves all pending changes that apply to that particular item and its subtree. Analogously, `Item.refresh` discards all pending changes that apply to that item and its subtree.

While most of the write methods of `Session`, `Item`, `Node` and `Property` require a `save` to persist their changes, the write methods of `Workspace` (`copy`, `clone` and `move`) do not. `Workspace` methods act directly on persistent storage and are not mediated by the transient storage of the `session`. Note the contrast, for example, between `session.move`, which requires a `save`, and `Workspace.move`, which does not (see [7.1.6 Moving and Copying](#)).

If an item has been modified in the `session` but not yet saved, and its corresponding item in the persistent workspace is altered through a direct-to-workspace method, this has no effect on the transient state of the `session`. The altered item in the `session` remains and may be saved later. Of course, the change made to the workspace may render the attempt to save 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 `save` on *this session* may throw an `InvalidItemStateException`.

In the context of a particular change to an item, we often refer to `save` generically, as in, “the change will be persisted on `save`”. Such statements refer to any invocation of `save`, (`Session.save` or `Item.save`) that include the changed item within its scope.

javax.jcr. Session	
void	save() Validates all pending changes currently recorded in this <code>session</code> . If validation of <i>all</i> pending changes succeeds, then this change information is cleared from the <code>session</code> . If the <code>save</code> occurs outside a transaction, the changes are persisted and thus made visible to other <code>sessions</code> . If the <code>save</code> occurs within a transaction, the changes are not persisted until the transaction is committed (see 8.1 Transactions for more details). If validation fails, then no pending changes are saved and they remain recorded on the <code>session</code> . There is no best-effort or partial <code>save</code> . The item in persistent storage to which a transient item is saved is determined by matching UUIDs and paths. See 7.1.1 Saving by UUID and Path , for details. AccessDeniedException will be thrown if any of the changes

	<p>to be persisted would violate the access permissions of the session. This exception is also thrown if any of the changes to be persisted would cause the removal of a node that is currently referenced by a REFERENCE property that this session <i>does not</i> have read access to.</p> <p>A ConstraintViolationException will be thrown if any of the changes to be persisted would violate a node type restriction. Additionally, a repository may use this exception to enforce implementation- or configuration-dependant restrictions.</p> <p>An InvalidItemStateException is thrown if any of the changes to be persisted conflicts with a change already persisted through another session and the implementation is such that this conflict can only be detected at save-time and therefore was not detected earlier, at change-time.</p> <p>A ReferentialIntegrityException is thrown if any of the changes to be persisted would cause the removal of a node that is currently referenced by a REFERENCE property that this session has read access to.</p> <p>A VersionException is thrown if the save would result in a change to persistent storage that would violate the read-only status of a checked-in node.</p> <p>A LockException is thrown if the save would result in a change to persistent storage that would violate a lock.</p> <p>A RepositoryException will be thrown if another error occurs.</p>
void	<p>refresh(boolean keepChanges)</p> <p>If keepChanges is false, this method discards all pending changes currently recorded in this session and returns all items to reflect the current saved state. Outside a transaction this state is simply the current state of persistent storage. Within a transaction, this state will reflect persistent storage as modified by changes that have been saved but not yet committed.</p> <p>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 saved state, thus revealing changes made by other sessions. In some implementations this may be a null operation. See 7.1.2.4 Seeing Changes Made by Other Sessions, for more details.</p> <p>A RepositoryException is thrown if an error occurs.</p>

boolean	hasPendingChanges() Returns true if this session holds pending (that is, unsaved) changes; otherwise returns false . A RepositoryException is thrown if an error occurs.
---------	---

There are also the more fine-grained **save** and **refresh** methods on **Item**.

javax.jcr. Item	
void	save() Validates all pending changes currently recorded in this session that apply to this Item or any of its descendants (that is, the subtree rooted at this Item). If validation of <i>all</i> pending changes succeeds, then this change information is cleared from the session . If the save occurs outside a transaction, the changes are persisted and thus made visible to other sessions . If the save occurs within a transaction, the changes are not persisted until the transaction is committed (see 8.1 Transactions for more details). If validation fails, then no pending changes are saved and they remain recorded on the session . There is no best-effort or partial save . The item in persistent storage to which a transient item is saved is determined by matching UUIDs and paths. See 7.1.1 Saving by UUID and Path , for details. AccessDeniedException will be thrown if any of the changes to be persisted would violate the access permissions of the session . This exception is also thrown if any of the changes to be persisted would cause the removal of a node that is currently referenced by a REFERENCE property that this session <i>does not</i> have read access to. A ConstraintViolationException will be thrown if any of the changes to be persisted would violate a node type restriction. Additionally, a repository may use this exception to enforce implementation- or configuration-dependant restrictions. An InvalidItemStateException is thrown if any of the changes to be persisted conflicts with a change already persisted through another session and the implementation is such that this conflict can only be detected at save -time and therefore was not detected earlier, at change-time.

	<p>A ReferentialIntegrityException is thrown if any of the changes to be persisted would cause the removal of a node that is currently referenced by a REFERENCE property that this session has read access to.</p> <p>A VersionException is thrown if the save would result in a change to persistent storage that would violate the read-only status of a checked-in node.</p> <p>A LockException is thrown if the save would result in a change to persistent storage that would violate a lock.</p> <p>A RepositoryException will be thrown if another error occurs.</p>
void	<p>refresh(boolean keepChanges)</p> <p>If keepChanges is false, this method discards all pending changes currently recorded in this session that apply to this Item or any of its descendants (that is, the subtree rooted at this Item) and returns these items to reflect the current saved state. Outside a transaction this state is simple the current state of persistent storage. Within a transaction, this state will reflect persistent storage as modified by changes that have been saved but not yet committed.</p> <p>If keepChanges is true then pending changes are not discarded but items (within the subtree rooted at this Item) that do not have changes pending have their state refreshed to reflect the current saved state, thus revealing changes made by other sessions. In some implementations this may be a null operation. See 7.1.2.4 Seeing Changes Made by Other Sessions, for more details.</p> <p>An InvalidItemStateException is thrown if this Item object represents a workspace item that has been removed (either by this session or another).</p> <p>A RepositoryException is thrown if an error occurs.</p>

7.1.1 Saving by UUID and Path

When an item is saved (either through **Item.save** or **session.save**) the item in persistent storage to which pending changes are written is determined as follows:

- If the transient item has a UUID, then the changes are written to the persistent item with the same UUID.
- If the transient item does not have a UUID, then a combination of the UUID of its nearest UUID-bearing ancestor and its relative path from that ancestor is used to

determine the persistent item to which the changes will be written. For example, if

- node `/a/b/c` and node `/a/b` do not have UUIDs;
- node `/a` has UUID `u`;
- then, pending changes to transient node `/a/b/c` are written to the persistent node located at path `b/c` *relative to* the persistent node with UUID `u`.

As a result of these rules, a **save** of an item that has a UUID will succeed even if that item has, in the meantime, been moved in persistent storage to a new location (that is, its path has changed). However, a **save** of a non-UUID item will fail (throwing an `InvalidItemStateException`) if it has, in the meantime, been moved in persistent storage to a new location. A **save** of a non-UUID item will also fail if it has, in addition to being moved, been replaced in its original position by a UUID-bearing item.

Note that **save** uses the same rules to match items between transient storage and persistent storage as **update** does to match nodes between two workspaces. See [4.10.2 Multiple Workspaces and Corresponding Nodes](#).

7.1.2 Reflecting Item State

Every `Item` object (instance of a `Node` or `Property`) is associated with the `session` object through which it was acquired. When changes are made to an `Item` object, those changes are recorded in its associated `session` and immediately reflected in the `Item` object itself. In other words, after a change is made, a subsequent re-retrieval of the same item entity *through the same session*, will return an `Item` object reflecting the recent change. In this context “retrieval through the same `session`” includes not just acquisition of nodes and properties through the getter methods (like `getNode`, `getProperty` etc.) but also items returned through other means, such as within the result set of a query.

7.1.2.1 Re-using Item Objects

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 (associated with the same `session`) representing the same actual item entity. Note that the criteria of item identity in this context are those described above in [7.1.1 Saving by UUID and Path](#).

7.1.2.2 Effect of Save and Refresh

When a **save** is performed on an `Item`, any changes recorded for that `Item` in the `session` are persisted and the record of that

change in the `session` is removed. From the perspective of the API client, the apparent state of the `Item` itself does not change (apart from the values returned by `isNew` or `isModified`), since the item has reflected the changes since they were initially made. If one or more of the pending changes cause an exception to be thrown on `save`, then *no* pending changes are saved, not even those which did not cause the problem. In this case the set of pending changes recorded on the session is left unaffected.

When a `refresh` is performed on an `Item`, any pending changes recorded for that `Item` in the `session` are discarded and the state of the `Item` object reverts to its current saved state in the workspace. If an exception occurs on `refresh`, the set of pending changes recorded on the session is left unaffected and `Item` state is similarly unaffected.

7.1.2.3 Invalid Items

Methods of an `Item` object (i.e., `Node` or `Property`) may throw an `InvalidItemStateException` in certain circumstances.

The first case is if `Item.remove` has been called on the item. In this case any subsequent calls to any read or write methods or invocations of `save` or `refresh` on that `Item` will throw an `InvalidItemStateException`.

Before the removal is persisted (by a `save` on the *parent* of the removed node) it may be cancelled by a `refresh` on the parent of the removed node. This has the effect of reverting the parent node to its current saved state in the workspace. 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 on a write method of an `Item` if the change being made would (upon `save`) conflict with a change made, and persisted, through another `session`. If detection of the conflict is only possible at `save`-time, then `save` will throw an `InvalidItemStateException`.

Whether a conflict is detected when the change is made to the `Item` or later, when an attempt is made to save that change, depends on the implementation. The key issue is when a particular `session` sees changes made in the persistent storage by other sessions.

7.1.2.4 Seeing Changes Made by Other Sessions

When recording pending changes to an `Item` in the `session` at least two approaches are possible. Which approach is taken is up to the implementation.

- **Copy-on-Read:** When an `Item` object is instantiated, its state in persistent storage is copied to transient storage associated with the `session`. Any subsequent changes are applied to the transient state object. Upon `save`, the transient state object is copied back to persistent storage and removed from transient storage. In such an implementation, when an `Item` is retrieved through a particular `session`, any changes made to the persistent state of that item by another `session` will not be visible to the first `session` until a `refresh` and reacquisition of the item in question. Under this system, conflicts with persistent state will only be detected upon `save`, meaning that `InvalidItemStateException` will only be thrown on `save`, not earlier.
- **Copy-on-Write:** An alternative approach is *not* to immediately copy the state of a newly instantiated `Item` object to transient storage, but rather to only copy the state when a change is made to that state. In this case, as long as no changes are made to an `Item` object, its state always reflects the current state in persistent storage and thus any changes in that persistent state made by other sessions are immediately visible through the methods of that `Item`. If, on the other hand, a change is made to the `Item` state by this `Session` then the item state is copied from persistent storage to transient storage and the change is applied to that copy. From this point until a `refresh` or `save`, changes made to the persistent item will not be visible through the `Item` object. Note that in copy-on-write implementations a `refresh(true)` (a refresh that does not discard pending changes) will, by definition, have no effect.

This specification does not prescribe either of these approaches. Implementations are free to use either one (or indeed any other that may be suitable) as long as the minimal requirement is met of never allowing two `Item` objects acquired through the same `session` to reflect conflicting state information.

7.1.2.5 Resolving Conflicts with Persistent State

When an `InvalidItemStateException` is thrown (either at write-time or `save`-time) the client will usually wish to resolve the conflict. The standard solution is to do the following:

- If the `Item` in question has unsaved changes pending, make a temporary copy of it.
- `refresh` the original `Item`, thus discarding the recent changes (including the one which caused the conflict).
- Merge the changes recorded in the temporary copy with the now up-to-date `Item` object.

In those repositories that support it, clients may avoid such conflicts by using the locking mechanism (see [8.5 Locking](#)).

7.1.2.6 Item Status

JCR provides the following methods on `Item` for determining whether a particular item has pending changes (`isModified`) or constitutes part of the pending changes of its parent (`isNew`).

javax.jcr. Item	
boolean	isNew() Returns true if this is a new item, meaning that it exists only in transient storage on the session and has not yet been saved. Within a transaction, isNew on an Item may return false (because the item has been saved) even if that Item is not in persistent storage (because the transaction has not yet been committed). Note that in level 1 (that is, read-only) implementations, this method will always return false .
boolean	isModified() Returns true if this Item has been saved but has subsequently been modified through the current session and therefore the state of this item as recorded in the session differs from the state of this item as saved. Within a transaction, isModified on an Item may return false (because the Item has been saved since the modification) even if the modification in question is not in persistent storage (because the transaction has not yet been committed). Note that in level 1 (that is, read-only) implementations, this method will always return false .

7.1.3 Adding Nodes

The methods for adding nodes are:

javax.jcr. Node	
Node	addNode(String relPath) Creates a new node at relPath . The new node will only be persisted on save if it meets the constraint criteria of the parent node's node type. In order to save a newly added node, save must be called

	<p>either on the <code>session</code>, or on the new node's parent or higher-order ancestor (grandparent, etc.). An attempt to call <code>save only</code> on the newly added node will throw a <code>RepositoryException</code>.</p> <p>In the context of this method the <code>relPath</code> provided must not have an index on its final element. If it does then a <code>RepositoryException</code> is thrown.</p> <p>Strictly speaking, the parameter is actually a <i>relative path</i> to the parent node of the node to be added, appended with the <i>name</i> desired for the new node (if the node is being added directly below <code>this</code> node then only the name need be specified). It does not specify a position within the child node ordering (if such ordering is supported). If ordering is supported by the node type of the parent node then the new node is appended to the end of the child node list.</p> <p>If <code>relPath</code> implies intermediary nodes that do not exist, then a <code>PathNotFoundException</code> is thrown.</p> <p>If an item already exists at <code>relPath</code>, and same-name siblings are not permitted, then an <code>ItemExistsException</code> is thrown immediately (not on <code>save</code>).</p> <p>A <code>ConstraintViolationException</code> will be thrown immediately if adding the node would violate a node type or implementation-specific constraint and that violation can be detected at the time that the <code>addNode</code> method is called. Otherwise, a <code>ConstraintViolationException</code> will be thrown on <code>save</code> if the violation can only be detected later, at the time that the <code>save</code> method is called. Note that implementations may differ as to which constraints are enforced immediately and which on <code>save</code>.</p> <p>A <code>ConstraintViolationException</code> will also be thrown immediately if an attempt is made to add a node as a child of a property.</p> <p>A <code>VersionException</code> is thrown if the node to which the new child is being added is versionable and checked-in or is non-versionable but its nearest versionable ancestor is checked-in.</p> <p>A <code>LockException</code> is thrown if a lock prevents the addition of the node.</p> <p>Since this signature does not allow explicit node type assignment, the new node's primary node type will be determined immediately (not on <code>save</code>) by the child node definitions in the node types of its parent. If there is no</p>
--	---

	<p>child node definition corresponding to the name specified for this new node, then a ConstraintViolationException is thrown. See 7.4.2 Assigning a Primary Node Type.</p> <p>A RepositoryException is thrown if the last element of relPath has an index or if another error occurs.</p> <p>Further node type related validation is deferred until save, at which point a ConstraintViolationException may be thrown by the save method.</p>
Node	<p>addNode(String relPath, String primaryNodeTypeName)</p> <p>Creates a new node at relPath of the specified primary node type.</p> <p>The same as addNode(String relPath) except that the primary node type of the new node is explicitly specified.</p> <p>If an item already exists at relPath, and same-name siblings are not permitted, then an ItemExistsException is thrown immediately (not on save).</p> <p>If primaryNodeTypeName is not recognized, then a NoSuchNodeTypeErrorException is thrown immediately (i.e., not on save).</p> <p>A ConstraintViolationException will be thrown immediately if adding the node would violate a node type or implementation-specific constraint and that violation can be detected at the time that the addNode method is called. Otherwise, a ConstraintViolationException will be thrown on save if the violation can only be detected later, at the time that the save method is called. Note that implementations may differ as to which constraints are enforced immediately and which on save.</p> <p>A ConstraintViolationException will also be thrown immediately if an attempt is made to add a node as a child of a property.</p> <p>A VersionException is thrown if the node to which the new child is being added is versionable and checked-in or is non-versionable but its nearest versionable ancestor is checked-in.</p> <p>A LockException is thrown if a lock prevents the addition of the node.</p> <p>A RepositoryException is thrown if the last element of relPath has an index or if another error occurs.</p> <p>Further node-type related validation is deferred until save,</p>

	at which point a <code>ConstraintViolation</code> may be thrown by the <code>save</code> method.
--	--

7.1.3.1 Example

If we wish to add a new “shape” to our product information, we might do it like this:

```
Node productsNode = root.getNode("products");
Node triangleNode = productsNode.addNode("triangle");
Node contentNode = triangleNode.addNode("jcr:content");
contentNode.setProperty("myapp:title", "Triangle: an
                        economical choice");
contentNode.setProperty("myapp:price", 50);
contentNode.setProperty("myapp:lead", "Triangles have
                        three sides, but they're not always
                        equal!");
productsNode.save();
```

This would add the new node `triangle` below the `products` node and add to `triangle`'s `jcr:content` node the properties `myapp:title`, `myapp:price` and `myapp:lead` with the specified values.

As another example, suppose we wish to iterate through a collection of strings and add each as a new paragraph under the node `triangle/jcr:content`. In that case, we might do the following:

```
Node contentNode = triangleNode.getNode("jcr:content");
for (Iterator i = strings.iterator(); i.hasNext();) {
    String text = (String) i.next();
    Node paraNode = contentNode.addNode("paragraph");
    paraNode.setProperty("text", text);
}
```

For each string retrieved from `strings` a new node is created called `paragraph` which is given a new property called `text`, which, in turn, is assigned the retrieved string.

7.1.4 Adding and Writing Properties

To add new properties or change the values of existing properties of a node we use the `setProperty` methods of `Node`:

javax.jcr. Node	
Property	<code>setProperty(String name, Value value)</code> Sets the specified (single value) property of this node to the specified <code>value</code> . If the property does not yet exist, it is created. The property type of the property

	<p>will be that specified by the node type of this node.</p> <p>If the property type of the supplied <code>value</code> object is different from that required, then a best-effort conversion is attempted. If the conversion fails, a <code>ValueFormatException</code> is thrown. If another error occurs, a <code>RepositoryException</code> is thrown.</p> <p>If the node type of this node does not indicate a specific property type, then the property type of the supplied <code>value</code> object is used and if the property already exists (has previously been set) it assumes both the new value and new property type.</p> <p>If the property is multi-valued, a <code>ValueFormatException</code> is thrown.</p> <p>A <code>ConstraintViolationException</code> is thrown if a node-type or other constraint violation is detected immediately. Otherwise, if the violation is only detected later, at <code>save</code>, then a <code>ConstraintViolationException</code> is thrown by that method. Implementations may differ as to which constraints are enforced immediately, and which on <code>save</code>.</p> <p>Passing a <code>null</code> as the second parameter removes the property. It is equivalent to calling <code>remove</code> on the <code>Property</code> object itself. For example, <code>N.setProperty("P", (Value)null)</code> would remove property called "P" of the node in <code>N</code>.</p> <p>To persist the addition, removal or change of a property to the workspace, <code>save</code> must be called on the session, <code>this</code> node, or a higher-order ancestor of the property.</p> <p>A <code>VersionException</code> is thrown if <code>this</code> node is versionable and checked-in or is non-versionable but its nearest versionable ancestor is checked-in.</p> <p>A <code>LockException</code> is thrown if a lock prevents the setting of the property.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>
Property	<p><code>setProperty(String name, Value[] values)</code></p> <p>Sets the specified (multi-value) property to the specified array of values. If the property does not yet exist, it is created. Same as</p>

	<p>setProperty(String name, Value value)</p> <p>except that an array of value objects is assigned instead of a single Value.</p> <p>The property type of the property will be that specified by the node type of this node. If the property type of the supplied value objects is different from that required, then a best-effort conversion is attempted. If the conversion fails, a ValueFormatException is thrown. All value objects in the array must be of the same type, otherwise a ValueFormatException is thrown. If the property is not multi-valued then a ValueFormatException is also thrown. If another error occurs, a RepositoryException is thrown.</p> <p>If the node type of this node does not indicate a specific property type, then the property type of the supplied value objects is used and if the property already exists it assumes both the new values and the new property type.</p> <p>A ConstraintViolationException is thrown if a node-type or other constraint violation is detected immediately. Otherwise, if the violation is only detected later, at save, then a ConstraintViolationException is thrown by that method. Implementations may differ as to which constraints are enforced immediately, and which on save.</p> <p>Passing a null as the second parameter removes the property. It is equivalent to calling remove on the Property object itself. For example, N.setProperty("P", (Value[])null) would remove property called "P" of the node in N.</p> <p>Note that this is different from passing an array that contains null elements. In such a case, the array is compacted by removing the nulls. The resulting set of values of the property never contains nulls. However, the set may be empty: N.setProperty("P", new Value[]{null}) would set the property to the empty set of values.</p> <p>To persist the addition, removal or change of a property to the workspace, save must be called on the session, this node, or a higher-order ancestor of the property.</p> <p>A VersionException is thrown if this node is versionable and checked-in or is non-versionable but</p>
--	---

	<p>its nearest versionable ancestor is checked-in.</p> <p>A LockException is thrown if a lock prevents the setting of the property.</p> <p>A RepositoryException is thrown if another error occurs.</p>
Property	<pre>setProperty(String name, Value[] values, int type)</pre> <p>Sets the specified (multi-value) property to the specified array of values. If the property does not yet exist, it is created.</p> <p>The property type is determined by the applicable property definition (found in the node type of this node).</p> <p>If that does not provide sufficient disambiguation then the type parameter is used.</p> <p>If the property type of the supplied value objects is different from that required, then a best-effort conversion is attempted. If the conversion fails, a ValueFormatException is thrown.</p> <p>If the property already exists it assumes both the new values and the new property type.</p> <p>All value objects in the array must be of the same type, otherwise a ValueFormatException is thrown. If the property is not multi-valued then a ValueFormatException is also thrown. If another error occurs, a RepositoryException is thrown.</p> <p>A ConstraintViolationException is thrown if a node-type or other constraint violation is detected immediately. Otherwise, if the violation is only detected later, at save, then a ConstraintViolationException is thrown by that method. Implementations may differ as to which constraints are enforced immediately, and which on save.</p> <p>Passing a null as the second parameter removes the property. It is equivalent to calling remove on the Property object itself. For example, N.setProperty("P", (Value[])null, type) would remove property called "P" of the node in N.</p> <p>Note that this is different from passing an array that contains null elements. In such a case, the array is</p>

	<p>compacted by removing the <code>nulls</code>. The resulting set of values of the property never contains <code>nulls</code>. However, the set may be empty: <code>N.setProperty("P", new Value[] { null }, type)</code> would set the property to the empty set of values.</p> <p>To persist the addition, removal or change of a property to the workspace, <code>save</code> must be called on the session, <code>this</code> node, or a higher-order ancestor of the property.</p> <p>A <code>VersionException</code> is thrown if <code>this</code> node is versionable and checked-in or is non-versionable but its nearest versionable ancestor is checked-in.</p> <p>A <code>LockException</code> is thrown if a lock prevents the setting of the property.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>
Property	<p><code>setProperty(String name, String[] values)</code></p> <p>Sets the specified property to the specified array of values. Same as</p> <p><code>setProperty(String name, Value[] values)</code></p> <p>except that the values are specified as <code>string</code> objects instead of <code>value</code> objects.</p>
Property	<p><code>setProperty(String name, String[] values, int type)</code></p> <p>Sets the specified property to the specified array of values. Same as</p> <p><code>setProperty(String name, Value[] values, int type)</code></p> <p>except that the values are specified as <code>string</code> objects instead of <code>value</code> objects.</p>
Property	<p><code>setProperty(String name, String value)</code></p> <p><code>setProperty(String name, InputStream value)</code></p> <p><code>setProperty(String name, boolean value)</code></p> <p><code>setProperty(String name, Calendar value)</code></p> <p><code>setProperty(String name, double value)</code></p>

	<pre>setProperty(String name, long value)</pre> <pre>setProperty(String name, Node node)</pre> <p>Sets the specified property to the specified value. In the context of these methods, each Java type <i>implies</i> a particular property type. The correspondence is:</p> <pre>String: PropertyType.STRING</pre> <pre>InputStream: PropertyType.BINARY</pre> <pre>boolean: PropertyType.BOOLEAN</pre> <pre>Calendar: PropertyType.DATE</pre> <pre>double: PropertyType.DOUBLE</pre> <pre>long: PropertyType.LONG</pre> <pre>Node: PropertyType.REFERENCE</pre> <p>In the case of the signature that takes a Node, the REFERENCE property in question is set to <i>refer</i> to the supplied node (see 6.2.5.4 Reference).</p> <p>The property type of the property being set is determined by the node type of this node. If this property type is something other than that implied by the (Java) type of the passed value, a best-effort conversion is attempted. If the conversion fails, a ValueFormatException is thrown. If the property is multi-valued, a ValueFormatException is also thrown. If another error occurs, a RepositoryException is thrown.</p> <p>If the node type of this node does not specify a particular property type for the property being set then the property type implied by the (Java) type of the passed value is used and if the property already exists (has previously been set) it assumes both the new value and new type.</p> <p>A ConstraintViolationException is thrown if a node-type or other constraint violation is detected immediately. Otherwise, if the violation is only detected later, at save, then a ConstraintViolationException is thrown by that method. Implementations may differ as to which constraints are enforced immediately, and which on save.</p> <p>Passing a null as the second parameter removes the property. It is equivalent to calling remove on the</p>
--	--

	<p>Property object itself. For example, N.setProperty("P", (Calendar)null) would remove property called "P" of the node in N. Obviously, a null cannot be passed to the signatures that take the primitive types boolean, long or double.</p> <p>To persist the addition, removal or change of a property to the workspace, save must be called on this node or a higher-order ancestor of this node.</p> <p>To create a property of PropertyType.NAME or PropertyType.PATH an explicit type must be specified using a three-argument signature.</p> <p>A VersionException is thrown if this node is versionable and checked-in or is non-versionable but its nearest versionable ancestor is checked-in.</p> <p>A LockException is thrown if a lock prevents the setting of the property.</p> <p>A RepositoryException is thrown if another error occurs.</p>
--	--

To change the value of a property that has already been retrieved you can also use the **setValue** methods in the **Property** Interface itself:

javax.jcr. Property	
void	setValue(Value value) setValue(Value[] values) setValue(String value) setValue(String[] values) setValue(InputStream value) setValue(double value) setValue(long value) setValue(Calendar value) setValue(boolean value) setValue(Node node) <p>Sets the value of this Property to the specified value. The behavior of these methods is identical with their corresponding Node.setProperty signature.</p>

7.1.4.1 Example

To change the price of a rhombus to \$200 we might do the following:

```
String path = "products/rhombus/jcr:content/myapp:price";
root.getProperty(path).setValue(200);
```

or, alternatively

```
String path = "products/rhombus/jcr:content";
Node rhombus = root.getNode(path);
rhombus.setProperty("myapp:price", 200);
```

7.1.4.2 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 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`.

7.1.5 Removing Nodes and Properties

Removing a node or property is done with the `Item.remove` method:

javax.jcr. Item	
void	remove() Removes this item (and its subtree). To persist a removal, a save must be performed that includes the (former) parent of the removed item within its scope. A ConstraintViolationException will be thrown immediately if removing the specified item would violate a node type or implementation-specific constraint and that violation can be detected at the time that the remove method is called. Otherwise, a ConstraintViolationException will be thrown on save if the violation can only be detected later, at the time that the save method is called. Note that implementations may differ as to which constraints are enforced immediately and which on save . A ReferentialIntegrityException will be thrown on save if the item in question is currently the target of a REFERENCE property elsewhere in the workspace and the current session has read access to that REFERENCE property.

	<p>An AccessDeniedException will be thrown on save if the item in question is currently the target of a REFERENCE property elsewhere in the workspace and the current session <i>does not</i> have read access to that REFERENCE property.</p> <p>A VersionException is thrown immediately (not on save) if the parent node of this item is versionable and currently checked-in or is non-versionable and its nearest versionable ancestor is checked-in. Note that if this item itself is checked-in (but not its parent) then it can be removed.</p> <p>A LockException is thrown if a lock prevents the removal of this item.</p> <p>A RepositoryException is thrown on remove if an error occurs.</p>
--	--

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 also [4.7.3 No Null Values](#).

7.1.6 Moving and Copying

Moving and copying of nodes is done through methods of the **session** and **Workspace** interfaces.

session provides a **move** method:

```
javax.jcr.  
Session
```

void	<p>move(String srcAbsPath, String destAbsPath)</p> <p>Moves the node at srcAbsPath (and its entire subtree) to the new location at destAbsPath.</p> <p>In order to persist the change, a save must be called on the parents of both the source and destination locations.</p> <p>A ConstraintViolationException is thrown if a node-type or other constraint violation is detected immediately. Otherwise, if the violation is only detected later, at save, then a ConstraintViolationException is thrown by that method. Implementations may differ as to which constraints are enforced immediately, and which on save.</p> <p>As well, a ConstraintViolationException will be thrown on save if an attempt is made to save <i>only</i> either the source or destination node separately.</p> <p>Note that this behavior differs from that of Workspace.move (see below), which operates directly in the persistent workspace and does not require a save.</p> <p>The destAbsPath provided must not have an index on its final element. If it does then a RepositoryException is thrown. Strictly speaking, the destAbsPath parameter is actually an <i>absolute path</i> to the parent node of the new location, appended with the new <i>name</i> desired for the moved node. It does not specify a position within the child node ordering. If ordering is supported by the node type of the parent node of the new location, then the newly moved node is appended to the end of the child node list.</p> <p>This method cannot be used to move just an individual property by itself. It moves an entire node and its subtree (including, of course, any properties contained therein).</p> <p>If no node exists at srcAbsPath or no node exists one level above destAbsPath (i.e. there is no node that will serve as the parent of the moved item) then a PathNotFoundException is thrown.</p> <p>An ItemExistsException is thrown if a property already exists at destAbsPath or a node already exists there and same-name siblings are not allowed.</p> <p>A VersionException is thrown if the parent node of destAbsPath or the parent node of srcAbsPath is versionable and checked-in, or is non-versionable and</p>
------	--

	<p>its nearest versionable ancestor is checked-in.</p> <p>A LockException is thrown if a lock prevents the move.</p> <p>A RepositoryException is thrown if another error occurs.</p>
--	--

Workspace provides a **move** method as well as the methods **copy** and **clone**:

javax.jcr. Workspace	
void	<p>copy(String srcAbsPath, String destAbsPath)</p> <p>This method copies the node at srcAbsPath and its entire subtree to the new location at destAbsPath. This operation is performed entirely within the persistent workspace, it does not involve transient storage and therefore does not require a save.</p> <p>Copies of referenceable nodes (nodes with UUIDs) are automatically given new UUIDs.</p> <p>The destAbsPath provided must not have an index on its final element. If it does, then a RepositoryException is thrown. Strictly speaking, the destAbsPath parameter is actually an <i>absolute path</i> to the parent node of the new location, appended with the new <i>name</i> desired for the copied node. It does not specify a position within the child node ordering. If ordering is supported by the node type of the parent node of the new location, then the newly moved node is appended to the end of the child node list.</p> <p>This method cannot be used to copy just an individual property by itself. It copies an entire node and its subtree (including, of course, any properties contained therein).</p> <p>A ConstraintViolationException is thrown if the operation would violate a node-type or other implementation-specific constraint.</p> <p>A VersionException is thrown if the parent node of destAbsPath is versionable and checked-in, or is non-versionable but its nearest versionable ancestor is checked-in.</p> <p>An AccessDeniedException is thrown if the current</p>

	<p>session (i.e., the session that was used to acquire this Workspace object) does not have sufficient access permissions to complete the operation.</p> <p>A PathNotFoundException is thrown if the node at srcAbsPath or the parent of the new node at destAbsPath does not exist.</p> <p>An ItemExistsException is thrown if a property already exists at destAbsPath or a node already exists there and same-name siblings are not allowed.</p> <p>A LockException is thrown if a lock prevents the copy.</p> <p>A RepositoryException is thrown if another error occurs.</p>
void	<pre>copy(String srcWorkspace, String srcAbsPath, String destAbsPath)</pre> <p>This method copies the subtree at srcAbsPath in srcWorkspace to destAbsPath in this workspace. Unlike clone, this method <i>does</i> assign new UUIDs to the new copies of referenceable nodes. This operation is performed entirely within the persistent workspace, it does not involve transient storage and therefore does not require a save.</p> <p>The destAbsPath provided must not have an index on its final element. If it does, then a RepositoryException is thrown. Strictly speaking, the destAbsPath parameter is actually an <i>absolute path</i> to the parent node of the new location, appended with the new <i>name</i> desired for the copied node. It does not specify a position within the child node ordering. If ordering is supported by the node type of the parent node of the new location, then the new copy of the node is appended to the end of the child node list.</p> <p>This method cannot be used to copy just an individual property by itself. It copies an entire node and its subtree (including, of course, any properties contained therein).</p> <p>A NoSuchWorkspaceException is thrown if srcWorkspace does not exist.</p> <p>A ConstraintViolationException is thrown if the operation would violate a node-type or other</p>

	<p>implementation-specific constraint.</p> <p>A VersionException is thrown if the parent node of destAbsPath is versionable and checked-in, or is non-versionable but its nearest versionable ancestor is checked-in.</p> <p>An AccessDeniedException is thrown if the current session (i.e., the session that was used to acquire this Workspace object) does not have sufficient access permissions to complete the operation.</p> <p>A PathNotFoundException is thrown if the node at srcAbsPath or the parent of the new node at destAbsPath does not exist.</p> <p>An ItemExistsException is thrown if a property already exists at destAbsPath or a node already exists there and same-name siblings are not allowed.</p> <p>A LockException is thrown if a lock prevents the copy.</p> <p>A RepositoryException is thrown if another error occurs.</p>
void	<pre>clone(String srcWorkspace, String srcAbsPath, String destAbsPath, boolean removeExisting)</pre> <p>Clones the subtree at the node srcAbsPath in srcWorkspace workspace to destAbsPath in this workspace. Unlike the signature of copy that copies between workspaces, this method <i>does not</i> assign new UUIDs to the new nodes but preserves the UUIDs (if any) of their respective source nodes.</p> <p>If removeExisting is true and an existing node in this workspace (the destination workspace) has the same UUID as a node being cloned from srcWorkspace, then the incoming node takes precedence, and the existing node (and its subtree) is removed. If removeExisting is false then a UUID collision causes this method to throw a ItemExistsException and no changes are made.</p> <p>If successful, the changes are persisted immediately, there is no need to call save.</p> <p>The destAbsPath provided must not have an index on its final element. If it does, then a RepositoryException is thrown. Strictly speaking,</p>

	<p>the destAbsPath parameter is actually an <i>absolute path</i> to the parent node of the new location, appended with the new <i>name</i> desired for the copied node. It does not specify a position within the child node ordering. If ordering is supported by the node type of the parent node of the new location, then the new clone of the node moved node is appended to the end of the child node list.</p> <p>This method cannot be used to clone just an individual property by itself. It clones an entire node and its subtree (including, of course, any properties contained therein).</p> <p>A NoSuchWorkspaceException is thrown if srcWorkspace does not exist.</p> <p>A ConstraintViolationException is thrown if the operation would violate a node-type or other implementation-specific constraint</p> <p>A VersionException is thrown if the parent node of destAbsPath is versionable and checked-in, or is non-versionable but its nearest versionable ancestor is checked-in.</p> <p>An AccessDeniedException is thrown if the current session (i.e. the session that was used to acquire this Workspace object) does not have sufficient access permissions to complete the operation.</p> <p>A PathNotFoundException is thrown if the node at srcAbsPath or the parent of destAbsPath do not exist.</p> <p>An ItemExistsException is thrown if a property already exists at destAbsPath or a node already exists there and same-name siblings are not allowed or if removeExisting is false and a UUID conflict occurs.</p> <p>A LockException is thrown if a lock prevents the clone.</p> <p>A RepositoryException if another error occurs.</p>
void	<p>move(String srcAbsPath, String destAbsPath)</p> <p>Moves the node at srcPath (and its entire subtree) to the new location at destPath. If successful, the change is persisted immediately, there is no need to call save. Note that this is in contrast to Session.move which operates within the transient</p>

	<p>space and hence requires a save.</p> <p>The destAbsPath provided must not have an index on its final element. If it does then a RepositoryException is thrown. Strictly speaking, the destAbsPath parameter is actually an <i>absolute path</i> to the parent node of the new location, appended with the new <i>name</i> desired for the moved node. It does not specify a position within the child node ordering. If ordering is supported by the node type of the parent node of the new location, then the newly moved node is appended to the end of the child node list.</p> <p>This method cannot be used to move just an individual property by itself. It moves an entire node and its subtree (including, of course, any properties contained therein).</p> <p>A ConstraintViolationException is thrown if the operation would violate a node-type or other implementation-specific constraint</p> <p>A VersionException is thrown if the parent node of destAbsPath or the parent node of srcAbsPath is versionable and checked-in, or is non-versionable but its nearest versionable ancestor is checked-in.</p> <p>An AccessDeniedException is thrown if the current session does not have sufficient access permissions to complete the operation.</p> <p>A PathNotFoundException is thrown if the item at srcAbsPath or the parent of destAbsPath does not exist.</p> <p>An ItemExistsException is thrown if a property already exists at destAbsPath or a node already exists there and same-name siblings are not allowed.</p> <p>A LockException is thrown if a lock prevents the move.</p> <p>A RepositoryException is thrown if another error occurs.</p>
--	---

7.1.6.1 Example

The following code,

```
Workspace workspace = ...;
workspace.move("/products/TV/Paragraph",
              "/products/Radio/Paragraph");
```

would move a paragraph from one location to another.
`Workspace.copy` works analogously.

7.1.7 Updating and Cloning Nodes across Workspaces

In repositories that have multiple workspaces a node in one workspace may have *corresponding nodes* in other workspaces.

Given a node N_1 in workspace w_1 , its corresponding node N_2 in workspace w_2 is defined as follows:

- If N_1 is the root node of w_1 then N_2 is the root node of w_2 .
- If N_1 is referenceable (has a UUID) then N_2 is the node in w_2 with the same UUID.
- If N_1 is not referenceable (does not have a UUID) then there is some node M_1 which is either the nearest ancestor of N_1 that is referenceable, or is the root node of w_1 . If the corresponding node of M_1 is M_2 in w_2 , then N_2 is the non-referenceable node with the same relative path from M_2 as N_1 has from M_1 .
- The above statements are qualified by the principle that referenceable nodes may only correspond to other referenceable nodes and non-referenceables to other non-referenceables. A referenceable and non-referenceable can never correspond with on another.

Corresponding nodes can be thought of as instances of “the same node” that exist across different workspaces.

Apart from the characteristics described above (UUID, UUID plus relative path, or absolute path), corresponding nodes need have nothing else in common. They can differ, for example, in terms of their properties and child nodes.

7.1.7.1 Creating a Corresponding Node

Corresponding nodes can be created by “cloning” a node (or subtree of nodes) from one workspace to another using the `Workspace.clone` method:

```
Workspace.clone(String srcWorkspace,  
                String srcAbsPath,  
                String destAbsPath,  
                boolean removeExisting)
```

This method clones the subtree at `srcAbsPath` in `srcWorkspace` to `destAbsPath` in `this` workspace. The `clone` method clones both referenceable and nonreferenceable nodes. In the case of referenceable nodes `clone` preserves the node's UUID so that the

new node in the destination workspace has the same UUID as the node in the source workspace.

For a non-referenceable node, a corresponding node in another workspace can be created either through `clone`, or simply by creating a node in the destination workspace (using `Node.addNode`) with the same relative path to the nearest referenceable node.

However, the use of `clone` is *required* for the creation corresponding referenceable nodes because simply creating a new referenceable node at the same path in the other workspace will not work, since the new node will automatically be assigned a new UUID and not the same UUID as its correspondee.

Recall (from [4.9 Referenceable Nodes](#)) that a repository may make its workspace root nodes `mix:referenceable`. If so, then they must all have the same UUID. The root node of a given workspace is created when that workspace is created.

If there already exists anywhere in this workspace a node with the same UUID 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 UUID from `srcWorkspace` is copied to this workspace as part of the copied subtree (that is, not into the former location of the old node). The subtree of the cloned node will reflect the clones state 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).

7.1.7.2 Update

Node correspondence governs the behavior of the `update` method. This method causes `this` node to be updated to reflect the state of its corresponding node in `srcWorkspace`.

javax.jcr. Node	
void	update(String srcWorkspaceName) If this node does have a corresponding node in the workspace <code>srcWorkspaceName</code> , then this method traverses down the subtree rooted at this node and replaces the state of each node with that of its

	<p>corresponding node in the specified source workspace.</p> <p>The “state” of the node in this context means the set of properties and child nodes it has. In other words, when a node is updated, its set of properties and child nodes is replaced by that of its corresponding node in the source workspace. One repercussion of this is that if a node further down the subtree does not have a corresponding node, then that node is removed (thus aligning the state of its parent with <i>its</i> correspondee in the other workspace).</p> <p>If this node does not have a corresponding node in the workspace <code>srcWorkspaceName</code>, then the update method has no effect (it does not traverse down the subtree).</p> <p>If the <code>update</code> succeeds, the changes made to <code>this</code> node and its subtree are persisted immediately, there is no need to call <code>save</code>.</p> <p>Note that <code>update</code> does not respect the checked-in status of nodes. An <code>update</code> may change a node even if it is currently checked-in (this fact is only relevant in systems that support versioning, see 8.2 Versioning).</p> <p>If the specified <code>srcWorkspace</code> does not exist, a <code>NoSuchWorkspaceException</code> is thrown.</p> <p>If the current session does not have sufficient permissions to perform the operation, then an <code>AccessDeniedException</code> is thrown.</p> <p>An <code>InvalidItemStateException</code> is thrown if this <code>Session</code> (not necessarily this <code>Node</code>) has pending unsaved changes.</p> <p>A <code>LockException</code> is thrown if a lock prevents the update.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>
--	---

7.1.7.3 getCorrespondingNodePath

The API also provides a method for quickly finding the path of a node's corresponding node (if it exists) in another workspace:

<code>javax.jcr.</code> Node	
<code>String</code>	<code>getCorrespondingNodePath(String workspaceName)</code> Returns the absolute path of the node in the specified

	<p>workspace that corresponds to this node.</p> <p>If no corresponding node exists then an ItemNotFoundException is thrown.</p> <p>If the specified workspace does not exist then a NoSuchWorkspaceException is thrown.</p> <p>If the current session does not have sufficient permissions to perform this operation, an AccessDeniedException is thrown.</p> <p>Throws a RepositoryException if another error occurs.</p>
--	--

7.1.8 Referenceable Nodes

A node that is referenceable has an independent identity from its position in the workspace hierarchy (by virtue of its UUID): it maintains its identity regardless of where it is moved in the hierarchy.

Non-referenceable nodes, on the other hand, are intrinsically tied to their position in the hierarchy relative to their nearest referenceable ancestor. If a non-referenceable node is moved from its position it becomes, in effect, a different node.

Consequently, a referenceable node and its non-referenceable sub-nodes form a natural unit within the WS hierarchy. It is this unit that is respected during a save, update and merge.

7.1.9 Treatment of UUIDs

A number of different methods in the JCR API transfer node state from one location to another. They often differ in how they treat the UUID 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.

Method	New UUID	Keep UUID (3 behaviors on conflict)			Comments
		Throw	Remove from existing location	Replace at existing location	
<code>Workspace.copy</code> (see 7.1.6 Moving and Copying)	yes	no	no	no	copy (both within and between workspaces) simply creates a new UUID for any referenceable nodes it copies.

Session.save Item.save (see 7.1 Writing Repository Content)	no	no	no	yes	save pushes items to the persistent workspace, replacing existing items using UUID matching, wherever they may be in terms of path (non-referenceable nodes are kept bound to their UUID-bearing ancestor, however).
Node.update (see 7.1.7 Updating and Cloning Nodes across Workspaces)	no	no	no	yes	update pulls the state of this node from another workspace using UUID (plus relative path, for non-referenceable nodes) regardless of where those source nodes are in the source workspace in terms of path.
Workspace.clone (see 7.1.6 Moving and Copying)	no	yes	yes	no	clone keeps UUIDs. There are two options to deal with cases where this workspace already contains a node with the same UUID as one being cloned over: either throw, or remove the existing node in this workspace.
Node.restore , Node.restoreByLabel Workspace.restore (see 8.2 Versioning)	no	yes	yes	no	restore and restoreByLabel keep UUIDs. Similar to clone , there are two options to deal with cases where this workspace already contains a node with the same UUID as being copied in as part of a restored version: either throw, or remove the existing node in this workspace.
Workspace.import (see 7.3 Importing)	yes	yes	yes	yes	All four options are supported. Note that

<i>Repository Content)</i>					<code>Session.import</code> is not in this table because it requires save in any case, and that case is already covered above.
----------------------------	--	--	--	--	---

7.1.10 Ordering Child Nodes

If a node supports *orderable child nodes* then the following method can be used to arrange the order of its child nodes.

javax.jcr. Node	
void	orderBefore(String srcChildRelPath, String destChildRelPath) <p>If this node supports child node ordering, this method inserts the child node at <code>srcChildRelPath</code> before its sibling, the child node at <code>destChildRelPath</code>, in the child node list. To place the node <code>srcChildRelPath</code> at the end of the list, a <code>destChildRelPath</code> of <code>null</code> is used.</p> <p>Note that (apart from the case where <code>destChildRelPath</code> is <code>null</code>) both of these arguments must be relative paths of depth one, in other words they are the <i>names</i> of the child nodes, possibly suffixed with an index (see 4.6.1 Names vs. Paths).</p> <p>Changes to ordering of child nodes are persisted on save of the parent node.</p> <p>If this node does not support child node ordering, then a UnsupportedRepositoryOperationException is thrown.</p> <p>If <code>srcChildRelPath</code> and <code>destChildRelPath</code> are the same, then a ConstraintViolationException is thrown.</p> <p>A ConstraintViolationException is also thrown if a node-type or implementation-specific constraint violation is detected immediately. Otherwise, if the violation can only be detected later, on save, then that method throws a ConstraintViolationException. Implementations may differ as to which constraints are enforced immediately, and which on save.</p> <p>If either parameter is not the relative path to a child node of this node, then an ItemNotFoundException is thrown.</p> <p>A VersionException is thrown if this node is versionable and checked-in or is non-versionable but its nearest versionable ancestor is checked-in.</p>

	<p>A LockException is thrown if a lock prevents the re-ordering.</p> <p>If another error occurs a RepositoryException is thrown.</p>
--	--

A node supports orderable child nodes if and only if it is of a primary node type that returns `true` on `NodeType.hasOrderableChildNodes()`.

If a node has orderable child nodes then at any time the current order of its child nodes is reflected in the order of nodes in the iterator returned by `Node.getNodes`.

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.

Note that the order of properties returned by `Node.getProperties` is never client-controllable.

See [4.4](#) *Orderable Child Nodes*.

7.2 Adding and Deleting Namespaces

As discussed in [6.3](#) *Namespaces*, above, each JCR repository has a single, persistent namespace registry represented by the `NamespaceRegistry` object accessed via `Workspace.getNamespaceRegistry()`. In level 1 only the `NamespaceRegistry` methods related to discovering information must function. In level 2 the `NamespaceRegistry` additionally allows for persistent changes to namespace mappings using the following methods:

javax.jcr. NamespaceRegistry	
void	registerNamespace(String prefix, String uri) Sets a one-to-one mapping between <code>prefix</code> and <code>uri</code> 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. In general this can almost always be done, though an implementation is free to prevent particular remappings by throwing a NamespaceException . On the other hand, taking a prefix that is already assigned to a URI and re-assigning it to a new URI in effect unregisters that URI. Therefore, the same restrictions apply to this operation as to <code>NamespaceRegistry.unregisterNamespace</code> : <ul style="list-style-type: none"> Attempting to re-assign a built-in prefix (<code>jcr</code>, <code>nt</code>, <code>mix</code>, <code>sv</code>) to a new URI will throw a NamespaceException.

	<ul style="list-style-type: none"> • Attempting to re-assign a prefix that is currently assigned to a URI that is present in content (either within an item name or within the value of a NAME or PATH property) will throw a NamespaceException. This includes prefixes in use within in-content node type definitions. • Attempting to assign any prefix beginning with "xml". These prefixes are reserved by the XML specification. • An implementation may prevent the re-assignment of any other namespace prefixes for implementation-specific reasons by throwing a NamespaceException. <p>In a level 1 implementation, this method always throws an UnsupportedRepositoryOperationException.</p> <p>A RepositoryException is thrown if another error occurs.</p>
void	<p>unregisterNamespace(String prefix)</p> <p>Removes a namespace mapping from the registry. The following restrictions apply:</p> <ul style="list-style-type: none"> • Attempting to unregister a built-in namespace (jcr, nt, mix, sv) will throw a NamespaceException. • Attempting to unregister a namespace that is currently present in content (either within an item name or within the value of a NAME or PATH property) will throw a NamespaceException. This includes prefixes in use within in-content node type definitions. • An attempt to unregister a namespace that is not currently registered will throw a NamespaceException. • An implementation may prevent the unregistering of any other namespace for implementation-specific reasons by throwing a NamespaceException. <p>In a level 1 implementation, this method always throws an UnsupportedRepositoryOperationException.</p> <p>A RepositoryException is thrown if another error occurs.</p>

Once registered, a prefix can be used in the name of any node or property in the repository. The prefix serves as shorthand for the URI to which it is mapped. Because the space of URIs is universal managed, the combination of the per-repository namespace and the larger URI namespace can be used to provide universal uniqueness of node or property names. Of course, just as in the case of XML namespaces, ensuring this universal uniqueness

requires applications to map their application-specific prefixes to URIs that are uniquely identified with that particular application.

The namespace registry always contains at least the following built-in mappings, which cannot be removed through the API:

- **jcr** -> <http://www.jcp.org/jcr/1.0>
Reserved for items defined within built-in node types. For example **jcr:content**.
- **nt** -> <http://www.jcp.org/jcr/nt/1.0>
Reserved for the names of built-in primary node types.
- **mix** -> <http://www.jcp.org/jcr/mix/1.0>
Reserved for the names of built-in mixin node types.
- **sv** -> <http://www.jcp.org/jcr/sv/1.0>
Reserved for use in the system view XML serialization format.
- **""** (the empty prefix) -> **""**(the empty URI)
This makes the *default namespace* the *empty URI*. In effect this means that a name without a prefix is identical in both its prefixed form and in its fully qualified form (i.e. when it is stored internally as *URI plus local name*). See [6.3.3 Internal Storage of Names and Values](#).

7.3 Importing Repository Content

Level 2 repositories must support the import of content from either of the standard XML mappings, system view and document view (see [6.4 XML Mappings](#)).

7.3.1 Import from System View

Given a system view XML document the subtree constructed upon import is determined by reversing the mapping discussed in [6.4.1 System View XML Mapping](#). The mapping is largely straightforward (though see [7.3.3 Respecting Property Semantics](#) and [7.3.6 Importing jcr:root](#)).

7.3.2 Import from Document View

One of the applications for which the document view mapping is designed is to allow the import of arbitrary XML into a JCR repository (the other application is to provide a context in which XPath queries are more readable than they would be in system view, see [6.6.3 XPath over Document View](#)). 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. The namespace declarations of the incoming XML document are added to the repository namespace registry.

2. Each XML element **E** becomes a JCR node of the same name, **E**.
3. At any point during import an exception may be thrown if the implementation respects a
4. Each child XML element **c** of XML element **E** becomes a child JCR node **c** of JCR node **E**.
5. Each XML attribute **A** within an XML element **E** becomes a JCR **STRING** property **A** of JCR node **E** (but see [7.3.3 Respecting Property Semantics](#)).
6. The value of each XML attribute **A** becomes the value of the corresponding JCR property **A**. The value is always of type **STRING**.
7. Escape sequences representing non-XML-valid characters in element names and whitespace in attribute values may be encountered if the incoming XML stream is the product of an earlier document view export (see [6.4.2 Document View XML Mapping](#)). In these cases, whether the escape sequences are decoded is left up to the implementation. (Note that the predefined entity references **&**, **<**, **>**, **'** and **"**, as well as all other entity and character references, must be decoded in any case, in accordance with the XML specification).
8. Text within an XML element **E** becomes a **STRING** property called **jcr:xmlcharacters** of a node called **jcr:xmltext** (of primary node type **nt:unstructured**), which itself becomes a child node of the node **E**.
9. If import is done through the **ContentHandler** returned by **getImportContentHandler**, the value of **E/jcr:xmltext/jcr:xmlcharacters** will be the character data passed to **ContentHandler.characters**. Data passed to **ContentHandler.ignorableWhitespace** is ignored. If import is done through **importXML**, pure whitespace between elements (that is, 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**.
10. An XML element can have a child element and an attribute with the same name while a JCR node cannot have a child node and property with the same name. For example, **** would imply a JCR node with one property called **b** and one child node also called **b**, which is not allowed. Therefore if such a fragment of XML is encountered on import it is an implementation issue as to how to deal with it.

7.3.2.1 Roundtripping

Upon document view import, a JCR repository will automatically add repository metadata in the form of extra properties (at least `jcr:primaryType`, for example) if these are not already present in the incoming XML, and of course, in the case of an arbitrary XML document, they will not be.

When re-exported using document view, the resulting XML will contain these extra properties in the form of XML attributes that may not have been present in the imported XML. As a result, if roundtripping of arbitrary XML is desired, the application must take care of stripping out unwanted repository-related meta-data like the `jcr:primaryType`.

7.3.2.2 Example

The following XML document

```
<?xml version="1.0" encoding="UTF-8"?>
<myapp:document xmlns:myapp="http://mycorp.com/myapp"
  myapp:title="JSR 170"
  myapp:lead="Content Repository">
  <myapp:body>
    <myapp:paragraph myapp:title="Node Types">
      myapp:text="An important feature..."</>
    </myapp:paragraph>
  </myapp:body>
</myapp:document>
```

when imported in document view would result in the addition of the following mapping to the repository namespace registry:

`myapp -> http://mycorp.com/myapp`

and the creation of the following subtree

Node

Property = "value"

```
myapp:document
├─ jcr:primaryType = "nt:unstructured"
├─ myapp:title = "JSR 170"
├─ myapp:lead = "Content Repository"
├─ myapp:body
│   └─ jcr:primaryType = "nt:unstructured"
│       └─ myapp:paragraph
│           ├── jcr:primaryType = "nt:unstructured"
│           ├── myapp:title = "Node Types"
│           └─ myapp:text = "An important feature..."
```

7.3.3 Respecting Property Semantics

During either system or document view import elements (in system view) or attributes (in document view) may be encountered that correspond to properties with repository-level semantics. This includes, in particular, the properties that are defined in `nt:base`, `mix:lockable`, `mix:referenceable` or `mix:versionable`.

When an element or attribute representing such a property is encountered, an implementation may either *skip* it or *respect* it.

To respect it means to import it and alter the internal state of the repository in accordance with the semantics of the property in the given implementation. For example, to respect `jcr:primaryType` means to attempt to create a node of the indicated primary node type. If the node type in question is not supported by the implementation, an exception is thrown (see [7.3.4 Workspace Import Methods](#) for details).

To skip the element or attribute means not to import it all. *It does not mean to import it but then ignore its semantic implications.*

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`.

A special case occurs if an implementation chooses to skip `jcr:primaryType`. In that case the node type of the imported node becomes `nt:unstructured`. If an implementation does not support the node type `nt:unstructured` then it must respect `jcr:primaryType`.

7.3.4 Workspace Import Methods

The `Workspace` interface provides the following methods for importing content into the repository:

javax.jcr. Workspace	
ContentHandler	<pre>getImportContentHandler(String parentAbsPath, int uuidBehavior)</pre> <p>Returns a <code>org.xml.sax.ContentHandler</code> which can be used to push SAX events into the repository. If the incoming XML stream (in the form of SAX events) does not appear to be a JCR <i>system view</i> XML document then it is interpreted as a JCR <i>document view</i> XML document.</p> <p>The incoming XML is deserialized into a subtree of items immediately below the node at <code>parentAbsPath</code>.</p> <p>The special properties <code>jcr:primaryType</code> and <code>jcr:mixinTypes</code> are taken into account during deserialization in order to determine the node types</p>

	<p>of the newly created nodes.</p> <p>This method simply returns the <code>ContentHandler</code> without altering the state of the repository; the actual deserialization is done through the methods of the <code>ContentHandler</code>. Invalid XML data will cause the <code>ContentHandler</code> to throw a <code>SAXException</code>.</p> <p>As SAX events are fed into the <code>ContentHandler</code>, changes are made directly at the workspace level, without going through the <code>session</code>. As a result, there is not need to call <code>save</code>. The advantage of this direct-to-workspace method is that a large import will not result in a large cache of pending nodes in the <code>session</code>. The disadvantage is that structures that violate node type constraints cannot be imported, fixed and then saved. Instead, a constraint violation will cause the <code>ContentHandler</code> to throw a <code>SAXException</code>. See <code>Session.getImportContentHandler</code> for a version of this method that <i>does</i> go through the <code>session</code>.</p> <p>The flag <code>uuidBehavior</code> governs how the UUIDs of incoming (deserialized) nodes are handled. There are four options:</p> <ul style="list-style-type: none"> • IMPORT_UUID_CREATE_NEW: Incoming referenceable nodes are assigned newly created UUIDs upon addition to the workspace. As a result UUID collisions never occur. • IMPORT_UUID_COLLISION_REMOVE_EXISTING: If an incoming referenceable node has the same UUID as a node already existing in the workspace then the already existing node (and its subtree) 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 subtree is being written. • IMPORT_UUID_COLLISION_REPLACE_EXISTING: If an incoming referenceable node has the same UUID 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 subtree being
--	--

	<p>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 UUID as an existing node elsewhere in the workspace.</p> <ul style="list-style-type: none"> • IMPORT_UUID_COLLISION_THROW: If an incoming referenceable node has the same UUID as a node already existing in the workspace then a SAXException is thrown by the ContentHandler during deserialization. <p>A SAXException will be thrown by the returned ContentHandler during deserialization if the topmost element of the incoming XML would deserialize to a node with the same name as an existing child of parentAbsPath and that child does not allow same-name siblings.</p> <p>A SAXException will also be thrown by the returned ContentHandler during deserialization if uuidBehavior is set to IMPORT_UUID_COLLISION_REMOVE_EXISTING and an incoming node has the same UUID as the node at parentAbsPath or one of its ancestors.</p> <p>A PathNotFoundException is thrown if no node exists at parentAbsPath.</p> <p>A ConstraintViolationException is thrown if the new subtree cannot be added to the node at parentAbsPath due to node-type or other implementation-specific constraints, and this can be determined before the first SAX event is sent.</p> <p>Unlike session.getImportContentHandler, this method will also enforce node type constraints by having the returned ContentHandler throw a SAXException during deserialization.</p> <p>A VersionException is thrown if the node at parentAbsPath is versionable and checked-in, or is non-versionable but its nearest versionable ancestor is checked-in.</p> <p>A LockException is thrown if a lock prevents the addition of the subtree.</p> <p>A RepositoryException is thrown if another error</p>
--	---

	occurs.
void	<p>importXML(String parentAbsPath, InputStream in, int uuidBehavior)</p> <p>Deserializes an XML document and adds the resulting item subtree as a child of the node at parentAbsPath.</p> <p>If the incoming XML stream does not appear to be a JCR <i>system view</i> XML document then it is interpreted as a <i>document view</i> XML document.</p> <p>The special properties jcr:primaryType and jcr:mixinTypes are taken into account during deserialization in order to determine the node types of the newly created nodes.</p> <p>Changes are made directly at the workspace level, without going through the session. As a result, there is not need to call save. The advantage of this direct-to-workspace method is that a large import will not result in a large cache of pending nodes in the session. The disadvantage is that invalid data cannot be imported, fixed and then saved. Instead, invalid data will cause this method to throw an InvalidSerializedDataException. See Session.importXML for a version of this method that <i>does</i> go through the session.</p> <p>The flag uuidBehavior governs how the UUIDs of incoming (deserialized) nodes are handled. There are four options:</p> <ul style="list-style-type: none"> • IMPORT_UUID_CREATE_NEW: Incoming referenceable nodes are assigned newly created UUIDs upon addition to the workspace. As a result UUID collisions never occur. • IMPORT_UUID_COLLISION_REMOVE_EXISTING: If an incoming referenceable node has the same UUID as a node already existing in the workspace then the already existing node (and its subtree) 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 subtree is being

	<p>written.</p> <ul style="list-style-type: none"> • IMPORT_UUID_COLLISION_REPLACE_EXISTING: If an incoming referenceable node has the same UUID 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 subtree being disaggregated and “spread around” to different locations in the workspace. In the most extreme edge 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 UUID as an existing node elsewhere in the workspace. • IMPORT_UUID_COLLISION_THROW: If an incoming referenceable node has the same UUID as a node already existing in the workspace then an ItemExistsException is thrown. <p>An ItemExistsException will be thrown if the topmost element of the incoming XML would deserialize to a node with the same name as an existing child of parentAbsPath and that child does not allow same-name siblings.</p> <p>An IOException is thrown if an I/O error occurs.</p> <p>If no node exists at parentAbsPath, a PathNotFoundException is thrown.</p> <p>If deserialization would overwrite an existing item, an ItemExistsException is thrown.</p> <p>If node-type or other implementation-specific constraints prevent the addition of the subtree, a ConstraintViolationException is thrown.</p> <p>A ConstraintViolationException will also be thrown if uuidBehavior is set to IMPORT_UUID_COLLISION_REMOVE_EXISTING and an incoming node has the same UUID as the node at parentAbsPath or one of its ancestors.</p> <p>A VersionException is thrown if the node at parentAbsPath is versionable and checked-in, or is non-versionable but its nearest versionable ancestor is checked-in.</p>
--	--

	<p>A LockException is thrown if a lock prevents the addition of the subtree.</p> <p>If another error occurs, a RepositoryException is thrown.</p>
--	---

7.3.5 Session Import Methods

The **session** contains the following methods for importing and exporting content:

javax.jcr. Session	
ContentHandler	<p>getImportContentHandler(String parentAbsPath)</p> <p>Returns a org.xml.sax.ContentHandler which can be used to push SAX events into the repository. If the incoming XML stream (in the form of SAX events) does not appear to be a JCR <i>system view</i> XML document then it is interpreted as a JCR <i>document view</i> XML document.</p> <p>The incoming XML is deserialized into a subtree of items immediately below the node at parentAbsPath.</p> <p>The special properties jcr:primaryType and jcr:mixinTypes are taken into account during deserialization in order to determine the node types of the newly created nodes.</p> <p>This method simply returns the ContentHandler without altering the state of the repository; the actual deserialization is done through the methods of the ContentHandler. Invalid XML data will cause the ContentHandler to throw a SAXException.</p> <p>As SAX events are fed into the ContentHandler, the tree of new items is built in the transient storage of the session. In order to persist the new content, save must be called. The advantage of this through-the-session method is that structures that violate node type constraints can be imported, fixed and then saved. The disadvantage is that a large import will result in a large cache of pending nodes in the session. See Workspace.getImportContentHandler for a version of this method that <i>does not</i> go through the session.</p> <p>A PathNotFoundException is thrown if no node exists at parentAbsPath.</p> <p>A ConstraintViolationException is thrown if the new subtree cannot be added to the node at parentAbsPath.</p>

	<p>due to node-type or other implementation-specific constraints, and this can be determined before the first SAX event is sent. Unlike <code>Workspace.getImportContentHandler</code>, this method does not enforce node type constraints by throwing <code>SAXExceptions</code> during deserialization. These constraints are checked on <code>save</code>.</p> <p>A <code>VersionException</code> is thrown if the node at <code>parentAbsPath</code> is versionable and checked-in, or is non-versionable but its nearest versionable ancestor is checked-in.</p> <p>A <code>LockException</code> is thrown if a lock prevents the addition of the subtree.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>
void	<p><code>importXML(String parentAbsPath, InputStream in)</code></p> <p>Deserializes an XML document and adds the resulting item subtree as a child of the node at <code>parentAbsPath</code>.</p> <p>If the incoming XML stream does not appear to be a JCR <i>system view</i> XML document then it is interpreted as a <i>document view</i> XML document.</p> <p>The special properties <code>jcr:primaryType</code> and <code>jcr:mixinTypes</code> are taken into account during deserialization in order to determine the node types of the newly created nodes.</p> <p>The tree of new items is built in the transient storage of the <code>session</code>. In order to persist the new content, <code>save</code> must be called. The advantage of this through-the-session method is that invalid data can be imported, fixed and then saved. The disadvantage is that a large import will result in a large cache of pending nodes in the <code>session</code>. See <code>Workspace.importXML</code> for a version of this method that <i>does not</i> go through the <code>session</code>.</p> <p>An <code>java.io.IOException</code> is thrown if an I/O error occurs.</p> <p>If no node exists at <code>parentAbsPath</code>, a <code>PathNotFoundException</code> is thrown.</p> <p>If deserialization would overwrite an existing item, an <code>ItemExistsException</code> is thrown.</p> <p>A <code>ConstraintViolationException</code> is thrown if the new subtree cannot be added to the node at <code>parentAbsPath</code>.</p>

	<p>due to node-type or other implementation-specific constraints, and this can be determined before deserialization begins. Unlike <code>Workspace.getImportContentHandler</code>, this method does not enforce node type constraints during deserialization. These constraints are checked on <code>save</code>.</p> <p>A <code>VersionException</code> is thrown if the node at <code>parentAbsPath</code> is versionable and checked-in, or is non-versionable but its nearest versionable ancestor is checked-in.</p> <p>A <code>LockException</code> is thrown if a lock prevents the addition of the subtree.</p> <p>If another error occurs a <code>RepositoryException</code> is thrown.</p>
--	--

7.3.6 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` (see [6.4 XML Mappings](#)). In addition, if the root node has a UUID (in many implementations it will, see [4.9 Referenceable Nodes](#)) this will also be recorded in the serialization.

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 UUID as the existing root node is encountered on import.

IMPORT_UUID_CREATE_NEW: The XML element representing `jcr:root` is rendered as a normal JCR node at the position specified (with the name `jcr:root`). It gets a new UUID, 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 UUID 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 according to UUID (plus relative path, for non-referenceables). In both cases the result is that the root node of the workspace will be replaced along with its subtree (i.e., the whole workspace), just as if the root node had been altered through the normal `getNode`, `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.

7.4 Assigning Node Types

Level 2 JCR implementations must support the assignment of primary and mixin node types to nodes upon creation and, optionally, the assignment and removal of mixin node types from existing nodes.

7.4.1 The Special Properties *jcr:primaryType* and *jcr:mixinTypes*

When a node is created, its `jcr:primaryType` property is automatically created and set to the name of the assigned primary node type. When a mixin type is assigned, its name is added to the multi-valued `jcr:mixinTypes` property, which is created if it does not yet exist.

7.4.2 Assigning a Primary Node Type

Assignment of a node type to a node on creation is done by supplying the node type name alongside the new node's path in a call to

```
Node.addNode(String relPath, String primaryNodeTypeName)
```

(see [7.1.3 Adding Nodes](#)).

Alternatively, in many cases the client of the API will not need to explicitly supply a node type since the very name of the new child node will be enough to unambiguously determine its node type by reference to one of the node definitions contained in the node type of the parent node. In such cases, `Node.addNode(String relPath)` will be sufficient.

Automatic determination of node types is only required to work if the name of the node being added is explicitly named in a child node definition of the parent node type, (or one of that type's supertypes) (see [6.7.14 NodeDef](#)). The implementation is not required to take residual definitions into account (see [6.7.15 Residual Definitions](#)).

If the node type of the new child node cannot be determined automatically and no primary node type is explicitly specified, then a `ConstraintViolationException` is thrown (see [7.1.3 Adding Nodes](#)).

7.4.3 Assigning Mixin Node Types

To assign a mixin type, the method `Node.addMixin(String mixinName)` is used. The mixin type adds additional child node or property definitions to a node (i.e., either permitting or requiring additional child nodes or properties).

Conflicts with other mixin node types or with the primary node type that are detected immediately will cause a `ConstraintViolationException` to be thrown on the `addMixin` call. Conflicts detected upon `save` will cause a `ConstraintViolationException` to be thrown at that time. Which conflicts are detected at which stage may differ across implementations.

Note that the *orderable child nodes status* of a mixin node type has no effect, so it will never conflict with the orderable child nodes status of the primary node type.

In some implementations it may be possible to add mixin types to a node only before the first `save` of that node (in effect, at node creation). Other implementations may support dynamic addition, and possibly removal, of mixin node types during a node's lifetime. The method `Node.removeMixin` is provided for those cases that support dynamic removal. If an implementation does not support dynamic addition or removal, the `addMixin` or `removeMixin` method will throw a `ConstraintViolationException`.

javax.jcr. Node	
void	addMixin(String mixinName) Adds the specified mixin node type to this node. Also adds <code>mixinName</code> to this node's <code>jcr:mixinTypes</code> property. The mixin node type assignment takes effect on <code>save</code> . If a conflict with another assigned mixin or the primary node type is detected, a <code>ConstraintViolationException</code> is thrown immediately. As well, mixin-related conflicts detected on <code>save</code> will cause that method to throw a <code>ConstraintViolationException</code> . Which conflicts are detected at which stage (immediately or on <code>save</code>) may differ across implementations. A repository can also prevent the addition of a mixin for implementation-specific reasons by throwing a

	<p>ConstraintViolationException.</p> <p>In some implementations it may only be possible to add mixin types before a node is first saved, and not after. In such cases any later calls to addMixin will throw a ConstraintViolationException.</p> <p>If the specified mixin node type is not recognized a NoSuchNodeTypeException is thrown.</p> <p>A VersionException is thrown if this node is versionable and checked-in, or is non-versionable but its nearest versionable ancestor is checked-in.</p> <p>A LockException is thrown if a lock prevents the addition of the mixin.</p> <p>A RepositoryException will be thrown if another error occurs.</p>
void	<p>removeMixin(String mixinName)</p> <p>Removes the specified mixin node type from this node. . Also removes mixinName from this node's jcr:mixinTypes property. The mixin node type removal takes effect on save.</p> <p>If this node does not have the specified mixin, a NoSuchNodeTypeException is thrown.</p> <p>A ConstraintViolationException will be thrown if the removal of a mixin is not allowed (implementations are free to enforce any policy they like with regard to mixin removal).</p> <p>A VersionException is thrown if this node is versionable and checked-in or is non-versionable but its nearest versionable ancestor is checked-in.</p> <p>A LockException is thrown if a lock prevents the removal of the mixin.</p> <p>A RepositoryException will be thrown if another error occurs.</p>
boolean	<p>canAddMixin(String mixinName)</p> <p>Returns true if the specified mixin node type, mixinName, can be added to this node. Returns false otherwise. Addition of a mixin can be prevented for any of the following reasons:</p> <ul style="list-style-type: none"> • The mixin's definition conflicts with the existing primary node type or one of the existing mixin

	<p>node types.</p> <ul style="list-style-type: none"> • The node is versionable and checked-in or is non-versionable and its nearest versionable ancestor is checked-in. • The addition is prevented because this node is <i>protected</i> (as defined in this node's NodeDef, found in this node's parent's node type). • The addition is prevented due to access control restrictions. • The addition is prevented due to a lock. • The addition is prevented for implementation-specific reasons. <p>A RepositoryException will be thrown if an error occurs.</p>
--	---

7.4.4 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 deserialized nodes of any mixin node types that the repository does not support.

7.4.5 Serialization and Node Types

When deserializing content from another JCR repository, each imported node will come with its attached **jcr:primaryType** and **jcr:mixinTypes** properties. This information will be used while deserializing to validate the node according to the specified node types (and to do whatever internal bookkeeping the implementation requires in terms of noting the node types of the incoming nodes).

Any *primary* node types referenced by the imported content will have to be already registered with the target repository. This implies that it will be necessary to first import and register those node types referenced by the content that are not already registered with the target repository.

Any mixin node types referenced by the imported content will either have to be already present (implying the necessity of ensuring this by importing these definitions first, as above) or be stripped from the incoming node, in accordance with [7.4.4 Automatic Addition and Removal of Mixins](#).

Though this specification does not attempt to define the details of the process of importing node type definitions, the fact that node type definitions may themselves be stored as normal content (see [6.7.22.10](#) *nt:nodeType*) means that the standard serialization/deserialization mechanism can be used to export and import their definitions (see [6.5](#) *Exporting Repository Content* and [7.3](#) *Importing Repository Content*). Actually registering them is, as mentioned, outside the scope of this specification (see [6.7.1](#) *Node Type Configuration*).

8 Optional Repository Features

This section provides an overview of optional features that may be supported by a JCR implementation. These are: Transactions, Versioning, Observation, Access Control and Locking. None of these features have any dependencies on each other or on any level 2 feature, therefore any combination of these five may be supported by either a level 1 or level 2 repository.

Like the sections above, this section is arranged into topics based on functional categories. For an overview of the specification by Java interface, please consult the accompanying Javadoc.

8.1 Transactions

A JCR compliant repository may support transactions. If it does so, it must adhere to the Java Transaction API (JTA) specification (see <http://java.sun.com/products/jta/index.html>).

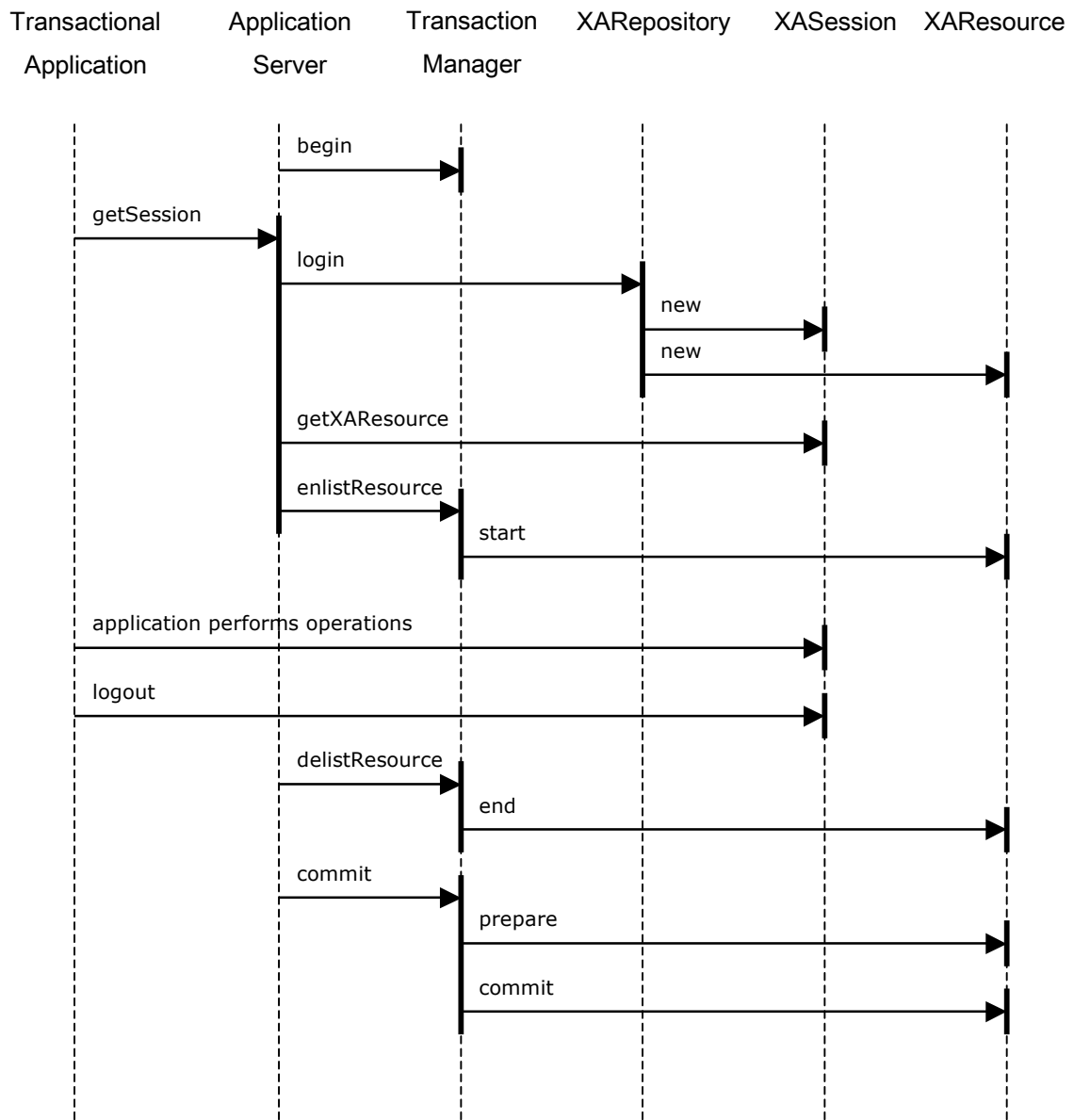
The actual methods used to control transaction boundaries are not defined by the JCR specification (that is why there are no `begin`, `commit` or `rollback` methods in this API). These methods are defined by the JTA specification.

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 client of the JCR 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 client of the JCR 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 JCR implementation must support both of these approaches.

8.1.1 Container Managed Transactions: Sample Request Flow



8.1.2 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();
```

8.1.3 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.

In order to avoid the awkwardness of qualifying every statement about `save` with the phrase “*unless the operation occurs within a transaction*” we simply assume no transactions throughout most of the specification and note the qualification here.

8.2 Versioning

A JCR compliant repository may support versioning. This feature allows the state of a node to be recorded in such a way that it can later be restored. The JCR versioning system is modelled after the Workspace Versioning and Configuration Management (WVCM) API defined by JSR 147.

A versioning repository has, in addition to one or more workspaces, a special *version storage* area. The version storage consists of *version histories*. Versionable nodes in different workspaces share the same version history if and only if they have the same UUID (see [4.10.2 Multiple Workspaces and Corresponding Nodes](#)).

A version history is a collection of versions connected to one another by the *successor* relationship. A new version is added to the version history of a versionable node when one of its workspace instances is *checked-in*. Every new version is attached to the version history as the successor of one (or more) of the existing versions. The result is that a version history is a directed acyclic graph of versions, where the arcs in the graph represent the successor relation.

The version storage objects are themselves defined as nodes. Though there is only one version storage per repository, the version storage data is reflected in each workspace as a special, protected, sub-tree of nodes of types `nt:versionHistory` and `nt:version` (see [8.2.2 Version Storage](#)).

When a versionable node is *checked-in* (using `Node.checkin`) a new version is created in the version history of that node. The versionable node is also set to be read-only. In order to alter it with a regular write method, it must be checked-out (using `Node.checkout`). A versionable node can also be restored to the state recorded in one of its versions using `Node.restore`.

8.2.1 Versionable Nodes

To be *versionable*, a node must have `mix:versionable` as one of its mixin node types. Recall from [6.7.21.3 mix:versionable](#), that this node type has the following definition:

```
NodeTypeName
  mix:versionable
Supertypes
  mix:referenceable
IsMixin
  true
HasOrderableChildNodes
  false
PrimaryItemName
  null
PropertyDef
  Name jcr:versionHistory
  RequiredType REFERENCE
  ValueConstraints ["nt:versionHistory"]
  DefaultValues null
  AutoCreate true
  Mandatory true
  OnParentVersion COPY
  Protected true
  Multiple false
PropertyDef
  Name jcr:baseVersion
  RequiredType REFERENCE
  ValueConstraints ["nt:version"]
  DefaultValues null
  AutoCreate true
  Mandatory true
  OnParentVersion IGNORE
  Protected true
```



```

    Multiple false
PropertyDef
  Name jcr:isCheckedOut
  RequiredType BOOLEAN
  ValueConstraints []
  DefaultValues [true]
  AutoCreate true
  Mandatory true
  OnParentVersion IGNORE
  Protected true
  Multiple false
PropertyDef
  Name jcr:predecessors
  RequiredType REFERENCE
  ValueConstraints [nt:version]
  DefaultValues null
  AutoCreate true
  Mandatory true
  OnParentVersion COPY
  Protected true
  Multiple true
PropertyDef
  Name jcr:mergeFailed
  RequiredType REFERENCE
  ValueConstraints []
  DefaultValues null
  AutoCreate true
  Mandatory false
  OnParentVersion ABORT
  Protected true
  Multiple true

```

As the definition indicates, **mix:versionable** is a subtype of **mix:referenceable** which mandates the property **jcr:uuid**, exposing a universally unique identifier for the node (see [6.7.21.2](#), *mix:referenceable*). The result is that all versionable nodes are guaranteed to have a UUID.

In addition to this inherited property, a **mix:versionable** node has the properties **jcr:versionHistory**, **jcr:baseVersion**, **jcr:isCheckedOut**, **jcr:predecessors** and **jcr:mergeFailed**.

jcr:versionHistory is a **REFERENCE** property which points to the **nt:versionHistory** node that holds as its children the **nt:version** nodes that make up this versionable node's version history. Note that the UUID of the **nt:versionHistory** node is different from the UUID shared by the set of corresponding versionable nodes (at most one per workspace) that it serves.

jcr:baseVersion is also a **REFERENCE** property. It points to the current base version of this node. The base version is one of the **nt:version** nodes within the version history pointed to by **jcr:versionHistory**, above. The base version (like all versions) is an **nt:version** node, and this property stores the UUID of *that* node. Again, the UUID of the version node is different from that shared by the set of corresponding versionable nodes in the workspaces, and from all other version nodes.

`jcr:isCheckedOut` is a **BOOLEAN** property that records whether this versionable node is checked-out or checked-in. When a versionable node is in the checked-in state, it is *read-only*, and cannot be altered by the regular write methods of the API. Note that this status is distinct from the node type-enforced *protected* status. When a versionable node is checked-out it can (if it is not protected) be altered by the API write methods. The checked-out status provides an indicator to other sessions on the same workspace telling them when a particular versionable node is “being worked on”. The read-only status enforced when a versionable node is checked-in propagates to all its non-versionable descendants. When a versionable node is checked in, it and its non-versionable subtree become read-only; when it is checked-out, it and its non-versionable subtree lose their read-only status.

`jcr:predecessors` is a multi-value **REFERENCE** property that points to one or more versions within the version graph of the version history pointed by `jcr:versionHistory`. These versions are those that are currently scheduled to become the predecessors of this versionable node when it is checked-in (and recorded in a version of its own).

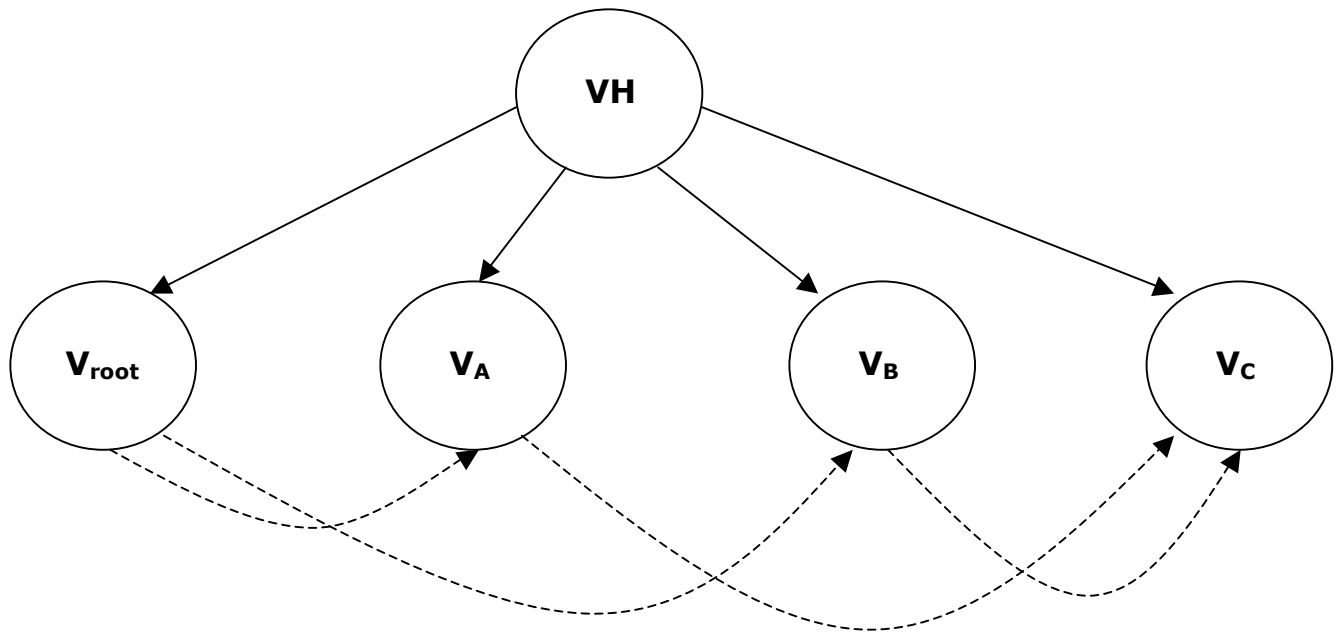
`jcr:mergeFailed` is a multi-value **REFERENCE** property that is used in the context of the `merge` method. A `merge` compares the base version of *this* versionable node with the base version of its corresponding node in some other workspace. If the system can determine which base version is a successor of the other, then it either leaves this versionable node alone (if *this* node's base version is the successor of the other node's base versions) or updates it to reflect the corresponding node (if that node's base version is a successor of *this* node's base version). In cases where the system cannot determine which node is the successor, the merge is said to “fail”. When that happens, a reference to the base version of the corresponding node in the other workspace is added to *this* node's `jcr:mergeFailed` property, thus keeping a record of which nodes could not be merged, and therefore allowing the client application to deal with these nodes appropriately. See [8.2.10 Merge](#), for more details.

All of these properties that store the versioning-related meta-data are protected (though of course the versionable node itself, and its other application-specific subitems may or may not be protected). This guarantees that the client cannot alter the meta-data values; they are maintained by the repository implementation itself.

8.2.2 Version Storage

A version history consists of a single `nt:versionHistory` node with a set of immediate child nodes all of type `nt:version`, representing all the versions within that version history.

An `nt:versionHistory` has at least one child, the `nt:version` node representing the *root version*. From the root version the version graph proceeds through a network of **REFERENCE** properties linking any additional child `nt:version` nodes into a version graph defining the successor relations among the versions. The version graph within any given version history must include all and only the children of that version history's `nt:versionHistory` node. The following diagram illustrates a single version history:



The solid arrows represent parent node to child node relations while the dotted arrows represent the successor relations between versions, implemented through **REFERENCE** properties.

Here we see an `nt:versionHistory` node, `VH` with child `nt:version` nodes `Vroot`, `VA`, `VB` and `VC`. The version graph begins at `Vroot`, which has successors `VA` and `VB`, both of which, in turn, have `VC` as their respective successor.

8.2.2.1 `jcr:versionStorage`

The full set of version histories in the version storage, though stored in a single location in the repository, must be reflected in each workspace as a subtree below the node `/jcr:system/jcr:versionStorage`. This subtree must be read-only. That is, clients cannot alter this subtree through standard write methods; though the implementation can, of course, alter it as a side-effect of the client calling version-related methods.

The read-only status of this subtree should be enforced by the implementation as a matter of access control. As a result, the

protected status of parts of this subtree (enforced as a matter of node type constraints) is not relevant since, in effect, the entire subtree is protected.

Though the general repository-wide version history is reflected in each workspace, the access that a particular `session` gets to that subtree is governed by that `session`'s authorization (which is determined either by the `session`'s `Credentials` or an external authorization mechanism, see [8.4 Access Control](#)), just as it is for any other part of the workspace.

All `nt:versionHistory` nodes are found under `/jcr:system/jcr:versionStorage`, though there may be a structure of intervening subnodes that sort the version histories by some implementation-specific criteria.

The node type of the node `jcr:versionStorage` is `nt:unstructured` (see [6.7.22.4](#), *nt:unstructured*, above).

8.2.2.2 Searching and Traversing Version Storage

Exposing the 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.

This allows, for example, a client to search for a particular version according to the value of its properties. In a repository that supports SQL queries, the following query would return all versions where `productName` is "Car" and `price` is greater than 30,000:

```
SELECT *
FROM nt:version
WHERE productName = "Car"
      AND price > "30000"
      AND jcr:path LIKE
          "/jcr:system/jcr:versionStorage/%"
```

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 client to then cast the returned object down to either `Version` or `VersionHistory` and then use it in methods that take those types, for example `Node.restore(Version version, boolean removeExisting)`.

8.2.2.3 nt:versionHistory

The `nt:versionHistory` node type has the following definition (repeated from [6.7.22.13](#) *nt:versionHistory*):

```
NodeTypeNames
nt:versionHistory
```

```

Supertypes
  nt:base
  mix:referenceable
IsMixin
  false
HasOrderableChildNodes
  false
PrimaryItemName
  null
PropertyDef
  Name jcr:versionableUuid
  RequiredType STRING
  ValueConstraints []
  DefaultValues null
  AutoCreate true
  Mandatory true
  OnParentVersion ABORT
  Protected true
  Multiple false
ChildNodeDef
  Name jcr:rootVersion
  RequiredPrimaryTypes [nt:version]
  DefaultPrimaryType nt:version
  AutoCreate true
  Mandatory true
  OnParentVersion ABORT
  Protected true
  SameNameSibs false
ChildNodeDef
  Name jcr:versionLabels
  RequiredPrimaryTypes [nt:versionLabels]
  DefaultPrimaryType nt:versionLabels
  AutoCreate true
  Mandatory true
  OnParentVersion ABORT
  Protected true
  SameNameSibs false
ChildNodeDef
  Name ""
  RequiredPrimaryTypes [nt:version]
  DefaultPrimaryType nt:version
  AutoCreate false
  Mandatory false
  OnParentVersion ABORT
  Protected true
  SameNameSibs false

```

nt:versionHistory, like all node types, is a subtype of **nt:base**, so it inherits the **jcr:primaryType** and **jcr:mixinTypes** properties.

Additionally, all **nt:versionHistory** nodes must also have mixin type **mix:referenceable**, which means that they have the property **jcr:uuid**.

This node type defines a **STRING** property called **jcr:versionableUuid** that stores the UUID of the versionable node whose version history this is.

It type also mandates a single auto-created subnode called `jcr:rootVersion`. This is a version that serves as the starting point for the version graph; it does not hold any state information (see [8.2.4, Initializing the Version History](#), below).

Every `nt:versionHistory` node also has an auto-created child node called `jcr:versionLabels` of node type `nt: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 which refers to that version within this version history to which the label applies.

All additional versions are added as needed by the versioning system as `nt:version` child nodes. These children are defined by the second *ChildNodeDef*, with name attribute of "*" (i.e., making this a *residual definition*, see [6.7.15, Residual Definitions](#), above). The names of the `nt:version` nodes are left up to the implementation.

8.2.2.4 nt:versionLabels

```

NodeTypeName
  nt:versionLabels
Supertypes
  nt:base
IsMixin
  false
HasOrderableChildNodes
  false
PrimaryItemName
  null
PropertyDef
  Name *
  RequiredType REFERENCE
  ValueConstraints ["nt:version"]
  DefaultValues null
  AutoCreate false
  Mandatory false
  OnParentVersion ABORT
  Protected true
  Multiple false

```

8.2.2.5 nt:version

The `nt:version` node type has the following definition (repeated from [6.7.22.15 nt:version](#)):

```

NodeTypeName
  nt:version
Supertypes
  nt:base
  mix:referenceable
IsMixin
  false
HasOrderableChildNodes
  false

```

```

PrimaryItemName
  null
PropertyDef
  Name jcr:created
  RequiredType DATE
  ValueConstraints []
  DefaultValues null
  AutoCreate true
  Mandatory true
  OnParentVersion ABORT
  Protected true
  Multiple false
PropertyDef
  Name jcr:predecessors
  RequiredType REFERENCE
  ValueConstraints ["nt:version"]
  DefaultValues null
  AutoCreate false
  Mandatory false
  OnParentVersion ABORT
  Protected true
  Multiple true
PropertyDef
  Name jcr:successors
  RequiredType REFERENCE
  ValueConstraints ["nt:version"]
  DefaultValues null
  AutoCreate false
  Mandatory false
  OnParentVersion ABORT
  Protected true
  Multiple true
ChildNodeDef
  Name jcr:frozenNode
  RequiredPrimaryTypes [nt:frozenNode]
  DefaultPrimaryType null
  AutoCreate false
  Mandatory false
  OnParentVersion ABORT
  Protected true
  SameNameSibs false

```

nt:version is a subtype of **nt:base**, so it has the properties **jcr:primaryType** and **jcr:mixinTypes**.

In addition, each **nt:version** node must also have the mixin type **mix:referenceable**, providing it with a **jcr:uuid** property.

Additionally it has:

- **jcr:created**: This property records the date and time that the version was created.
- **jcr:predecessors**: A multi-value **REFERENCE** property that points to the immediate predecessors of this version in the version history.

- **jcr:successors**: A multi-value **REFERENCE** property that points to the immediate successors of this version in the version history.

These properties store the meta-data that is needed by the repository to manage the version. In addition to these properties, of course, the version entity must also store the actual state of the node that was versioned to produce it. This is done by storing a “frozen” copy of the versionable node in the form of a special child node of the version node, called **jcr:frozenNode**:

- **jcr:frozenNode**: A child node of type **nt:frozenNode** which holds the actual state of the versionable node at the time that this version was created.

8.2.2.6 nt:frozenNode

```

NodeTypeName
    nt:frozenNode
Supertypes
    nt:base
    mix:referenceable
IsMixin
    false
HasOrderableChildNodes
    true
PrimaryItemName
    null
PropertyDef
    Name jcr:frozenPrimaryType
    RequiredType STRING
    ValueConstraints []
    DefaultValues null
    AutoCreate true
    Mandatory true
    OnParentVersion ABORT
    Protected true
    Multiple false
PropertyDef
    Name jcr:frozenMixinTypes
    RequiredType STRING
    ValueConstraints []
    DefaultValues null
    AutoCreate false
    Mandatory false
    OnParentVersion ABORT
    Protected true
    Multiple true
PropertyDef
    Name jcr:frozenUuid
    RequiredType STRING
    ValueConstraints []
    DefaultValues null
    AutoCreate true
    Mandatory true
    OnParentVersion ABORT
    Protected true
    Multiple false
PropertyDef

```



```

Name ""
RequiredType UNDEFINED
ValueConstraints []
DefaultValues null
AutoCreate false
Mandatory false
OnParentVersion ABORT
Protected true
Multiple false
PropertyDef
Name ""
RequiredType UNDEFINED
ValueConstraints []
DefaultValues null
AutoCreate false
Mandatory false
OnParentVersion ABORT
Protected true
Multiple true
ChildNodeDef
Name ""
RequiredPrimaryTypes [nt:base]
DefaultPrimaryType null
AutoCreate false
Mandatory false
OnParentVersion ABORT
Protected true
SameNameSibs true

```

The properties and child nodes of the versioned node (call it *N*) are dealt with according to their respective `OnParentVersion` attribute, as defined in the node type of *N*. Those child nodes and properties of *N* with `OnParentVersion=COPY` are copied to `jcr:frozenNode`. The residual property and child node definitions in `nt:frozenNode` provide the “space” into which these copies are placed.

Those child nodes and properties of *N* with `OnParentVersion=IGNORE` are not copied.

Those versionable child nodes of *N* (i.e., children of *N* that are themselves also versionable) with `OnParentVersion=VERSION` are dealt with in a special way: a node with the same name as the child node but of type `nt:versionedChild` is placed as a child of `jcr:frozenNode`. This special node is not a copy of the child node of *N* but instead holds a single reference property (called `jcr:childVersionHistory`) that points to the version history of the child of *N*. The `OnParentVersion` mechanism has other options as well, for a full discussion, see [8.2.11 The OnParentVersion Attribute](#).

Among the properties of *N* that are copied over to `jcr:frozenNode`, a special exception must be made for `jcr:primaryType`, `jcr:mixinTypes` and `jcr:uuid`. These properties cannot be copied to their corresponding `jcr:frozenNode` node without conflicting with that node’s existing properties of the same name (recall for example, that `jcr:frozenNode` is of type `nt:frozenNode`, and so

its `jcr:primaryType` property will, of course, hold the value `"nt:frozenNode"`, not the node type of `N`). To address this problem, the copies are renamed `jcr:frozenPrimaryType`, `jcr:frozenMixinTypes`, and `jcr:frozenUuid`, respectively.

8.2.2.7 nt:versionedChild

```

NodeTypeName
    nt:versionedChild
Supertypes
    nt:base
IsMixin
    false
HasOrderableChildNodes
    false
PrimaryItemName
    null
PropertyDef
    Name jcr:childVersionHistory
    RequiredType REFERENCE
    ValueConstraints ["nt:versionHistory"]
    DefaultValues null
    AutoCreate true
    Mandatory true
    OnParentVersion ABORT
    Protected true
    Multiple false

```

8.2.2.8 Version Graph

The structure of the version graph is based on the following principles:

- A version graph consists of one or more versions.
- A version graph has exactly one root version.
- The root version does not have a predecessor version.
- All other versions (apart from the root version) have one or more predecessors (merges are allowed).
- Each version may have one or more successors (branches are allowed).
- A version cannot be one of its own successors. The version graph is a directed acyclic graph.

8.2.2.9 Reference Properties within a Version

When a **REFERENCE** property is stored as part of the frozen state of a version, the referential integrity requirement is lifted. For example, given the following situation:

- Nodes **A** and **B** in a workspace `ws` (i.e., in the workspace proper, not in the protected version storage subtree)
- **A** is (at least) versionable.

- **B** is (at least) referenceable.
- **A** has **REFERENCE** property **P**.
- **P** has an *OnParentVersion* setting of **COPY**.
- **P** holds a reference to **B**. **B** has no other references pointing to it.

Let us assume that **A** is checked in.

When **A** is checked in, **P** will be recorded as part of the frozen state of the newly created version **A'** by being copied to version storage as a property **P'** of **A'**.

At this point **B** cannot be removed from the workspace because it has a reference (**P**) pointing to it. However, if **P** is removed from **A**, then **B** can be removed. Because referential integrity is not enforced for frozen reference properties in version storage, the reference from **P'** will not prevent the removal of **B**. This is despite the fact that **P'** does appear in the same workspace as **B** (though only in the special version storage subtree at `/jcr:system/jcr:versionStorage`). Note that this also means that a call to `getReferences` on **B** will *not* return **P'**.

8.2.2.10 Removal of Versions

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 each successor of the removed version becomes a successor of every predecessor of the removed version. Note that allowing remove in this context would not constitute an exception to the requirement that the version storage be protected, since protected status applies to standard write methods (e.g. like `Node.addNode`) and not version-specific methods (like `Node.checkin`) that alter the version history as a *side-effect*.

8.2.3 The Base Version

For a given version history, every versionable node that shares that version history (there being at most one such node per workspace) contains a reference to its particular *base version* within the version history. Among any set of nodes with a common version history, each node may identify a different version as its base version. The base version of a particular node **N** is the one that will serve as the default immediate predecessor of the next version of **N** that is created.

8.2.4 Initializing the Version History

When a new versionable node is created, a new version history is created for it. At first, the version history consists of only the `nt:versionHistory` node and its single child, the `nt:version` node representing the *root version*, which will serve as the starting point, from which the version graph of successors will proceed. The root version does not store any state information; it serves only to make the semantics of subsequent operations consistent. Initially, the root version also serves as the *base version* for the new versionable node.

In terms of actual nodes and properties being created or changed, here is what happens when a new `mix:versionable` node `N` is created in workspace `w1`:

- `N` is created by the call `M.addNode("N")` where `M` is some suitable parent node for `N`.
- Before being saved, `N` is made versionable by the call `N.addMixin("mix:versionable")`. In some implementations, dynamic assignments of mixins may be supported, thus allowing a node to be rendered versionable at any time in its lifecycle, not just upon creation. See [7.4.3 Assigning Mixin Node Types](#).
- On **save** of `N`, a new version history is automatically created for `N`. This means that the repository automatically creates a new node of type `nt:versionHistory` (call it `VH`). `VH` automatically gets a child node of type `nt:version` called `jcr:rootVersion`. Let us call this node `v0`.
- `v0` is the root version of `VH`. This root version does not contain any state information about `N` other than the node type and UUID information in the `jcr:frozenPrimaryType`, `jcr:frozenMixinTypes`, and `jcr:frozenUuid`. It is a dummy version.
- The **REFERENCE** property `jcr:versionHistory` of `N` is initialized to the UUID of `VH`. This constitutes a reference from `N` to its version history.
- The **REFERENCE** property `jcr:baseVersion` of `N` is initialized to the UUID of `v0`. 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 UUID, that of `v0` (the same as `jcr:baseVersion`).

The **BOOLEAN** property `jcr:isCheckedOut` is set to `true`.

8.2.5 Check In

To create a new version of a versionable node **N**, the client calls **N.checkin**. If **N** is already checked-in, this method has no effect but simply returns the current base version of this node. If **N** is not versionable then a **UnsupportedRepositoryOperationException** is thrown. Otherwise, the following preconditions must hold:

- **N** must not have any unsaved changes pending, otherwise an **InvalidItemStateException** is thrown.
- **N**'s **jcr:mergeFailed** (multi-value) property must not be present, otherwise 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).

Given these preconditions, **N.checkin** will cause the following series of events:

- A new **nt:version** node **v** is created and added as a child node to **VH**, the **nt:versionHistory** pointed to by **N**'s **jcr:versionHistory** property.
- **N**'s current **jcr:predecessors** property is copied to **v**, and **N**'s **jcr:predecessors** property is then set to the empty array (it is a multi-value property, therefore it can be set to empty). Note that **N**'s **jcr:predecessors** property also forms part of the frozen state of **N** (because it has an **OnParentVersion** attribute of **COPY**) and therefore will also be copied to **v/jcr:frozenNode**.
- A reference to **v** is added to the **jcr:successors** property of each of the versions identified in **v**'s **jcr:predecessors** property.
- **N**'s **jcr:baseVersion** property is set to refer to **v**.
- **N**'s **jcr:isCheckedOut** property is set to **false**.
- The state of **N** is recorded in the form of the **jcr:frozenNode** child of **v**. The extent of the state stored (i.e. exactly which child items are included and which ignored, etc.) will typically be partial, as prescribed by the **OnParentVersion** attribute of each of **N**'s child items. See [8.2.11 OnParentVersion Attribute](#), for the details. The **jcr:primaryType**, **jcr:mixinTypes** and **jcr:uuid** properties of **N** are copied over to the child **jcr:frozenNode** of **v** but renamed to **jcr:frozenPrimaryType**, **jcr:frozenMixinTypes** and **jcr:frozenUuid** to avoid conflict with **jcr:frozenNode**'s own properties with these names.

- `v` is given a automatically generated name. How this is done is implementation specific.
- The node `N` and its *connected non-versionable subtree* become read-only. `N`'s connected non-versionable subtree is the set of non-versionable descendant nodes reachable from `N` through child links without encountering any versionable nodes. In other words, the read-only status flows down from the checked-in node along every child link until either a versionable node is encountered or an item with no children is encountered.
- Read-only status means that an item cannot be altered by the client using standard API methods (`addNode`, `setProperty`, etc.). The only exceptions to this rule are the `restore`⁷, `Node.merge` and `Node.update` operations; these do not respect read-only status due to check-in. 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).

This method acts directly on the workspace and the version storage. All changes are persisted immediately. There is no need to call `save`.

8.2.6 Check Out

In order to alter a versionable node (and its non-versionable subtree) the node must be checked-out. The checked-out state indicates to the repository and other clients that the current base version (the one pointed to be `jcr:baseVersion`) of `N` is "being worked on" and will (usually) be checked-in again at some point in the future, thus creating a new version. When a versionable node is first created (or an existing node is first made versionable, in those implementations that allow that) it will already be in the checked-out state (its `jcr:checkedOut` property is set to `true`).

To check-out a versionable node `N`, the client calls `N.checkout`. If the node is already checked out, this method has no effect. If `N` is not versionable then an

`UnsupportedRepositoryOperationException` is thrown.

Otherwise, a `N.checkout` will cause the following series of events:

- The current value of `N`'s `jcr:baseVersion` is copied to `N`'s `jcr:predecessors` property.
- `N`'s `jcr:isCheckedOut` property is set to `true`.

⁷ `Workspace.restore`, `Node.restore` (all signatures) and `Node.restoreByLabel`.

- `N` and `N`'s *connected non-versionable subtree* lose their read-only status (see [8.2.5 Check In](#), for an explanation of the term “connected non-versionable subtree”).

This method acts directly on the workspace and the version storage. All changes are persisted immediately. There is no need to call `save`.

8.2.7 Restoring a Version

To restore a node `N` to the state recorded by its version with version name “`x.y`”, the client calls `N.restore("x.y", removeExisting)`⁸. Let us say that the version node representing the version named “`x.y`” is node `V`. In that case the following will occur:

- The child node and properties of `N` will be changed, removed or added to, depending on their corresponding copies in `V` and their own `OnParentVersion` attributes (see [8.2.11 OnParentVersion Attribute](#), for details). The second parameter of `Node.restore` is the `removeExisting` flag which governs what happens if nodes that are being introduced into the subtree of `N` as a result of the `restore` have the same UUID as existing node is in the workspace outside the subtree of `N` (see [8.2.14.1 Node Versioning Methods](#)).
- `N`'s `jcr:baseVersion` property will be changed to point to `V`.
- `N`'s `jcr:isCheckedOut` property is set to `false`.

Unlike most other operations that alter the state of a node, `restore` works regardless of whether the node in question is *checked-out* or *checked-in*.

8.2.8 Restoring a Group of Versions

In certain circumstances a “chicken and egg” problem may arise due to a cycle of `REFERENCE` properties when attempting to restore a node that has been removed.

For example, let us say that there is a node `/A` with child nodes `/A/B` and `/A/C`. Furthermore let there be `REFERENCE` properties `/A/B/X` `/A/C/Y` such that `X` refers to `/A/B` and `Y` refers to `/A/C`. Now assume that `A`, `B` and `C` are first checked-in (thus creating versions of all three nodes) and then `B` and `C` are deleted from the workspace.

⁸ There is also a variant, `Node.restoreByLabel`, which allows the version to be selected by (one of) its `jcr:versionLabel` value; otherwise, the semantics are the same.

In order to restore **B** or **C** the other must be restored first, since the reference properties **x** and **y** both require the existence of the node to which they refer. This is the “chicken and egg” problem.

To deal with such situations the method

```
Workspace.restore(Version[] versions,  
                  boolean removeExisting)
```

is provided. This method allows the client to simultaneously restore two or more versions. In this case the client must first find the **Version** objects (call them **va**, **vb** and **vc**) that correspond to the versions of **A**, **B** and **C** that are to be restored and calling

```
ws.restore(new Version[]{va, vb, vc}, removeExisting)
```

Notice that in order to restore **B** and **C**, the previous version of **A** must also be restored because its state contains the child links to **B** and **C**.

The **removeExisting** flag governs what happens in cases of UUID collision.

See [8.2.14.2 Workspace Versioning Methods](#), for more information.

8.2.9 Update

The method **Node.update(String srcWorkspace)** works in the same way as it does in repositories without versioning: it updates **this** node and all nodes in its subtree to reflect the state of their corresponding nodes in **srcWorkspace**. Unlike most other methods that change the state of a node, **update** will work if the node in question is read-only due to a checked-in node. See also, [7.1.7 Updating and Cloning Nodes across Workspaces](#).

8.2.10 Merge

The method **Node.merge** can be thought of as a *version-sensitive* **Node.update**. It works as follows:

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 **this** node does have a corresponding node, then the following happens:

- For each versionable node \mathbf{n} in the subtree rooted at **this** node, a *merge test* is performed comparing \mathbf{n} with its corresponding node in **srcWorkspace**, \mathbf{n}' .
- The merge test is done by comparing *the base version of \mathbf{n}* (call it \mathbf{v}) and *the base version of \mathbf{n}'* (call it \mathbf{v}').
- For any versionable node \mathbf{n} there are three possible outcomes of the merge test: *update*, *leave* or *failed*.
- If \mathbf{n} does not have a corresponding node then the merge result for \mathbf{n} is *leave*.
- If \mathbf{n} is currently checked-in then:
 - If \mathbf{v}' is a successor (to any degree) of \mathbf{v} , then the merge result for \mathbf{n} is *update*.
 - If \mathbf{v}' is a predecessor (to any degree) of \mathbf{v} or if \mathbf{v} and \mathbf{v}' are identical (i.e., are actually the same version), then the merge result for \mathbf{n} is *leave*.
 - If \mathbf{v} is neither a successor of, predecessor of, nor identical with \mathbf{v}' , then the merge result for \mathbf{n} is *failed*. This is the case where \mathbf{n} and \mathbf{n}' represent divergent branches of the version graph, thus determining the result of a merge is non-trivial.
- If \mathbf{n} is currently checked-out then:
 - If \mathbf{v}' is a predecessor (to any degree) of \mathbf{v} or if \mathbf{v} and \mathbf{v}' are identical (i.e., are actually the same version), then the merge result for \mathbf{n} is *leave*.
 - If any other relationship holds between \mathbf{v} and \mathbf{v}' , then the merge result for \mathbf{n} is *fail*.
- If **bestEffort** is **false** then the first time a merge result of *fail* occurs, the entire merge operation on this subtree is aborted, no changes are made to the subtree and a **MergeException** is thrown. If no merge result of *fail* occurs then:
 - Each versionable node \mathbf{n} with result *update* is updated to reflect the state of \mathbf{v}' .
 - Each versionable node \mathbf{n} with result *leave* is left unchanged, *unless \mathbf{n} is the child of a node with status update and \mathbf{n} does not have a corresponding node in **srcWorkspace**, in which case it is removed*.
- If **bestEffort** is **true** then:
 - Each versionable node \mathbf{n} with result *update* is updated to reflect the state of \mathbf{v}' .

- 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.*
- Each versionable node *N* with result *failed* is left unchanged except that the UUID 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 UUID 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 subtree that have failed to merge and thus require special attention (see [8.2.10.1 Merging Branches](#), immediately below). 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 UUID (for a referenceable nodes) or UUID plus relative path (for non-referenceable nodes).

Note that as a result of the final rule, above, a **merge** performed on a subtree 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**.

8.2.10.1 Merging Branches

As mentioned, 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 successor of the other in terms of version history. Instead, the *content* (that is, the subtree) of N' must be merged into the content 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.

In the above example, after the *content* of N' is merged into N , the client will want to 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 client 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 predecessor connection to each version for which `doneMerge` was called, thus joining those branches of the version graph.

See [8.2.14 Versioning API](#).

8.2.11 OnParentVersion Attribute

Every item (node or property) in the repository has a status indicator that governs what happens to that item when its parent node is versioned. This status is defined by the `onParentVersion` attribute in the `PropertyDef` or `NodeDef` that applies to the item in question.

For example, let N be a versionable node, meaning it has mixin node type `mix:versionable`. Also let N have a primary node type that allows it to have one property called P and one child node called C .

What happens to **p** and **c** when a new version of **n** is checked in depends on their respective **OnParentVersion** attribute as defined in the **PropertyDef** for **p** and the **NodeDef** for **c**.

The possible values for the **OnParentVersion** attribute are: **COPY**, **VERSION**, **INITIALIZE**, **COMPUTE**, **IGNORE** and **ABORT**.

The sections below describe, for each possible value of the **OnParentVersion** attribute, what happens to **c** and **p** when,

- **N.checkin()** is performed, creating the new version **v_N** and adding to the version history.
- **N.restore(v_N, b)** is performed, restoring the version **v_N** (the boolean parameter **b** governs what happens on UUID collision).

8.2.11.1 COPY

Child Node

On **checkin** of **N**, **c** and all its descendent items, down to the leaves of the subtree, will be copied to the version storage as a child subtree of **v_N**. The copy of **c** and its subtree will not have its own version history but will be part of the state preserved in **v_N**. **c** itself need not be versionable.

On **restore** of **v_N**, the copy of **c** and its subtree stored will be restored as well, replacing the current **c** and its subtree in the workspace.

Property

On **checkin** of **N**, **p** will be copied to the version storage as a child of **v_N**. This copy of **p** is part of the state preserved in **v_N**.

On **restore** of **v_N**, the copy of **p** stored as its child will be restored as well, replacing the current **p** in the workspace.

8.2.11.2 VERSION

Child Node

On **checkin** of **N**, the node **v_N** will get a subnode of type **nt:versionedChild** with the same name as **c**. The single property of this node, **jcr:childVersionHistory** is a **REFERENCE** to the *version history of c* (not to **c** or any actual version of **c**). This also requires that **c** itself be versionable (otherwise it would not have a version history). If **c** is not versionable then the behavior of **IGNORE** applies on **checkin** (see below).

NodeType Name

nt:versionedChild

Supertypes

nt:base

```

IsMixin
    false
HasOrderableChildNodes
    false
PrimaryItemName
    null
PropertyDef
    Name jcr:childVersionHistory
    RequiredType REFERENCE
    ValueConstraints []
    DefaultValues null
    AutoCreate true
    Mandatory true
    OnParentVersion ABORT
    Protected true
    Multiple false

```

On **restore** of v_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 **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. 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.

Property

In the case of properties, an **OnParentVersion** attribute of **VERSION** has the same effect as **COPY**.

8.2.11.3 INITIALIZE

Child Node

On **checkin** of N , a new node c will be created and placed in version storage as a child of v_N . This new c will be initialized just as it would be if created normally in a workspace. No state information of the current c in the workspace is preserved.

On **restore** of v_N , the c stored as its child will be ignored, and the current c in the workspace will be left unchanged.

Property

On **checkin** of N , a new P will be created and placed in version storage as a child of v_N . The new P will be initialized just as it would be if created normally in a workspace.

On **restore** of v_N , the p stored as its child will be ignored, and the current p in the workspace will be left unchanged.

8.2.11.4 COMPUTE

Child Node

On **checkin** of N , a new node c will be created and placed in version storage as a child of v_N . This new c will be initialized by some procedure defined for that type of child node.

On **restore** of v_N , the c stored as its child will be ignored, and the current c in the workspace will be left unchanged.

Property

On **checkin** of N , a new p will be created and placed in version storage as a child of v_N . The new p will be initialized by some procedure defined for that type of property.

On **restore** of v_N , the p stored as its child will be ignored, and the current p in the workspace will be left unchanged.

8.2.11.5 IGNORE

Child Node

On **checkin** of N , no state information about c will be stored in v_N .

On **restore** of v_N , the child node c of the current N will remain and not be removed.

Property

On **checkin** of N , no state information about p will be stored in v_N .

On **restore** of v_N , the property p of the current N will remain and not be removed.

8.2.11.6 ABORT

Child Node or Property

On **checkin** of N a `VersionException` will be thrown. Having a child node or property with an `OnParentVersion` attribute of **ABORT** prevents the parent node from being checked-in.

8.2.12 The `OnParentVersionAction` Class

The above six legal values from the `OnParentVersion` attribute are represented in the Java API by six integer constants defined by the class `OnParentVersionAction`.

```
javax.jcr.version.  
OnParentVersionAction
```

int	COPY
int	VERSION
int	INITIALIZE
int	COMPUTE
int	IGNORE
int	ABORT

8.2.13 Removal of Versions

Though a version history is meant, in theory, to provide a permanent record of a versionable node, in practice it sometimes becomes necessary to clean-up a version history by removing a version. To do this, JCR provides the `VersionHistory.removeVersion` method. See [8.2.14.3 VersionHistory Interface](#).

8.2.14 Versioning API

The versioning API consists the version-related methods in the `Node` interface as well as two interfaces that extend the `Node` interface, `VersionHistory` and `Version`. `VersionHistory` is the interface for an `nt:versionHistory` node and `Version` is the interface for an `nt:version` node.

8.2.14.1 Node Versioning Methods

The `Node` interface has the following version-related methods.

javax.jcr. Node	
Version	checkin() Creates a new version with a system generated name and returns that version. The <code>jcr:isCheckedOut</code> property of this node is set to <code>false</code> thus putting the node into the <i>checked-in</i> state. This means that this node and its <i>connected non-versionable subtree</i> become read-only. A node's connected non-versionable subtree is the set of non-versionable descendant nodes reachable from that node through child links without encountering any versionable nodes. In other words, the read-only status flows down from the checked-in node along every child link until either a versionable node is encountered or an item with no children is encountered. Read-only status means that an item cannot be altered by the client using standard API methods (<code>addNode</code> ,

	<p><code>setProperty</code>, etc.). The only exceptions to this rule are the <code>restore</code>⁹, <code>Node.merge</code> and <code>Node.update</code> operations; these do not respect read-only status due to check-in. Note that <code>remove</code> 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).</p> <p>See 8.2.5 Check In for more details.</p> <p>If this node is already checked-in, this method has no effect but returns the current base version of this node.</p> <p>If this node is not versionable, an <code>UnsupportedRepositoryOperationException</code> is thrown.</p> <p>A <code>VersionException</code> is thrown or if a child item of this node has an <code>OnParentVersion</code> status of <code>ABORT</code>. This includes the case where an unresolved merge failure exists on this node, as indicated by the presence of the <code>jcr:mergeFailed</code> property.</p> <p>If <code>checkin</code> succeeds, the change to the <code>jcr:checkedOut</code> property is automatically saved (there is no need to do an additional <code>save</code>).</p> <p>A <code>LockException</code> is thrown if a lock prevents the checkin.</p> <p>A <code>RepositoryException</code> is thrown if an error occurs.</p>
<code>void</code>	<p><code>checkout()</code></p> <p>Sets this versionable node to checked-out status by setting its <code>jcr:isCheckedOut</code> property to <code>true</code>, sets the <code>jcr:predecessors</code> property to be a reference to the current base version (the same value as held in <code>jcr:baseVersion</code>). This method puts the node into the <i>checked-out</i> state, making it and its connected non-versionable subtree no longer read-only (see <code>checkin</code>, above, for an explanation of the term “connected non-versionable subtree”).</p> <p>If successful, these changes are persisted immediately, there is no need to call <code>save</code>.</p> <p>See 8.2.6 Check Out for more details.</p> <p>If this node is not versionable, an</p>

⁹ `Workspace.restore`, `Node.restore` (all signatures) and `Node.restoreByLabel`.

	<p>UnsupportedRepositoryOperationException is thrown.</p> <p>A LockException is thrown if a lock prevents the checkout.</p> <p>A RepositoryException is thrown if an error occurs.</p>
<code>void</code>	<p>merge(String srcWorkspace, boolean bestEffort)</p> <p>This method can be thought of as a version-sensitive update (see 7.1.7 Updating and Cloning Nodes across Workspaces).</p> <p>It recursively tests each versionable node in the subtree of this node against its corresponding node in srcWorkspace with respect to the relation between their respective base versions and either updates the node in question or not, depending on the outcome of the test. For details see 8.2.10 Merge.</p> <p>A VersionException is thrown if bestEffort is false and a versionable node is encountered whose corresponding node's base version is on a divergent branch from this node's base version. See 8.2.10 Merge for details.</p> <p>If successful, the changes are persisted immediately, there is no need to call save.</p> <p>If the specified srcWorkspace does not exist, a NoSuchWorkspaceException is thrown.</p> <p>If the current session does not have sufficient permissions to perform the operation, then an AccessDeniedException is thrown.</p> <p>An InvalidItemStateException is thrown if this session (not necessarily this Node) has pending unsaved changes.</p> <p>A LockException is thrown if a lock prevents the merge.</p> <p>A RepositoryException is thrown if another error occurs.</p>
<code>void</code>	<p>doneMerge(Version version)</p> <p>Completes the merge process with respect to this node and the specified version.</p> <p>See 8.2.10 Merge for more details.</p> <p>If successful, the changes are persisted immediately,</p>

	<p>there is no need to call save.</p> <p>A VersionException is thrown if the version specified is not among those referenced in this node's jcr:mergeFailed property or if this node is currently checked-in.</p> <p>An UnsupportedRepositoryOperationException is thrown if this repository does not support versioning.</p> <p>If there are unsaved changes pending on this node, an InvalidItemStateException is thrown.</p> <p>A LockException is thrown if a lock prevents the operation.</p> <p>A RepositoryException is thrown if another error occurs.</p>
void	<p>cancelMerge(Version version)</p> <p>Cancels the merge process with respect to this node and the specified version.</p> <p>See 8.2.10 Merge for more details.</p> <p>If successful, the changes are persisted immediately, there is no need to call save.</p> <p>A VersionException is thrown if the version specified is not among those referenced in this node's jcr:mergeFailed property or if this node is currently checked-in.</p> <p>An UnsupportedRepositoryOperationException is thrown if this repository does not support versioning.</p> <p>If there are unsaved changes pending on this node, an InvalidItemStateException is thrown.</p> <p>A LockException is thrown if a lock prevents the operation.</p> <p>A RepositoryException is thrown if another error occurs.</p>
boolean	<p>isCheckedOut()</p> <p>Returns true if this node is either</p> <ul style="list-style-type: none"> • versionable and currently checked-out, • non-versionable and its nearest versionable ancestor is checked-out or • non-versionable and it has no versionable

	<p>ancestor.</p> <p>Returns false if this node is either</p> <ul style="list-style-type: none"> • versionable and currently checked-in or • non-versionable and its nearest versionable ancestor is checked-in. <p>A RepositoryException is thrown if an error occurs.</p>
void	<p>restore(String versionName, boolean removeExisting)</p> <p>Restores this node to the state defined by the version with the specified versionName.</p> <p>If this node is not versionable, an UnsupportedRepositoryOperationException is thrown.</p> <p>If successful, the change is persisted immediately and there is no need to call save.</p> <p>A VersionException is thrown if no version with the specified versionName exists in this node's version history.</p> <p>An InvalidItemStateException is thrown if this session (not necessarily this Node) has pending unsaved changes.</p> <p>This method will work regardless of whether this node is checked-in or not.</p> <p>A UUID collision occurs when a node exists <i>outside the subtree rooted at this node</i> with the same UUID as a node that would be introduced by the restore operation <i>into the subtree at this node</i>. 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 subtree) is removed. If removeExisting is false, then a ItemExistsException is thrown and no changes are made. Note that this applies not only to cases where the restored node itself conflicts with an existing node but also to cases where a conflict occurs with <i>any</i> node that would be introduced into the workspace by the restore operation. In particular conflicts involving subnodes of the restored node that have onParentVersion settings of COPY or VERSION (see 8.2.11 OnParentVersion Attribute) are also governed by the removeExisting flag.</p>

	<p>A LockException is thrown if a lock prevents the restore.</p> <p>A RepositoryException is thrown if another error occurs.</p>
void	<p>restore(Version version, boolean removeExisting)</p> <p>Restores this node to the state defined by the specified version.</p> <p>If this node is not versionable, an UnsupportedRepositoryOperationException is thrown.</p> <p>If successful, the change is persisted immediately and there is no need to call save.</p> <p>A VersionException is thrown if the specified version is not part of this node's version history.</p> <p>An InvalidItemStateException is thrown if this session (not necessarily this Node) has pending unsaved changes.</p> <p>This method will work regardless of whether this node is checked-in or not.</p> <p>A UUID collision occurs when a node exists <i>outside the subtree rooted at this node</i> with the same UUID as a node that would be introduced by the restore operation <i>into the subtree at this node</i>. 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 subtree) is removed. If removeExisting is false, then a ItemExistsException is thrown and no changes are made. Note that this applies not only to cases where the restored node itself conflicts with an existing node but also to cases where a conflict occurs with <i>any</i> node that would be introduced into the workspace by the restore operation. In particular conflicts involving subnodes of the restored node that have OnParentVersion settings of COPY or VERSION (see 8.2.11 OnParentVersion Attribute) are also governed by the removeExisting flag.</p> <p>A LockException is thrown if a lock prevents the restore.</p> <p>A RepositoryException is thrown if another error occurs.</p>

<pre>void</pre>	<pre>restore(Version version, String relPath, boolean removeExisting)</pre> <p>Restores the specified <code>version</code> to <code>relPath</code>, relative to this node.</p> <p>A node need not exist at <code>relPath</code>, though the parent of <code>relPath</code> must exist, otherwise a <code>PathNotFoundException</code> is thrown.</p> <p>If a node <i>does</i> exist at <code>relPath</code> then it must correspond to the version being restored (the version must be a version <i>of that node</i>), otherwise a <code>VersionException</code> is thrown.</p> <p>If no node exists at <code>relPath</code> then a <code>VersionException</code> is thrown if the parent node of <code>relPath</code> is versionable and checked-in or is non-versionable but its nearest versionable ancestor is checked-in.</p> <p>If there is a node at <code>relPath</code> then the checked-in status of that node itself and the checked-in status of its parent are irrelevant. The restore will work even if one or both are checked-in.</p> <p>A UUID collision occurs when a node exists <i>outside the subtree rooted at <code>relPath</code></i> with the same UUID as a node that would be introduced by the <code>restore</code> operation <i>into the subtree at <code>relPath</code></i> (note that in cases where there is no node at <code>relPath</code>, this amounts to saying that a UUID collision occurs if there exists a node <i>anywhere</i> in this workspace with the same UUID as a node that would be introduced by the <code>restore</code>). The result in such a case is governed by the <code>removeExisting</code> flag. If <code>removeExisting</code> is <code>true</code> then the incoming node takes precedence, and the existing node (and its subtree) is removed. If <code>removeExisting</code> is <code>false</code>, then a <code>ItemExistsException</code> is thrown and no changes are made. Note that this applies not only to cases where the restored node itself conflicts with an existing node but also to cases where a conflict occurs with <i>any</i> node that would be introduced into the workspace by the restore operation. In particular conflicts involving subnodes of the restored node that have <code>OnParentVersion</code> settings of <code>COPY</code> or <code>VERSION</code> (see 8.2.11 OnParentVersion Attribute) are also governed by the <code>removeExisting</code> flag.</p> <p>If the would-be parent of the location <code>relPath</code> is actually a property, or if a node type restriction would be violated, then a <code>ConstraintViolationException</code> is</p>
-----------------	---

	<p>thrown.</p> <p>If the restore succeeds, the changes made to this node are persisted immediately, there is no need to call save.</p> <p>An InvalidItemStateException is thrown if this session (not necessarily this Node) has pending unsaved changes.</p> <p>An UnsupportedRepositoryOperationException is thrown if versioning is not supported.</p> <p>A LockException is thrown if a lock prevents the restore.</p> <p>A RepositoryException is thrown if another error occurs.</p>
void	<p>restoreByLabel(String versionLabel, boolean removeExisting)</p> <p>Restores this node to the state defined by the version with the specified versionLabel.</p> <p>If this node is not versionable, an UnsupportedRepositoryOperationException is thrown.</p> <p>If successful, the change is persisted immediately and there is no need to call save.</p> <p>A VersionException is thrown if no version with the specified versionLabel exists in this node's version history.</p> <p>This method will work regardless of whether this node is checked-in or not.</p> <p>A UUID collision occurs when a node exists <i>outside the subtree rooted at this node</i> with the same UUID as a node that would be introduced by the restore operation <i>into the subtree at this node</i>. 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 subtree) is removed. If removeExisting is false, then a ItemExistsException is thrown and no changes are made. Note that this applies not only to cases where the restored node itself conflicts with an existing node but also to cases where a conflict occurs with <i>any</i> node that would be introduced into the workspace by the restore operation. In particular conflicts involving subnodes of the restored node that have onParentVersion settings of COPY or VERSION (see 8.2.11 OnParentVersion</p>

	<p><i>Attribute</i>) are also governed by the <code>removeExisting</code> flag.</p> <p>An <code>InvalidItemStateException</code> is thrown if this <code>Session</code> (not necessarily this <code>Node</code>) has pending unsaved changes.</p> <p>A <code>LockException</code> is thrown if a lock prevents the restore.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>
<code>VersionHistory</code>	<p><code>getVersionHistory()</code></p> <p>Returns the <code>VersionHistory</code> object of this node. This object provides access to the <code>nt:versionHistory</code> node holding this node's versions.</p> <p>If this node is not versionable, an <code>UnsupportedRepositoryOperationException</code> is thrown.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>
<code>Version</code>	<p><code>getBaseVersion()</code></p> <p>Returns the current base version of this versionable node.</p> <p>If this node is not versionable, an <code>UnsupportedRepositoryOperationException</code> is thrown.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>

8.2.14.2 Workspace Versioning Methods

The `Workspace` object provides the "group restore" method.

<code>javax.jcr. Workspace</code>	
<code>void</code>	<p><code>restore(Version[] versions, boolean removeExisting)</code></p> <p>Restores a set of versions at once. Used in cases where a "chicken and egg" problem of mutually referring <code>REFERENCE</code> properties would prevent the restore in any serial order.</p> <p>If the restore succeeds, the changes made are</p>

	<p>persisted immediately, there is no need to call save.</p> <p>This method will work regardless of whether the nodes corresponding to the specified versions are checked-in or not.</p> <p>The following restrictions apply to the set of versions specified:</p> <p>If <i>s</i> is the set of versions being restored simultaneously,</p> <ul style="list-style-type: none"> • For every version <i>v</i> in <i>s</i> that corresponds to a <i>missing</i> node in the workspace, there must also be a parent of <i>v</i> in <i>s</i>. • <i>s</i> must contain at least one version that corresponds to an <i>existing</i> node in the workspace. <p>If either any of these restrictions does not hold, the restore will fail because the system will be unable to determine the path locations to which one or more versions are to be restored. In this case a VersionException is thrown.</p> <p>The versionable nodes in this workspace that correspond to the versions being restored define a set of (one or more) subtrees. A UUID collision occurs when this workspace contains a node <i>outside these subtrees</i> that has the same UUID as one of the nodes that would be introduced by the restore operation into one of these subtrees. 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 subtree) is removed. If removeExisting is false then a ItemExistsException is thrown and no changes are made. Note that this applies not only to cases where the restored node itself conflicts with an existing node but also to cases where a conflict occurs with <i>any</i> node that would be introduced into the workspace by the restore operation. In particular conflicts involving subnodes of the restored node that have OnParentVersion settings of COPY or VERSION (see 8.2.11 OnParentVersion Attribute) are also governed by the removeExisting flag.</p> <p>An UnsupportedRepositoryOperationException is thrown if versioning is not supported.</p> <p>A LockException is thrown if a lock prevents the</p>
--	--

	<p>restore.</p> <p>An InvalidItemStateException is thrown if this session (not necessarily this Node) has pending unsaved changes.</p> <p>Throws a RepositoryException if another error occurs.</p>
--	---

8.2.14.3 VersionHistory Interface

A **VersionHistory** object provides an interface for an **nt:versionHistory** node. It provides convenient access to version history information.

javax.jcr.version. VersionHistory extends Node	
String	getVersionableUUID() Returns the UUID of the versionable node for which this is the version history. A RepositoryException is thrown if an error occurs.
Version	getRootVersion() Returns the root version of this version history. A RepositoryException is thrown if an error occurs.
VersionIterator	getAllVersions() Returns an iterator over all the versions within this version history. The order of the returned objects will not necessarily correspond to the order of versions in terms of the successor relation. To traverse the version graph one must traverse the jcr:successors REFERENCE properties starting with the root version (see above). A version history will always have at least one version, the root version. Therefore, this method will always return an iterator of at least size 1. A RepositoryException is thrown if an error occurs.
Version	getVersion(String versionName) Retrieves a particular version from this version history by version name. Throws a VersionException if the specified version is not in this version history. A RepositoryException is thrown if an error occurs.

Version	<p>getVersionByLabel(String label)</p> <p>Retrieves a particular version from this version history by version label.</p> <p>Throws a VersionException if the specified label is not in this version history.</p> <p>A RepositoryException is thrown if an error occurs.</p>
void	<p>addVersionLabel(String versionName, String label, boolean moveLabel)</p> <p>Adds the specified label to the specified version. This corresponds to adding a value to the jcr:versionLabels multi-value property of the nt:version node that represents the specified version.</p> <p>Note that this change is made immediately; there is no need to call save. In fact, since the version storage is read-only with respect to normal repository methods, save does not even function in this context.</p> <p>Within a particular version history, a given label may appear a maximum of once. 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 VersionException.</p> <p>A VersionException is also thrown if the named version is not in this VersionHistory.</p> <p>A RepositoryException is thrown if another error occurs.</p>
boolean	<p>hasVersionLabel(String label)</p> <p>Returns true if any version in the history has the given label.</p>
boolean	<p>hasVersionLabel(Version version, String label)</p> <p>Returns true if the given version has the given label</p>
String[]	<p>getVersionLabels()</p> <p>Returns all version labels of the history or an empty array if there are none.</p>
String[]	<p>getVersionLabels(Version version)</p>

	<p>Returns all version labels of the given version - empty array if none.</p> <p>Throws a VersionException if the specified version is not in this version history.</p> <p>Throws a RepositoryException if another error occurs.</p>
Void	<p>removeVersionLabel(String label)</p> <p>Removes the specified label from among the labels of this version history. This corresponds to removing a property from the jcr:versionLabels child node of the nt:versionHistory node that represents this version history.</p> <p>Note that this change is made immediately; there is no need to call save. In fact, since the version storage is read-only with respect to normal repository methods, save does not even function in this context.</p> <p>If a label is specified that does not exist in this version history, a VersionException is thrown.</p> <p>A RepositoryException is thrown if another error occurs.</p>
Void	<p>removeVersion(String versionName)</p> <p>Removes the named version from this version history and automatically repairs the version graph. If the version to be removed is v, v's predecessor set is p and v's successor set is s, then the version graph is repaired as follows:</p> <ul style="list-style-type: none"> • For each member of p, remove the reference to v from its successor list and add references to each member of s. • For each member of s, remove the reference to v from its predecessor list and add references to each member of p. <p>Note that this change is made immediately; there is no need to call save. In fact, since the version storage is read-only with respect to normal repository methods, save does not even function in this context.</p> <p>A ReferentialIntegrityException will be thrown if the specified version is currently the target of a REFERENCE property elsewhere in the repository (not necessarily in this workspace) and the current session</p>

	<p>has read access to that REFERENCE property.</p> <p>An AccessDeniedException will be thrown if the current session does not have permission to remove the specified version or if the specified version is currently the target of a REFERENCE property elsewhere in the repository (not necessarily in this workspace) and the current session <i>does not</i> have read access to that REFERENCE property.</p> <p>Throws an UnsupportedRepositoryOperationException if this operation is not supported by the implementation.</p> <p>Throws a VersionException if the named version is not in this VersionHistory.</p> <p>Throws a RepositoryException if another error occurs.</p>
--	---

8.2.14.4 The Version Interface

A **Version** object provides an interface for an **nt:version** node. It provides convenient access to version information.

javax.jcr.version. Version extends Node	
VersionHistory	getContainingHistory() Returns the versionHistory that contains this Version .
Calendar	getCreated() Returns the date this version was created. This corresponds to the value of the jcr:created property in the nt:version node that represents this version. A RepositoryException is thrown if an error occurs.
Version[]	getSuccessors() Returns the successor versions of this version. This corresponds to returning all the nt:version nodes referenced by the jcr:successors multi-value property in the nt:version node that represents this version. A RepositoryException is thrown if an error occurs.
Version[]	getPredecessors()

	<p>Returns the predecessor versions of this version. This corresponds to returning all the <code>nt:version</code> nodes whose <code>jcr:successors</code> property includes a reference to the <code>nt:version</code> node that represents this version.</p> <p>A <code>RepositoryException</code> is thrown if an error occurs.</p>
--	--

8.2.15 Serialization of Version Storage

Serialization of version information can be done in the same way as normal serialization by serializing the subtree below `/jcr:versionHistory`. 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 deserialized 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.

8.2.16 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.

8.3 Observation

A JCR compliant repository may support observation. This feature enables applications to register interest in events that describe changes to a workspace, and then monitor and respond to those events. The observation mechanism dispatches events when a *persistent change* is made to the workspace.

Note that (in those repositories that support transactions) in the case of changes made within a transaction, the corresponding events will only be dispatched upon *commit* of the transaction, whereas in the case of changes made outside a transaction the events will be dispatched upon *save* (or immediately in the case of direct-to-workspace methods). See [8.3.4 Event Production](#).

An object implementing the `Event` interface represents an event generated by a repository. It also contains the constants representing the five event types.

javax.jcr.observation. Event	
int	getType() Returns the type of this event. A constant defined by in this interface. One of NODE_ADDED , NODE_REMOVED , PROPERTY_ADDED , PROPERTY_REMOVED and PROPERTY_CHANGED .
String	getPath() Returns the absolute path of the item associated with this event. The interpretation given to the returned path depends upon the type of the event: <ul style="list-style-type: none"> • If the event type is NODE_ADDED then this method returns the absolute path of the node that was added. • If the event type is NODE_REMOVED then this method returns the absolute path of the node that was removed. • If the event type is PROPERTY_ADDED then this method returns the absolute path of the property that was added. • If the event type is PROPERTY_REMOVED then this method returns the path of the property that was removed. • If the event type is PROPERTY_CHANGED then this method returns the absolute path of the changed property. A RepositoryException is thrown if an error occurs.
String	getUserId() Returns the user ID connected with this event. This is the string returned by getUserId of the session that caused the event.
int	NODE_ADDED An event of this type is generated when a node is added.
int	NODE_REMOVED An event of this type is generated when a node is removed.

int	PROPERTY_ADDED An event of this type is generated when a property is added.
int	PROPERTY_REMOVED An event of this type is generated when a property is removed.
int	PROPERTY_CHANGED An event of this type is generated when the value of a property is changed.

8.3.1 Event Listeners

An application registers its interest in events by registering an event listener with the workspace. Listeners are *per workspace*, not repository-wide; they only receive events for the workspace in which they are registered.

When an persistent change occurs, the repository calls the **onEvent** method of each registered listener that is entitled (based on the filters set for that listener) to receive notification, and passes it an **EventIterator** object. The **EventIterator** contains the bundle of events (again, filtered for that particular listener) that describe the persistent changes made to the workspace.

javax.jcr.observation. EventListener	
void	onEvent(EventIterator event) This method is called when a bundle of events is dispatched. See 8.3.4 Event Production .

8.3.2 Listener Registration

Registration of event listeners is done through the **ObservationManager** object acquired from the **Workspace**.

javax.jcr. Workspace	
Observation Manager	getObservationManager() Returns the ObservationManager object. If the implementation does not support observation, an UnsupportedRepositoryOperationException is thrown. A RepositoryException is thrown if an error occurs.

8.3.3 Observation Manager

The `ObservationManager` interface supports listener registration and deregistration.

javax.jcr.observation. ObservationManager	
void	<pre>addEventListener(EventListener listener, int eventTypes, String absPath, boolean isDeep, String[] uuid, String[] nodeTypeName, boolean noLocal)</pre> <p>Adds an event listener that listens for the specified <code>eventTypes</code> (a combination of one or more event types encoded as a bit mask value).</p> <p>The set of events can be filtered by specifying restrictions based on characteristics of the <i>associated parent node</i> 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 <code>Event.getPath</code>. The following restrictions are available:</p> <ul style="list-style-type: none">• absPath, isDeep: Only events whose associated parent node is at <code>absPath</code> (or within its subtree, if <code>isDeep</code> is <code>true</code>) will be received. It is permissible to register a listener for a path where no node currently exists.• uuid: Only events whose associated parent node has one of the UUIDs in this list will be received. If this parameter is <code>null</code> then no UUID-related restriction is placed on events received. Note that specifying an empty array instead of <code>null</code> would result in no nodes being listened to.• nodeTypeName: Only events whose associated parent node has one of the node types (or a subtype of one of the node types) in this list will be received. If this parameter is <code>null</code> then no node type-related restriction is placed on events received. Note that specifying an empty array instead of <code>null</code> would result in no node types being listened to.

	<p>The restrictions are "ANDed" together. In other words, for a particular node to be listened to, it must meet <i>all</i> the restrictions.</p> <p>Additionally, if <code>noLocal</code> is <code>true</code>, then events generated by the session through which the listener was registered are ignored. Otherwise, they are not ignored.</p> <p>The filters of an already-registered <code>EventListener</code> can be changed at runtime by re-registering the same <code>EventListener</code> object (i.e. the same actual Java object) with a new set of filter arguments. The implementation must ensure that no events are lost during the changeover.</p> <p>A <code>RepositoryException</code> is thrown if an error occurs.</p>
<code>void</code>	<p><code>removeEventListener(EventListener listener)</code></p> <p>Deregisters an event listener.</p> <p>A <code>RepositoryException</code> is thrown if an error occurs.</p>
<code>EventListenerIterator</code>	<p><code>getRegisteredEventListeners()</code></p> <p>Returns all event listeners that have been registered through this session. If no listeners have been registered, an empty iterator is returned.</p> <p>A <code>RepositoryException</code> is thrown if an error occurs.</p>

8.3.4 Event Production

Events are dispatched upon each *persistent change* to the workspace. Changes that affect only the transient session level are not tracked by the observation mechanism. This means that events will be dispatched as follows:

- If a set of operations is *within a transaction* (see [8.1 Transactions](#)) then the events reflecting the resulting changes will only be dispatched after the changes are persisted by a successful *commit*.
- If a set of operations is *not within a transaction* then:
 - If an operation is *immediately persistent* (like `Workspace.copy`, for example), the events reflecting

the resulting changes will be dispatched upon the successful completion of the operation.

- If a set of operations is *not immediately persistent* (like most `Node` and `Session` methods, for example) then the events reflecting the resulting changes will be dispatched upon the successful `save` of those changes.

8.3.5 Event Filtering

An event listener will only receive events for which its session (the session through which it was registered) has sufficient access control permissions and which meet the filtering restrictions specified upon registration. See [8.3.3 Observation Manager](#).

8.3.6 Event Bundles

On each persistent change, those listeners that are entitled to receive one or more events will have their `onEvent` method called and be passed an `EventIterator`.

The `EventIterator` will contain the event bundle reflecting the persistent changes made but excluding those to which that particular listener is not entitled, according to the listeners access permissions and filters.

8.3.7 Interpretation of Events

The set of available event types is small, consisting of only five types: `NODE_ADDED`, `NODE_REMOVED`, `PROPERTY_ADDED`, `PROPERTY_REMOVED` and `PROPERTY_CHANGED`. The intent of the event notification system is to describe, for every persistent operation, the resulting state change in the workspace, and not necessarily the operational steps performed by the client that lead to that change. The set of five event types and the bundling of those events is sufficient to describe any state change and make that change correctly interpretable by the consumer of the events. The following describes the events generated as a result of a number of common operations. Note that the following describes the events that would be generated when the change caused by the operation in question is persisted.

8.3.7.1 Creating a new Node

When a new node is created, distinct events are generated for the addition of the actual new node itself (a `NODE_ADDED` event) as well for each of the automatically created child nodes or properties (either `NODE_ADDED` or `PROPERTY_ADDED`, as the case may be). This includes properties required by the system such as `jcr:primaryType`. Events for these system-mandated additions

may of course be filtered out by specifying filtering by, for example, node type.

8.3.7.2 Creating a Property

When a new property is created, a `PROPERTY_ADDED` event is generated. No `PROPERTY_CHANGED` event is generated.

8.3.7.3 Changing a Property

When an existing property's value is changed, a `PROPERTY_CHANGED` event is generated.

8.3.7.4 Removing a Child Node

When a node is removed, a `NODE_REMOVED` event is generated.

8.3.7.5 Removing a Property

When a property is removed, a `PROPERTY_REMOVED` event is generated.

8.3.7.6 Copying a Subtree

When a subtree is copied, the implementation *may either* generate appropriate events for each resulting node and property addition, *or* it may simply generate a single `NODE_ADDED` reflecting the addition of the root of the subtree at the destination location.

8.3.7.7 Moving a Subtree

When a subtree is moved a `NODE_REMOVED` is generated for the removal from the source location and a `NODE_ADDED` is generated for the addition at the destination location.

8.3.7.8 Adding a Mixin

If this is the *first* mixin to be added to this node, a `PROPERTY_ADDED` event will be generated reflecting the addition of the multi-value `jcr:mixinTypes` property. If this is *not* the first mixin to be added then a `PROPERTY_CHANGED` event will be generated reflecting the addition of the new value to `jcr:mixinTypes`.

8.3.7.9 Removing a Mixin

Assuming an implementation allows removal of mixin types then a `PROPERTY_CHANGED` event is produced reflecting the removal of the relevant value from `jcr:mixinTypes`.

8.3.7.10 Checking in a Node

In versioning repositories, the version storage appears as a protected subtree of each workspace. By placing listeners on this subtree a client can be alerted to versioning events. The events

generated will reflect the changes made to the version storage area (for example, the addition of a new `nt:version` node below an `nt:versionHistory` node) as a result of the check-in operation. From this information the source of the check-in can be determined (for example, the `nt:frozenNode`'s `jcr:frozenUuid` property holds the UUID of the workspace node that was checked-in).

8.3.7.11 Restoring, Updating or Merging a Node

Restoring updating and merging of nodes will generate events that reflect the changes made to those nodes as a result of the operation.

8.3.7.12 Locking and Unlocking a Node

By listening for changes on `mix:lockable` nodes, locking events can be detected. Locking a node will generate `PROPERTY_ADDED` events reflecting the addition of the `jcr:lockOwner` and `jcr:lockIsDeep` properties. Unlocking a node will generate `PROPERTY_REMOVED` events reflecting the removal of these properties.

8.3.8 Deserializing Content

Whether events are generated for each node and property addition that occurs when content is deserialized into a workspace is left up to the implementation.

8.3.9 External Mechanisms

Whether events are generated for changes made to a workspace through mechanisms external to this specification is left up to the implementation.

8.3.10 Location of Listeners

The classes implementing the listener interfaces will reside on the same JVM as the repository itself. In implementations where both the application using the API and the repository itself are operating on the same JVM, this poses no particular problems.

In client-server implementations that use RMI to connect the application to a remote repository, the application must ensure that any listeners registered to the repository are serializable, thus allowing them to be passed to the JVM running the repository instance.

8.3.11 Persistence of Event Listeners

Though not explicitly defined in this specification, nothing prevents a repository from registering "Persistent Event Listeners" through its configuration.

Since the “persistence” of an event listener is only limited through the registering `session`’s lifespan, the repository can use the same mechanisms as for a non-durable registration but use a session that has the lifespan of the repository instance (the “system” session, for example).

Persistent event listeners may be used to provide more system level functionality such as specialized access control and syndication/replication mechanisms.

8.3.12 Vetoable Event Listeners

This specification defines only *asynchronous* event delivery. It is possible for a repository to also implement *synchronous* events in order to support the vetoing of changes before they happen. However, this functionality is outside the scope of this specification.

8.3.13 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.

8.4 Access Control

A JCR compliant repository may support access control. This specification defines a mechanism for the *discovery* of access control information (that is, whether a particular `session` has permission to perform a particular action). However, the specification does not attempt to define mechanisms for the *setting* of access control policies.

As mentioned above (see [6.1 Accessing the Repository](#)), the `session` object returned by `Repository.login` reflects a particular set of access permissions. These permissions may be determined by the `Credentials` passed on `login` or by some external authentication and authorization mechanism within which the repository implementation is embedded.

8.4.1 JAAS

If an external mechanism is employed, a likely candidate is the Java Authentication and Authorization Service (JAAS) (see <http://java.sun.com/products/jaas/>).

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

The discovery mechanism for finding what permissions apply is also JAAS-compatible since it uses the JAAS-like concept of *actions*.

8.4.2 Checking Permissions

Permission checking is done through the method `Session.checkPermission`:

javax.jcr. Session	
void	<p>checkPermission(String absPath, String actions)</p> <p>Determines whether this <code>Session</code> has permission to perform the specified <code>actions</code> at the specified <code>absPath</code>. This method quietly returns if the access request is permitted, or throws a suitable <code>java.security.AccessControlException</code> otherwise.</p> <p>The actions parameter is a comma separated list of action strings. The following action strings are defined:</p> <p>add_node If <code>checkPermission(path, "add_node")</code> returns quietly, then this <code>Session</code> has permission to add a node at <code>path</code>, otherwise permission is denied.</p> <p>set_property If <code>checkPermission(path, "set_property")</code> returns quietly, then this <code>Session</code> has permission to set (add or change) a property at <code>path</code>, otherwise permission is denied.</p> <p>remove If <code>checkPermission(path, "remove")</code> returns quietly, then this <code>Session</code> has permission to remove an item at <code>path</code>, otherwise permission is denied.</p> <p>read If <code>checkPermission(path, "read")</code> returns quietly, then this <code>Session</code> has permission to retrieve (and read the value of, in the case of a property) an item at <code>path</code>, otherwise permission is denied.</p> <p>When more than one action is specified in the <code>actions</code> parameter, this method will only return quietly if this <code>Session</code> has permission to perform <i>all</i> of the listed actions at the specified <code>path</code>.</p> <p>The information returned through this method will only reflect access control policies and not other restrictions that may exist. For example, even though <code>checkPermission</code> may indicate that a particular <code>Session</code> may add a property at <code>/A/B/C</code>,</p>

	the node type of the node at <code>/A/B</code> may prevent the addition of a property called <code>c</code> .
--	---

8.5 Locking

In those repositories that support it, locking allows a user to temporarily lock nodes in order to prevent other users from changing them.

This function is typically used to serialize access to a node in order to forestall the "lost update problem". Though JCR already prevents this kind of inadvertent overwriting of repository content through the `InvalidItemStateException`, the use of locking can prevent the exception from occurring in the first place.

8.5.1 Discovery of Lock Capabilities

In order to support locking, a repository must provide the mixin type `mix:lockable`. Therefore, to determine whether a particular repository implementation supports locking, the user must call `NodeTypeManager.getNodeType("mix:lockable")`.

8.5.2 Lockable

A lock is placed on a node by calling `Node.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` may hold locks. The definition of `mix:lockable` is:

```

NodeTypeName
  mix:lockable
Supertypes
  []
IsMixin
  true
HasOrderableChildNodes
  false
PrimaryItemName
  null
PropertyDef
  Name jcr:lockOwner
  RequiredType STRING
  ValueConstraints []
  DefaultValues null
  AutoCreate false
  Mandatory false
  OnParentVersion IGNORE
  Protected true
  Multiple false
PropertyDef
  Name jcr:lockIsDeep
  RequiredType BOOLEAN
  ValueConstraints []
  DefaultValues null
  AutoCreate false

```

Mandatory **false**
OnParentVersion **IGNORE**
Protected **true**
Multiple **false**

8.5.3 Shallow and Deep Locks

A lock can be specified as either *shallow* or *deep*. A shallow lock applies only to its holding node. 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 subtree. 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 subtree, 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:isDeep** 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.

[it is possible to]

8.5.4 Lock Owner

The user who places a lock on a node is called the lock owner. The lock owner is identified by the user ID bound to the **session** through which the node in question was accessed (i.e., the string returned by **session.getUserId**). The user ID is recorded in the property **jcr:lockOwner** of the lock holding node. The lock owner's ID is provided for informational purposes only, it is not used in testing whether a particular user has any permissions with respect to the lock (this is governed purely by whether the **session** holds the applicable lock token, see below)

8.5.5 Placing and Removing a Lock

When **Node.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 user ID of the user who set the lock (this is the value returned by **session.getUserId**).
- **jcr:lockIsDeep** is set to reflect whether the lock is deep or not.

When **Node.unlock** is performed on a locked **mix:lockable** node, by a user with the correct lock token (see below) these two

properties are removed. The identity of the holder of the lock token does not matter (it does not have to be the lock owner). Anyone with the correct token can remove the lock.

Additionally, the system may give permission to some users to unlock locks for which they do not have the lock token. Typically such “lock-superuser” capability is intended to facilitate administrative clean-up of orphaned open-scoped locks.

An attempt to call `lock` or `unlock` on a node that is not `mix:lockable` will throw a `UnsupportedRepositoryOperationException`,

An attempt to lock an already locked node or unlock an already unlocked node will throw a `LockException`.

8.5.6 Lock Token

The method `Node.lock` returns a `Lock` object, which in turn contains a lock token. A lock token is a string that uniquely identifies a particular lock and acts as a “key” allowing a user to alter a locked node.

In order to use the lock token as a key, it must be added to the `session` of the user, thus empowering that session to alter the nodes to which the lock applies. When a lock token is attached to a `session`, the user of that session becomes a *token holder* of that lock token.

The method `Node.lock` automatically adds the lock token for a newly placed lock to the current `session`. If a user requires more control over which lock tokens are attached to the session, the `session` interface provides the methods `addLockToken`, `removeLockToken` and `getLockTokens`.

At any given time within a given repository, at most one `session` may hold a particular lock token.

Note that, as mentioned above, any user with the correct lock token assumes the power to remove a lock and alter nodes under that lock. It does not have to be the lock owner.

8.5.7 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 either explicitly unlocked or an implementation-specific limitation intervenes (like a timeout, see below).

In both cases, 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, however, the user need not explicitly do anything since the token is automatically attached to the session and expires with it in any case.

With open-scoped locks the token is also 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. By assumption, an open-scoped lock is being used to avoid co-expiration with the initial session. Otherwise, there would be no point in using an open-scoped lock, since session scoping would suffice. It is for handling these cases of attaching an existing lock token from a previous session to a new session that the methods `addLockToken`, `removeLockToken` and `getLockTokens` are provided.

8.5.8 Effect of a Lock

If a lock applies to particular node (i.e., the node either holds the lock or is a descendant of a node holding a deep lock), that node cannot be changed by anyone except the user who is the token holder for that lock. The user need not be the lock owner.

Note that since at most one session per repository may hold the same lock token, serial access to the locked item is ensured.

More precisely, a lock applying to a node prevents all non token holders from doing any of the following:

- Adding or removing its properties.
- Changing the values of its properties.
- Adding or removing its child nodes.
- Adding or removing its mixin node types.

Removing a node is considered an alteration of *its parent*. This means that a locked node may be removed by any user with sufficient access permissions as long as its parent node is not locked. Similarly, a locked node and its subtree may be moved, if both the source parent and the destination parent-to-be are not locked. Locked nodes can always be read and copied by any user with sufficient access permissions.

When an action is prevented due to a lock, a `LockException` is thrown immediately (not on `save`).

There is at most one lock on any node at one time.

8.5.9 Timing Out

An implementation may unlock any lock at any time due to implementation-specific criteria, such as time limits on locks.

8.5.10 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
```

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 published (that is, made visible to other `sessions`) 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.

8.5.11 Locking Methods

The methods for locking, unlocking and querying the locking status of a node are found in the `Node` interface itself:

javax.jcr. Node	
Lock	<p>lock(boolean isDeep, boolean isSessionScoped)</p> <p>Places a lock on this node. If successful, this node is said to <i>hold</i> the lock.</p> <p>If isDeep is true then the lock <i>applies</i> to this node and all its descendant nodes; if false, the lock applies only to this, the holding node.</p> <p>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 explicitly unlocked or automatically unlocked due to a implementation-specific limitation, such as a timeout.</p> <p>Returns a Lock object reflecting the state of the new lock and including a lock token.</p> <p>The lock token is also automatically added to the set of lock tokens held by the current session.</p> <p>If successful, then the property jcr:lockOwner is created and set to the value of session.getUserId for the current session and the property jcr:lockIsDeep is set to the value passed in as isDeep. These changes are persisted automatically; there is no need to call save.</p> <p>Note that it is possible to lock a node even if it is checked-in (the lock-related properties will be changed despite the checked-in status). See 8.2 Versioning for an explanation of "checked-in" status.</p> <p>If this node is not of mixin node type mix:lockable then an UnsupportedRepositoryOperationException is thrown.</p> <p>If this node is already locked (either because it holds a lock or a lock above it applies to it), a LockException is thrown.</p> <p>If isDeep is true and a descendant node of this node already holds a lock, then a LockException is thrown.</p> <p>If the current session does not have sufficient permissions to place the lock, an AccessDeniedException is thrown.</p>

	<p>An InvalidItemStateException is thrown if this node has pending unsaved changes.</p> <p>A RepositoryException is thrown if another error occurs.</p>
Lock	<p>getLock()</p> <p>Returns the Lock object that applies to this node. This may be either a lock on this node itself or a deep lock on a node above this node.</p> <p>If this session (the one through which this Node was acquired) holds the lock token for this lock, then the returned Lock object contains that lock token (accessible through Lock.getLockToken). If this session does not hold the applicable lock token, then the returned Lock object will not contain the lock token (its Lock.getLockToken method will return null). See Lock, below.</p> <p>If this node is not of mixin node type mix:lockable then an UnsupportedRepositoryOperationException is thrown.</p> <p>If no lock applies to this node, a LockException is thrown.</p> <p>If the current session does not have sufficient permissions to get the lock, an AccessDeniedException is thrown.</p> <p>A RepositoryException is thrown if another error occurs.</p>
void	<p>unlock()</p> <p>Removes the lock on this node. Also removes the properties jcr:lockOwner and jcr:lockIsDeep from this node. These changes are persisted automatically; there is no need to call save.</p> <p>If this node is mix:lockable but either does not currently hold a lock or holds a lock for which this session does not have the correct lock token, then a LockException is thrown. Note however that the system may give permission to some users to unlock locks for which they do not have the lock token. Typically such "lock-superuser" capability is intended to facilitate administrative clean-up of orphaned open-scoped locks.</p> <p>If this node is not mix:lockable then an UnsupportedRepositoryOperationException is thrown.</p> <p>Note that either of these exceptions may be thrown when an attempt is made to unlock a node that is locked due to a deep lock above it. In such cases the unlock method fails</p>

	<p>because the lock is not <i>held</i> by this node.</p> <p>Note also that it is possible to unlock a node even if it is checked-in (the lock-related properties will be changed despite the checked-in status). See 8.2 Versioning for an explanation of “checked-in” status.</p> <p>If the current session does not have sufficient permissions to remove the lock, an <code>AccessDeniedException</code> is thrown.</p> <p>An <code>InvalidItemStateException</code> is thrown if this node has pending unsaved changes.</p> <p>A <code>RepositoryException</code> is thrown if another error occurs.</p>
boolean	<p>holdsLock()</p> <p>Returns <code>true</code> if this node holds a lock; otherwise returns <code>false</code>. To <i>hold</i> a lock means that this node has actually had a lock placed on it specifically, as opposed to just having a lock <i>apply</i> to it due to a deep lock held by an node above.</p> <p>A <code>RepositoryException</code> is thrown if an error occurs.</p>
boolean	<p>isLocked()</p> <p>Returns <code>true</code> if this node is locked either as a result of a lock held by this node or by a deep lock on a node above this node; otherwise returns <code>false</code>.</p>

8.5.12 The Lock Object

The `Lock` object represents a lock on a particular node:

javax.jcr.lock. Lock	
String	<p>getLockOwner()</p> <p>Returns the user ID of the user who owns this lock. This is the value of the <code>jcr:lockOwner</code> property of the lock-holding node. It is also the value returned by <code>Session.getUserId</code> at the time that the lock was placed. The lock owner's identity is only provided for informational purposes. It does not govern who can perform an <code>unlock</code> or make changes to the locked nodes; that depends entirely upon who the token holder is.</p>
boolean	<p>isDeep()</p> <p>Returns <code>true</code> if this is a deep lock; <code>false</code> otherwise.</p>

Node	<p>getNode()</p> <p>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 subtree of the lock holder, H, then this call will return H.</p>
String	<p>getLockToken()</p> <p>May return the lock token for this lock. If this session holds the lock token for this lock, then this method will return that lock token. If this session does not hold the applicable lock token then this method will return null.</p>
boolean	<p>isLive()</p> <p>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 JCR-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.</p>
void	<p>refresh()</p> <p>Refreshes (brings back to life) a previously unlocked Lock object (one for which isLive returns false). If this lock is still live (isLive returns true) or if this session does not hold the correct lock token for this lock, then a LockException is thrown.</p> <p>The implementation may either revive an existing lock or issue a new lock (removing this one from the session and substituting the new one). This is an implementation-specific issue.</p> <p>A RepositoryException is thrown if another error occurs.</p>

8.5.13 Session Methods Related to the Lock Token

The **session** object provides the following methods for managing lock tokens:

```
javax.jcr.  
Session
```

void	addLockToken(String lt) Adds the specified lock token to this session. Holding a lock token allows the <code>session</code> object of the lock owner to alter nodes that are locked by the lock specified by that particular lock token.
String[]	getLockTokens() Returns an array containing all lock tokens currently held by this session.
void	removeLockToken(String lt) Removes the specified lock token from this session.

8.6 Searching Repository Content with SQL

A repository (either level 1 or level 2) may support search using the SQL query syntax.

8.6.1 The SQL Language

If supported, the SQL language syntax is invoked by specifying the constant `Query.SQL` in `QueryManager.createQuery` (see [6.6.7 Query API](#)):

<code>javax.jcr.query. Query</code>	
String	SQL A string constant representing the SQL query language applied to the <i>database view</i> of the workspace.

8.6.2 Database View

SQL queries in JCR work on *database view* of the workspace being searched (similar in principle to the two XML views described in [6.4 XML Mappings](#)). The database view of a workspace is defined below.

8.6.2.1 Node Types as Tables

- Each node type (primary or mixin) corresponds to a table.
- Each column in the table corresponds to a property defined in or inherited by that node type (including properties inherited from either primary or mixin node types). Modeling residual properties as columns is optional (see below).
- Each row corresponds to a node in the workspace.

- Note that because of the hierarchical structure of node type definitions, nodes will appear in more than just one table. For example, querying the `nt:base` table will show all nodes in the workspace, but that table will be limited to the columns corresponding to the properties defined by `nt:base: jcr:primaryType` and `jcr:mixinTypes`.
- Child node relationships are not recorded in the database view.

8.6.2.2 Pseudo-property *jcr:path*

- A special column, `jcr:path`, that does not correspond to any actual property, may be present in the node type tables. The `jcr:path` column, if present, holds the normalized absolute path for the node represented by each row.
- Discovery of support for `jcr:path` is done by checking for the existence of the repository descriptor key `QUERY_JCRPATH`. See [6.1.1.1 Repository Descriptors](#).
- If supported, then `jcr:path` appears in all node type tables and may be used in the `SELECT` and `WHERE` clauses of a SQL query.
- When a `jcr:path` value is returned in a result table will be in *compact form*, where index notation is only used if necessary, i.e., where lack of an index indicates an implicit index of [1]. However, when a test is performed within a `WHERE` clause against a `jcr:path`, the query mechanism will intelligently match both compact and explicit forms of the same path. For example, the following `WHERE` clauses define the same constraint:

```
WHERE jcr:path='/foo/bar'
```

```
WHERE jcr:path='/foo[1]/bar[1]'
```

Consequence, to select all same name siblings one uses the following syntax:

```
WHERE jcr:path LIKE '/foo/bar[%]'
```

- Predicates in the `WHERE` clause that test `jcr:path` are only required to support the operators `=` and `LIKE`. In the case of `LIKE` predicates, support is only required for tests using the `%` wildcard character either as the last path segment or with in index brackets. This set of minimum requirements would, for example, allow the following path queries:
 - Exact path:
`jcr:path='/books/mybooks/EffectiveJava'`

- Descendant:
`jcr:path LIKE '/books/mybooks/%'`
- Descendant or self:
`jcr:path LIKE '/books/mybooks/%' OR
jcr:path='/books/mybooks'`
- Index test:
`jcr:path LIKE '/books[%]/mybooks[%]'`

8.6.2.3 Path Literals

- Any path literals mentioned within a **WHERE** clause are automatically normalized ("`..`" and "`.`" are resolved, trailing `/` are removed) before further processing.

8.6.2.4 Pseudo-property *jcr:score*

- Another special column, `jcr:score`, that does not correspond to any actual property, may be present in the result table.
- Discovery of support for the `jcr:score` pseudo-property is done by checking for the existence of the repository descriptor `QUERY_JCRSCORE` (see [6.1.1.1 Repository Descriptors](#)).
- If supported, the `jcr:score` comes into play only when a full-text search is specified within a SQL query. This is done by using the `CONTAINS()` function. The `CONTAINS` function takes one argument, a Google-style search string (see [8.6.5.5 CONTAINS](#)).
- If `jcr:score` is supported and **SELECTED**, then when a full-text search is specified within the SQL query by a `CONTAINS` function in the **WHERE** clause, the result table will include the `jcr:score` column.
- The `jcr:score` value indicates ranking information relevant to full text search. The `jcr:score` column is only meaningful in queries that do a `CONTAINS` on a property that is indexed for full text search, otherwise the `jcr:score` value is not guaranteed to have any meaning. This specification does not mandate which properties in which node types must be indexed; this is left up to the implementation.
- Unlike `jcr:path`, `jcr:score` only appears in the result table not in any of the node type tables that define the database view of the workspace. Consequently `jcr:score` cannot appear in a **WHERE** clause.

- If multiple **CONTAINS** functions appear in a query the method of determining the combined **jcr:score** is left up to the implementation.
- The range of the **jcr:score** value is not mandated. This is implementation specific.
- The greater the **jcr:score** value, the more relevant the specific result. Note that this means that when using **ORDER BY jcr:score**, the keyword **DESCENDING** in order to rank the results by relevance, since the default of **ORDER BY** is **ASCENDING**.

8.6.2.5 Namespace delimiting colons

- Since table and column names are, respectively, node type and property names, they will in many cases contain a namespace-delimiting colon character. In JCR SQL the colon is therefore considered a valid character within table and column names and does not indicate a parameter, as it would in standard SQL.

8.6.2.6 Joins

- Support for joins is required only on the **jcr:path** column and only for joining a primary node type table with a mixin node type tables. For example:

```
SELECT nt:file.jcr:path, jcr:lockOwner
FROM nt:file, mix:lockable
WHERE jcr:lockOwner = 'John'
      AND mynt:file.jcr:path=mix:lockable.jcr:path
```

Additional join support, on other columns and with other tables, is optional.

8.6.2.7 Multi-value Properties

- Multi-value properties cannot be specified in the **SELECT** clause and are excluded when the **SELECT** clause specifies a **"*"**.
- In the **WHERE** clause the syntax of **value IN property** is used to specify that the stated **value** must be among those in **property**. The syntax **property = v**, where **property** is multi-valued will only evaluate to **true** if **property** has exactly one value and that value is equal to **v**.

8.6.2.8 Null Values

- In the **WHERE** clause the term **property IS NULL** is true if **property** is missing from a particular node instance. This covers the cases where:

- The property is declared explicitly in the node type but happens not to be present on this particular node.
 - The node type has a residual property declaration that *would* allow the specified property to be present on the node, but it happens not to be present on this particular node.
 - The property is not declared at all (neither explicitly nor implicitly) in the node type and hence is not present on this particular node.
- Similarly, **property IS NOT NULL** tests for the existence of the property in the node instance (and of course if does exist it will either be explicitly defined or present due to an implicit declaration as a residual property).

8.6.2.9 Undefined Property Types

- When a node type τ defines a property p with an **UNDEFINED** property type this means that two node instances of type τ can each have a property called p but with differing property types. In such cases the database view of τ is a single column of type **VARCHAR** (in other words, the different types are converted to strings).

8.6.2.10 Data Type Mapping

The following type mapping governs the usage of property variables and literals within a JCR SQL statement.

JCR property type	SQL type
STRING	VARCHAR
BINARY	BINARY
DOUBLE	DOUBLE
LONG	BIGINT
BOOLEAN	BIT
DATE	DATE
NAME	VARCHAR (namespace aware)
PATH	VARCHAR (namespace aware)
REFERENCE	CHAR (36)

8.6.2.11 Optional Features

The following are some common optional features that some implementations may choose to support (though this list should not be taken to exclude additional extensions as well).

- It is optional to support the specifying of residual properties (by name, not wildcard) in the **SELECT**, **WHERE** and **ORDER BY** clauses.
- It is optional to support properties in the **SELECT**, **WHERE** and **ORDER BY** clauses that are not explicitly defined in the node types listed in the **SELECT** clause but which are defined in subtypes of those node types.
- It is optional to support the specifying of properties in the **SELECT**, **WHERE** and **ORDER BY** clauses that are not explicitly defined in the node types listed in the **SELECT** clause but which are defined in mixin node types that may be assigned to node instances of the types that are mentioned in the **SELECT** clause.

8.6.3 SQL EBNF

Terminals are in bold or in single quotes.

```
query ::= select [from] [where] [orderby]

select ::= SELECT ( '*' | proplist )

from ::= FROM ntlist

where ::= WHERE whereexp

orderby ::= ORDER BY [DESCENDING|ASCENDING] propname

proplist ::= propname { ',' propname }

ntlist ::= ntname { ',' ntname }

whereexp ::= propname op value |
            propname IS NULL |
            propname IS NOT NULL |
            value IN propname |
            like |
            contains |
            whereexp AND whereexp |
            whereexp OR whereexp |
            NOT whereexp |
            '(' whereexp ') '

op ::= '=' | '>' | '<' | '>=' | '<=' | '<>'

propname ::= joinpropname | simplepropname

joinpropname ::= ntname '.jcr:path'
```

```

simplepropname ::= quotedpropname | unquotedpropname

quotedpropname ::= "'" unquotedpropname "'"

unquotedpropname ::= /* A property name, possible a
pseudo-property: jcr:score or jcr:path */

ntname ::= quotedntname | unquotedntname

quotedntname ::= "'" unquotedntname "'"

unquotedntname ::= /* A node type name */

value ::= "'" literalvalue "'" | literalvalue

literalvalue ::= /* A property value (in standard string
form) */

like ::= propname LIKE likepattern

likepattern ::= "'" likechar { likepattern } "'"

likechar ::= char | '%' | '_'

char ::= /* Any character valid within the
string representation of a value
except for the characters % and _
themselves. These must be escaped */

contains ::= CONTAINS( searchexp ')'

searchexp ::= "'" [-]term {[OR}[-]term} "'"

term ::= word | "'" word { ' ' word } "'"

word ::= /* A string containing no whitespace */

```

8.6.4 SQL Syntax in Detail

A SQL statement for JCR query is composed of a **SELECT** clause optionally followed by up to three more clauses: a **FROM** clause, a **WHERE** clause and an **ORDER BY** clause.

8.6.4.1 SELECT

The **SELECT** clause specifies a list of properties (columns) by name, or the wildcard character ("*") to mean "all properties". Notice the special case of **joinpropname**, this provides for joins, but only on the **jcr:path** column, as described in [8.6.3 Database View](#).

8.6.4.2 FROM

The **FROM** clause narrows the search to include only the specified tables (node types). For example, the query

```
SELECT myapp:productName FROM mynt:shippable
```

would return all properties in the workspace called `myapp:product` that belong to nodes of node type `mynt:shippable`. Note that when a node type is specified in **FROM** clause the “tables” (in the database view) searched will include the table corresponding to the named node type plus the tables corresponding to all subtypes of that node type. For example,

```
FROM nt:hierarchyNode
```

is equivalent to

```
FROM nt:hierarchyNode, nt:file, nt:linkedFile, nt:folder
```

(assuming that all and only the predefined primary node types have been defined in the repository, see [6.7.22](#) *Predefined Primary Node Types*).

To search all node types (tables) one would specify **FROM nt:base** (though an implementation can always prevent searches whose scope is unfeasibly large).

8.6.4.3 WHERE

The **WHERE** clause allows you to place constraints on the nodes (rows) returned by specifying values or ranges of values for the properties (columns) of those nodes (rows). For example, the query,

```
SELECT myapp:image FROM mynt:document WHERE height < 100  
AND keyword LIKE "%apple%"
```

would find all the properties called `myapp:image` of nodes of type `mynt:document` (and subtypes) that also have:

- a property called `height` with value less than 100 and
- a property called `keyword` with substring “apple”.

The evaluation order within an expression is:

- **(...)** (Parentheses)
- **<, >, =, <=, >=, <>, LIKE, IN, IS NULL, IS NOT NULL** (Operators)
- **CONTAINS** (Function)
- **NOT** (Logical negation)
- **AND** (Logical conjunction)
- **OR** (Logical disjunction)

Literal values of types **NAME**, **PATH** and **STRING** must be enclosed in single quotes. Any literal single quote within the pattern must be escaped as two consecutive single quotes.

The collation sequence used when comparing **STRING** values using **>**, **<**, **>=** or **<=** is implementation-specific.

For types **LONG** and **DOUBLE** comparisons are done by numeric value, not string representation.

In case of type mismatches in a comparison **LONGS** can be converted to **DOUBLES**; for any other type mismatch each operand is converted to **STRING** (see [6.2.6 Property Type Conversion](#)) and compared using the established collation sequence.

See also [6.6.3.10 Searching Multi-value Properties](#), above, in the XPath section.

8.6.4.4 LIKE

Within the **WHERE** clause, the **LIKE** operator is used to pattern match the string form of a property's value. Its argument (**likepattern**) may contain the percent ("%") and underscore ("_") characters.

The **likepattern** must be enclosed in single quotes. Any literal single quote within the pattern must be escaped as two consecutive single quotes.

A **likepattern** with neither a percent ("%") nor an underscore ("_") character makes the **LIKE** operator equivalent to an equals ("=") operator.

Within the **likepattern**, literal instances of percent ("%") or underscore ("_") must be escaped using the SQL **ESCAPE** clause which allows the definition of an arbitrary escape character within the context of a single **LIKE** statement. For example

```
SELECT * FROM mytype WHERE a LIKE 'foo\%' ESCAPE '\'
```

the above statement will select nodes of type 'mytype' with property 'a' that contain exactly the value 'foo%'

8.6.4.5 CONTAINS

Within the **WHERE** clause, the **CONTAINS** function is used to embed a statement in a full-text search language. At minimum, all implementations must support the *simple search-engine syntax* defined by **searchexp** in the EBNF above. This syntax is based on the syntax of search engines like Google.

The semantics of the simple search expression are as follows:

- Terms separated by whitespace are implicitly **AND**ed together.
- Terms may also be **OR**ed with explicit use of the **OR** keyword.

- **AND** has higher precedence than **OR**.
- Terms may be excluded by prefixing with a **-** (minus sign) character. This means that the result set *must not* contain the excluded term.
- A term may be either a single word or a phrase delimited by single quotes (`' '`).
- The entire text search expression (**searchexp** in the EBNF, above) *must be* delimited by double quotes (`" "`).
- Within the **searchexp** literal instances of single quote (`' '`), double quote (`" "`) and hyphen (`-`) must be escaped with a backslash (`\`). Backslash itself must therefore also be escaped, ending up as double backslash (`\\`).

The scope of the **CONTAINS** clause is the intersection of two sets. These two sets are:

- The values of those properties that are the immediate children of the nodes specified by the **FROM** clause and other subclauses of **WHERE**.
- The contents of the full-text index of the repository. A repository may, for example, index only the values of **STRING** properties. Additionally, it may index some binary properties according to some application-specific encoding. The scope of full-text indexing is implementation specific.

For example, the query,

```
SELECT * FROM mynt:product WHERE
CONTAINS("apples 'good for you' -oranges")
```

would return a result containing all properties of all nodes of type **mynt:product** that have a property whose value contains the string "apples", the string "good for you" and does not contain the string "oranges".

The relevance score for each matching node is returned as the pseudo-property **jcr:score** (if supported, see [6.1.1.1 Repository Descriptors](#)). The specification does not define the range of the **jcr:score** value, it is implementation specific.

An implementation may support other embedded full-text search languages other than the simple search engine style shown here.

8.6.4.6 IN

In the **WHERE** clause the syntax of

```
v IN p
```

is used to specify that the value `v` must be among those in property `p`.

If `p` is multi-valued then the expression will evaluate to `true` if at least one value among those in `p` is equal to `v`.

If `p` is single valued then the expression is equivalent to `p = v`.

8.6.4.7 ORDER BY

The **ORDER BY** clause causes the returned `QueryResult` objects to be sorted according to the value of a particular property.

If the **CONTAINS** function is used (thus invoking full text search) a special `jcr:score` column may be added to the results. The **ORDER BY** clause can be especially useful for ordering the results by this value.

If neither **ASCENDING** nor **DESCENDING** are specified, the default is **ASCENDING** (least value first).

The collation sequence used when an **ORDER BY** is applied to a **STRING** property is implementation-specific.

8.6.5 Query Results

Results are returned in the same structure as with XPath queries. See [6.6.11 Query Results](#).