Content Repository API for Java™ Technology Specification

Java Specification Request 170

version 1.0.1

1 March 2006

CONTENT REPOSITORY API FOR JAVA™ TECHNOLOGY SPECIFICATION

Acknowledgements

License

Day Management AG ("Licensor") 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.

Content Repository for Java™ Technology API Specification ("Specification")
Version: 1.0
Status: FCS
Release: 11 May 2005

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

NOTICE; LIMITED LICENSE GRANTS

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

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

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

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

5. Definitions. For the purposes of this Agreement: "Independent Implementation" shall mean an implementation of the Specification that neither derives from any of Licensor's source code or binary code materials nor, except with an appropriate and separate license from Licensor, includes any of Licensor's source code or binary code materials; "Licensor Name Space" shall mean the public class or interface declarations whose names begin with "java", "javax", "javax.jcr" or their equivalents in any subsequent naming convention adopted by Licensor through the Java Community Process, or any recognized successors or replacements thereof; and "Technology Compatibility Kit" or "TCK" shall mean the test suite and accompanying TCK User's Guide provided by Licensor which corresponds to the particular version of the Specification being tested.

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

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

8. Disclaimer of Warranties. The Specification is provided "AS IS". LICENSOR MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR

PURPOSE, NON-INFRINGEMENT (INCLUDING AS A CONSEQUENCE OF ANY PRACTICE OR IMPLEMENTATION OF THE SPECIFICATION), OR THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE. This document does not represent any commitment to release or implement any portion of the Specification in any product.

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

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

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

1 Preface

This is version 1.0.1 of the Content Repository API for Java Technology specification (Java Specification Request 170, Maintenance Release 1).

1.1 Documents Included

The specification includes:

- This document in Adobe Portable Document Format (*jsr170-1.0.1.pdf*).

- 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 Content Repository for Java Technology 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 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 as well as a set of optional features.

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 a large portion of the existing code-base of content-related applications. Level 1 is also designed to be easy to implement on top of any existing content repository.

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 Content Repository for Java Technology API 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 this specification in a Java framework around JSP.
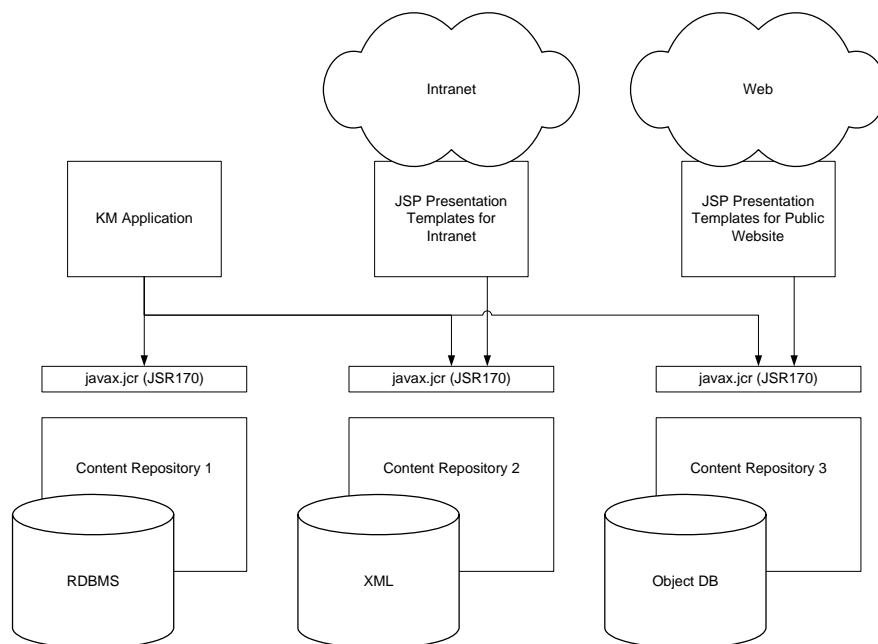
A year later, two divisions switch their CM vendors. As part of the migration process, the KM discovery application is tested against the new CMS in those divisions. Other than having its configuration file updated, the application works as before.

Through their standardized interface these repositories allow 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 through a simple export and import.
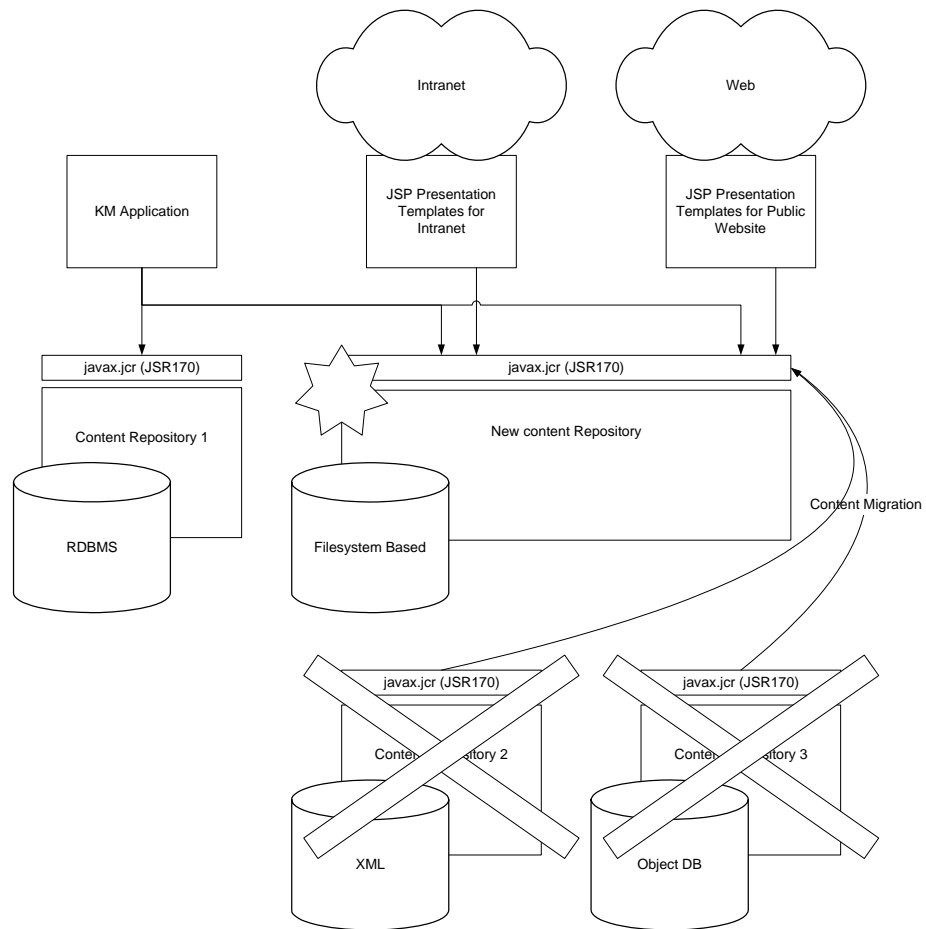
Furthermore, all the content presentation templates developed against this specification keep working as before. In the case of ENT the development of this presentation logic was one of the major investments made when the intranet and the public Internet website were first developed. Being able to migrate 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.
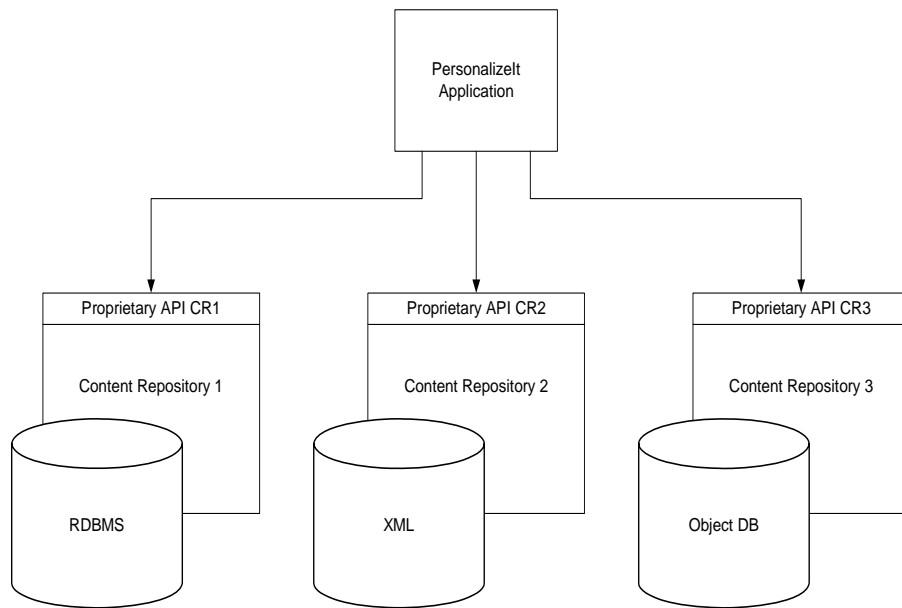
Since the adoption of this specification, PersonalizeIt offers a standard interface to compliant repositories. Therefore, they no longer have to maintain dozens of proprietary integrations and can concentrate on their core competency.
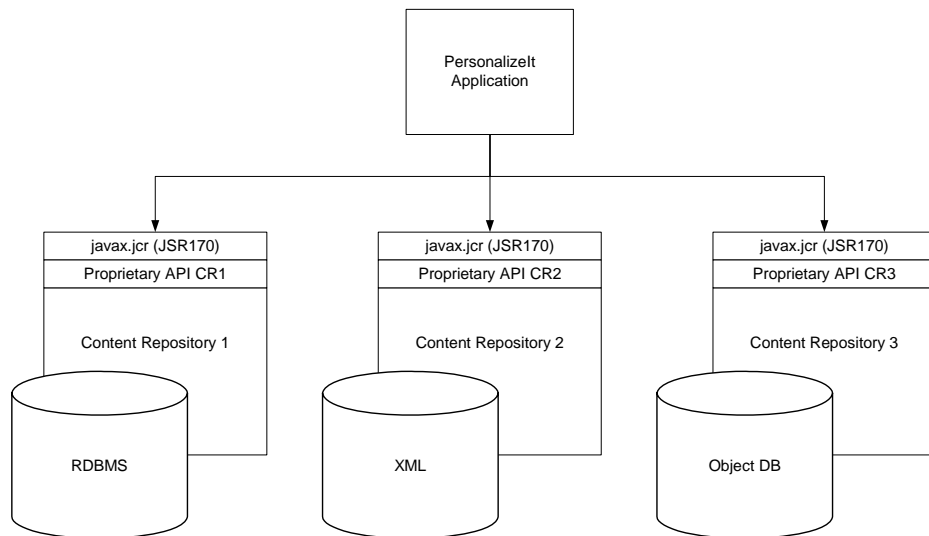
4 The Repository Model

A content 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 per workspace, 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. All of the actual content in the repository is stored within the values of the properties.
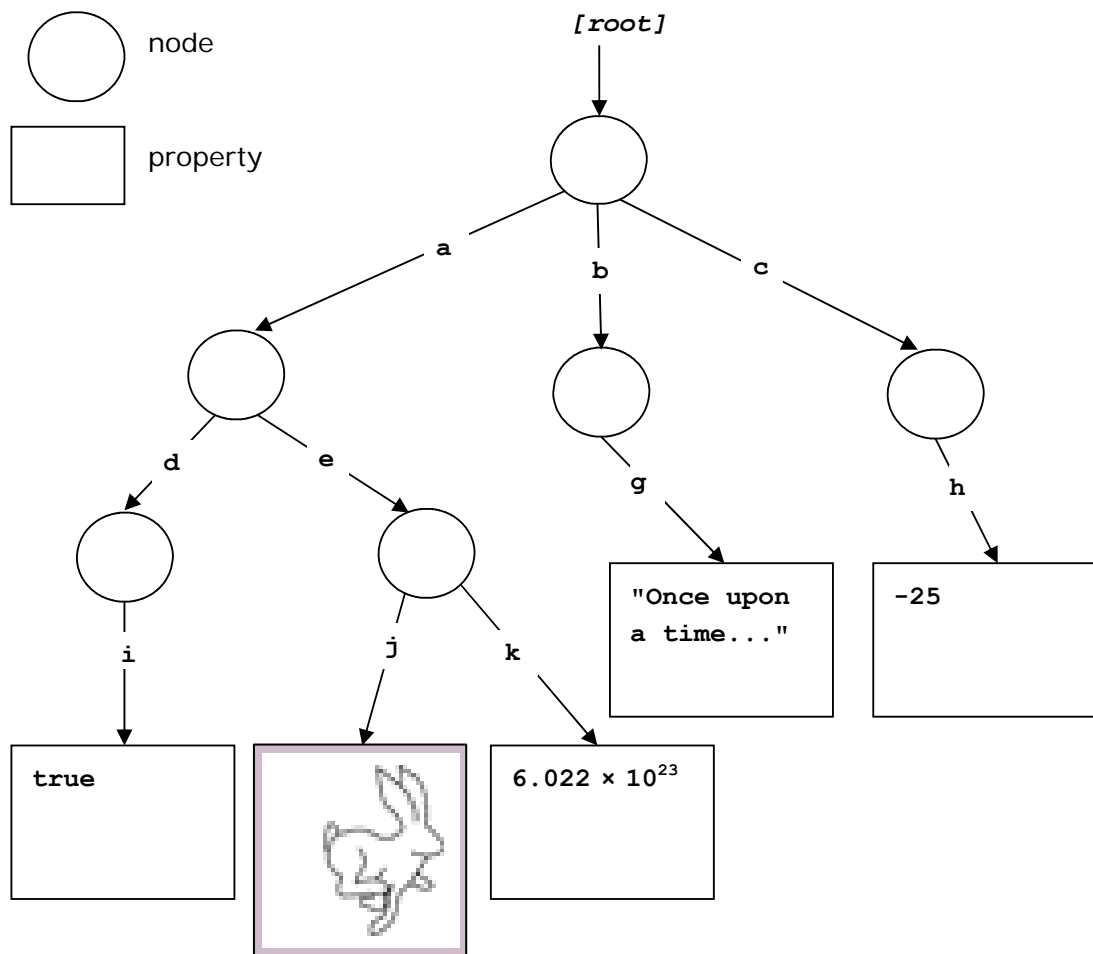
In the diagram above, we see the single root node of some workspace with child nodes **a**, **b** and **c**, each of which have further child nodes or properties. For example, the node **a** has two child nodes, **d** and **e**. Node **e**, in turn, has two properties, **j** and **k**. The property **j** 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**).

Every item in a workspace apart from the root node has a non-empty *name*. The root node always has the empty string as its name.

A property and node with the same parent cannot have the same name. Two or more nodes with the same parent *may* in some circumstances have the same name, in which case they are distinguished by *index* (see 4.3 *Same-Name Siblings* for more details).

The chain of names from the root to any item defines the *absolute path* of that item. 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** object[1]. A client connects to the repository by calling **Repository.login** optionally specifying a workspace name and a **Credentials** object[2]. 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
InitialContext ctx = ...
Repository repository = (Repository)ctx.lookup("myrepo");
```

[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.

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

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 x 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 x 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 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 x 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 string 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 (and its subtree)
myNode.remove();

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

would result in the node at **/a/e** (and its child node **/a/e/n**) being deleted.

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

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` and other methods that act directly on the persistent workspace will not *make 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`.

---

[3] A few `Node` methods act immediately on the persistent workspace and do not require `save`. See 7.1 *Writing Repository Content* for details.

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.

```
                    Item                    *
                                     ◁─────  child
                                                    1
         △                △
    Property    *         1    Node
                ─────────────▶
                parent
                              ───▶
                       0..1
                       parent              *
```
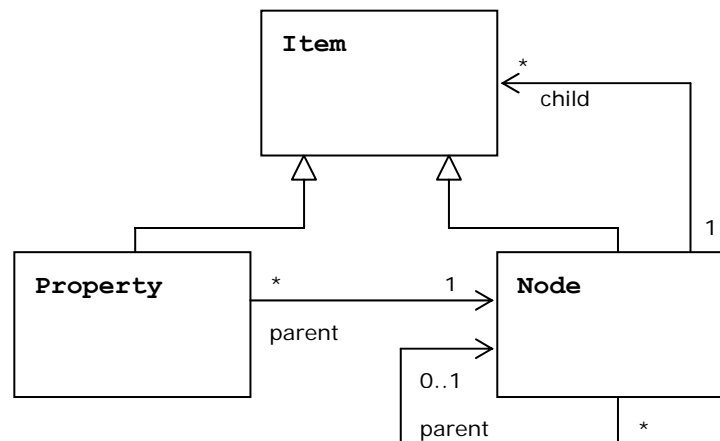
This UML diagram indicates that **Property** and **Node** are subinterfaces of **Item**. A single **Property** has one and only one parent **Node**. A **Node** can have either zero parents (only if it is the root node) or one parent. A **Node** can have any number of child **Item** objects (i.e., either **Property** or **Node** objects).

4.2 Compliance Levels

This specification is divided into two compliance levels and 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

- Discovery of access control permissions

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

Optional:

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

- Transactions

- Versioning

- Observation (Events)

- 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*).

However, as opposed to the semantics of XPath, a name in a content repository path that does not explicitly specify an index implies an index of `1`. For example, `/a/b/c` is equivalent to `/a[1]/b[1]/c[1]`.

The indexing is based on the order in which child nodes are returned in the iterator acquired through `Node.getNodes()`.

Same-name siblings are indexed by their position relative to each other in this larger ordered set. For example, the order of child nodes returned by a `getNodes` on some parent might be:

`[A, B, C, A, D]`

In this case, `A[1]` refers the first node in the list and `A[2]` refers to the fourth node in the list.

Note that regardless of whether orderable child nodes are supported in general (see 4.4 *Orderable Child Nodes*), the relative ordering of a set of same name sibling nodes must be persistent; it cannot change arbitrarily between read method calls or between sessions. This requirement stems from the fact that the path of a node must not change arbitrarily, and in the case of a same-name sibling, its position relative to its co-named siblings defines part of its path.

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 orderable by the application 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[2]","A[1]")`

will cause the child node currently called `A[2]` to be moved to the position before the child node currently called `A[1]`, the resulting order will be:

`[A, A, B, C, D]`

where the first `A` is the one that was formerly after `C` and the second `A` is the one that was formerly at the head of the list.

Note, however, that after the completion of this operation *the indices of the two nodes have now switched*, due to their new positions relative to each other. What was formerly `A[2]` is now `A[1]` and what was formerly `A[1]` is now `A[2]`.

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. The only exception to this rule is that same-name siblings must maintain their relative order across read method invocations and across sessions.

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 a content repository 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 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.

- **xml** Reserved for reasons of compatibility with XML.

- "" (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 ::= abspath | relpath

abspath ::= '/' relpath | '/'

relpath ::= pathelement | relpath '/' pathelement

pathelement ::= name | name '[' number ']' | '..' | '.'

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

name ::= [prefix ':'] simplename

simplename ::= onecharsimplename |
               twocharsimplename |
               threeormorecharname

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

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

```
                           onecharsimplename onecharsimplename

threeormorecharname ::= nonspace string nonspace

prefix ::= /* Any valid non-empty XML NCName */

string ::= char | string char

char ::= nonspace | ' '

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

4.6.1 Names vs. Paths

A "name" is valid if satisfies the above **name** production. It can be thought of, informally, as a single path element *without any square-bracket index* (and not including the '.' and '..'). 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).

4.6.2 Current Item and Parent Item

The syntax of paths includes the segments "**.**" and "**..**" indicating current item and parent item, respectively. These can be used within JCR paths just as they can in Unix-like file system paths. For example, **/a/b/../c** is equivalent to **/a/c** while **/a/b/./c** is equivalent to **/a/b/c**.

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).

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 7.1.5 *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 persist the removal of a node (that is, either on **save**, or, if the change was made within a transaction, on *commit*; in any case, the check 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(String name, 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

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 7.1.5 *Adding and Writing Properties*.

4.8 Node Types

Every node *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 a node is allowed (or required) to have. Every node 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 can 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 this API is 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 a Version*).

4.10 Workspaces

A content 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*) 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:
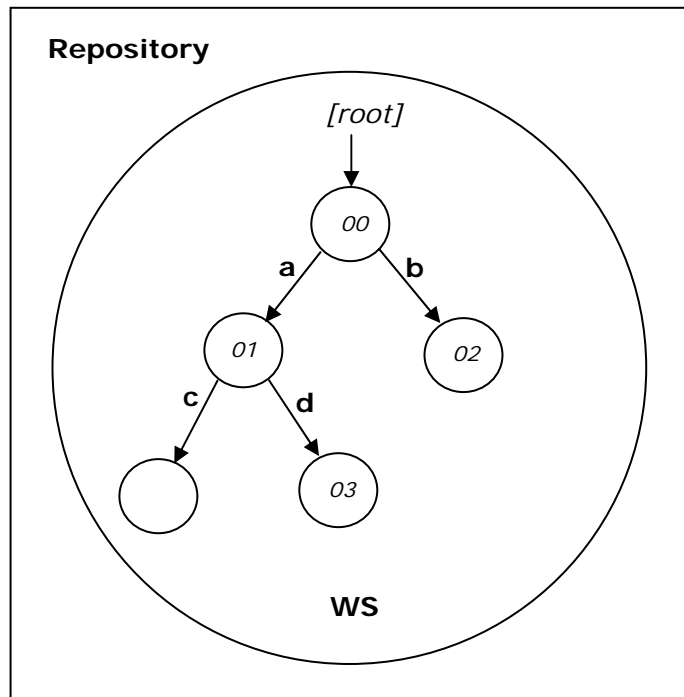
The small circles represent nodes. The arrows point from parent to child and are labeled with the name of the child. The name of the root node is actually the empty string though, for clarity, it is indicated here with the string "*[root]*". The numbers within the nodes represent the UUIDs of the nodes. 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 is defined as follows:

- A node's corresponding nodes are those with the same *correspondence identifier*.

- The correspondence identifier of a referenceable node is its UUID.

- The correspondence identifier of a non-referenceable node with at least one referenceable ancestor is the pair consisting of the UUID of its nearest referenceable ancestor and its relative path from that ancestor.

- The correspondence identifier of a non-referenceable node with no referenceable ancestor is its absolute path.

Recall also that (as stated in 4.9 *Referenceable Nodes*) if a repository has a workspace with a referenceable root node then *all*

its workspaces must have referenceable root nodes *and* those root nodes must all have the same UUID.

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

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 its subtree to be replaced by a clone of this nodes corresponding node and its subtree in **srcWorkspace**.

For more details on corresponding nodes and the update method see 7.1.8 *Updating and Cloning Nodes across Workspaces*.

4.10.2.1 Example

The following diagram shows a schematic of a two-workspace repository.
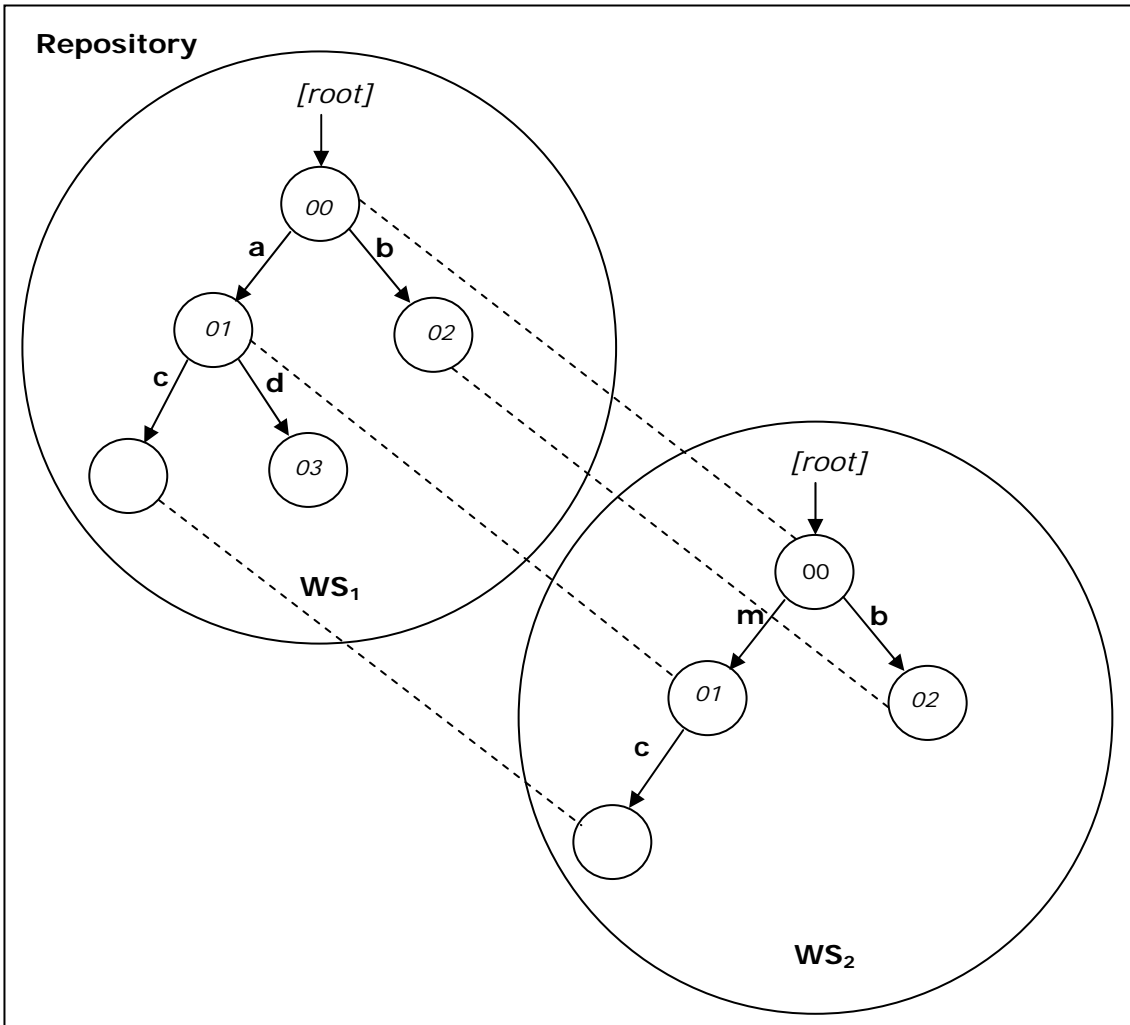
Here we see two workspaces, WS_1 and WS_2. The dotted lines indicate corresponding nodes. For example, the node **/a** in WS_1 corresponds to **/m** in WS_2 because both have a UUID of *01.* Similarly, **/b** in **WS1** corresponds with **/b** in **WS2**. In these cases, because the nodes are referenceable, 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.
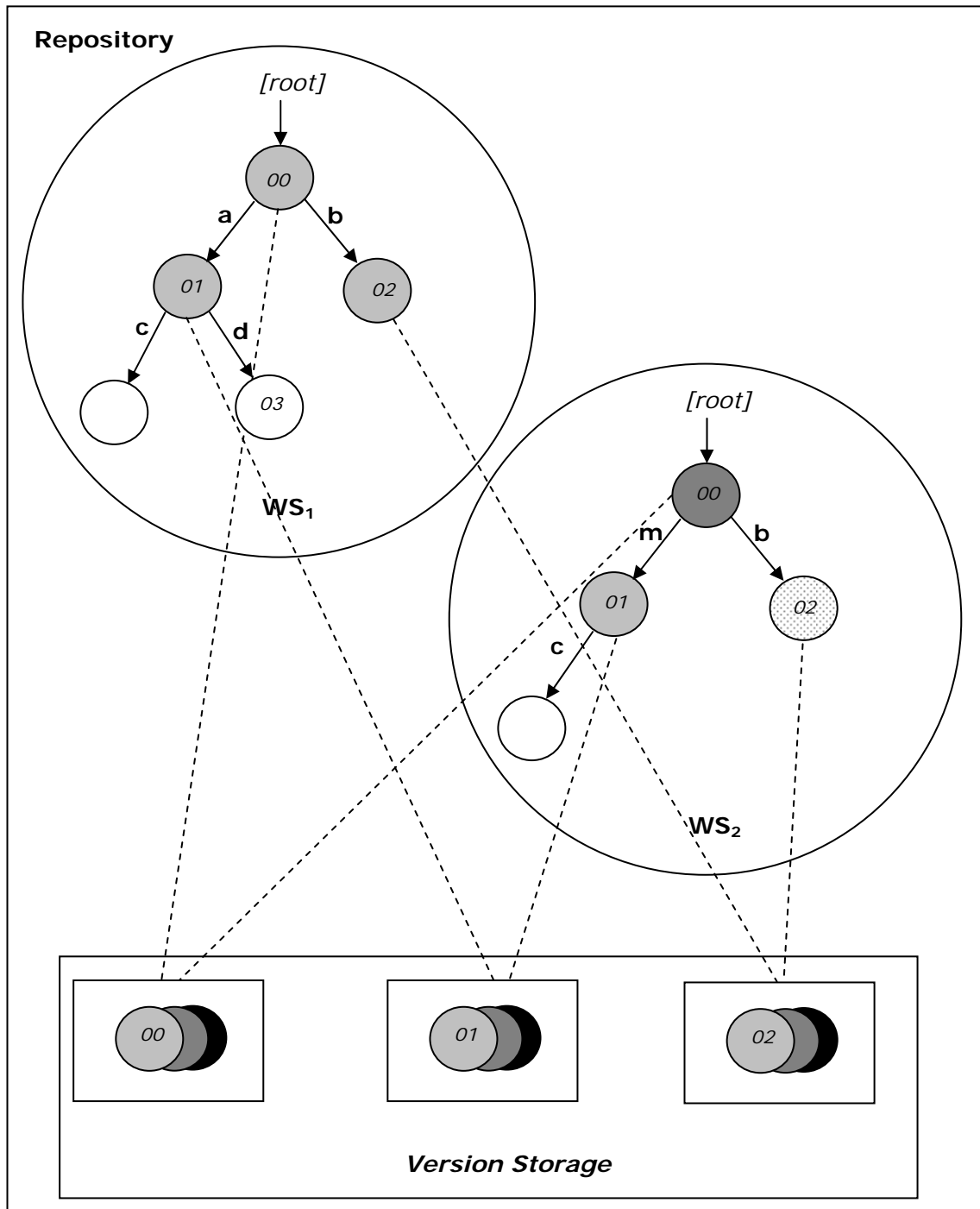
This diagram shows a repository that supports versioning and contains two workspaces. The version storage is represented by the area in the bottom. It contains a version history for each versionable node in the repository. The versionable nodes in the workspaces are shown in various shadings. The nonversionable 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 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 primary item of a particular node (if it has one) is declared in its node type.

4.13 Hierarchical versus Direct Access

Though this 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 this specification ensures that this need not be a particularly restrictive requirement. The following strategies, amongst others, 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 A File System-backed Content Repository

An obvious implementation of a content repository 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 the content repository:

```
                                             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  │   ├─jcr:created = "2001-05-03T00:00:00.000Z"
        whichever was first)           │   │
```

\<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** combined with application-specific content node types are used).

5.2 A WebDAV-backed Content Repository

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 repository properties.

	`/`
/newpaintings/	└─`newpaintings`
bigredstripe.gif	└─`bigredstripe.gif`
\<PROPFIND/newpaintings/bigredstripe.gif\>	
DAV:displayname	├─`myapp:name = "Big Red Stripe"`
DAV:creationdate	├─`jcr:created = "2001-05-03` `T00:00:00.000Z"`
	└─`jcr:content`
DAV:getlastmodified	├─`jcr:lastModified = "2001-05-03` `T00:00:00.000Z"`
DAV:getcontenttype	├─`jcr:mimeType = "image/gif"`
\<GET /newpaintings/bigredstripe.gif\>	└─`jcr:data = <binary data>`

In this example, the file **bigredstripe.gif** is represented by a subtype of **nt:file** and the **jcr:content** subnode is of type **nt:resource**.

5.3 Database-backed Content Repository

A 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
`<jcr:root>`	0	0
`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 XML-backed Content Repository

Another possible implementation 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
<?xml version="1.0" encoding="UTF-8"?>
<content>
  <title>Our Products</title>
  <lead>Geometrixx is proud to offer...</lead>
  <paragraph>
    <text>Geometrixx 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
<?xml version="1.0" encoding="UTF-8"?>
<content>
  <title>Rhombus: The shape of things to come!</title>
  <price>123.00</price>
  <lead>Here at Geometrixx...</lead>
  <paragraph>
    <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 content repository would be:

	Node Property = "Some value"
/	*[root]*
products.xml	├─products
[creation date of the file]	├─jcr:created = "2001-01-01T..."
<?xml version="1.0"...?> <content>	├─jcr: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"...?> <content>	└─jcr: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"...?> <content>	├─jcr: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 the repository and XML (see, for example, 6.4 XML 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 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:** These include the file and directory names **products** and **rhombus** in example 5.4. Since in these examples these names are taken directly from the underlying resources, they happen not to be namespaced. In general however, a mapping to namespaced names could just as well be used.

For more details see 6.3 *Namespaces*.

6 Level 1 Repository Features

The following section explains level 1 of the 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 acquired through 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 authorization to access that workspace is granted, then a new **Session** object is returned. The format of the string **workspaceName** depends upon the implementation.
	
	If **credentials** is **null**, 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.
	
	If **workspaceName** is **null**, 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 **Workspace.getAccessibleWorkspaceNames**, allowing the client to select from among available "real" workspaces (see 6.2.2 *Workspace Read Methods*).
	
	If authentication or authorization for the specified workspace fails, a **LoginException** is thrown.
	
	If **workspaceName** is not recognized, a **NoSuchWorkspaceException** is thrown.
	
	A **RepositoryException** is thrown if another error occurs.

Session	**login(Credentials credentials)** Equivalent to **login(credentials, null)**.
Session	**login(String workspaceName)** Equivalent to **login(null, workspaceName)**.
Session	**login()** Equivalent to **login(null, null)**.
String[]	**getDescriptorKeys()** 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 **Repository.getDescriptor(String name)** to query information about this repository implementation.
String	**getDescriptor(String key)** 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 **getDescriptorKeys**. If the specified key is not found, **null** is returned.

6.1.1.1 Repository Descriptors

The methods **Repository.getDescriptorKeys** 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
SPEC_VERSION_DESC	For this specification the value of this descriptor is "**1.0**".
SPEC_NAME_DESC	For this specification the value of this descriptor is "**Content Repository for Java Technology API**".
REP_VENDOR_DESC	The name of the vendor of this repository implementation.
REP_VENDOR_URL_DESC	The URL of the repository vendor.

`REP_NAME_DESC`	The name of this repository implementation.
`REP_VERSION_DESC`	The version of this repository implementation.
`LEVEL_1_SUPPORTED`	Indicates whether this implementation supports all level 1 features. This descriptor should always be "**true**".
`LEVEL_2_SUPPORTED`	Indicates whether this implementation supports all level 2 features. This descriptor will be either "**true**" or "**false**".
`OPTION_TRANSACTIONS_SUPPORTED`	Indicates whether this implementation supports transactions. This descriptor will be either "**true**" or "**false**".
`OPTION_VERSIONING_SUPPORTED`	Indicates whether this implementation supports versioning. This descriptor will be either "**true**" or "**false**".
`OPTION_OBSERVATION_SUPPORTED`	Indicates whether this implementation supports observation. This descriptor will be either "**true**" or "**false**".
`OPTION_LOCKING_SUPPORTED`	Indicates whether this implementation supports locking. This descriptor will be either "**true**" or "**false**".
`OPTION_QUERY_SQL_SUPPORTED`	Indicates whether this implementation supports queries using the SQL language. This descriptor will be either "**true**" or "**false**".
`QUERY_XPATH_POS_INDEX`	Indicates whether this implementation supports index position notation for same-name siblings within XPath queries. This descriptor will be either "**true**" or "**false**". See 6.6.4.1 *Adapting XPath to the Content Repository::Same-Name Siblings* for more details.
`QUERY_XPATH_DOC_ORDER`	Indicates whether this implementation returns the results of XPath queries in document order. This descriptor will be either "**true**" or "**false**". See 6.6.4.2 *Adapting XPath to the Content Repository::Document Order*.

Implementations may add additional descriptors.

6.1.1.2 Thread Safety of Repository Methods

An implementation is required to provide thread-safe implementations of all the methods of the **Repository** interface. Note that this requirement is of more relevance in a level 2 implementation than a pure level 1 implementation. See 7.5 *Thread-Safety Requirements.*

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 `Session`s access a particular workspace 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**.
Node	**getRootNode()** Returns the root node of the workspace, **/**. This node is the main access point to the content of the workspace. A **RepositoryException** is thrown if an error occurs.
Item	**getItem(String absPath)**

| | Returns the item at the specified absolute path in the workspace.

A **PathNotFoundException** is thrown if no item at **absPath** exists.

A **RepositoryException** is thrown if another error occurs. |
|---|---|
| **boolean** | **itemExists(String absPath)**

Returns **true** if an item exists at **absPath** and this **Session** has read access to it; otherwise returns **false**.

A **RepositoryException** is thrown if **absPath** is not a well-formed absolute path. |
| **Node** | **getNodeByUUID(String uuid)**

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**.

An **ItemNotFoundException** is thrown if no item with the specified UUID exists.

A **RepositoryException** is thrown if another error occurs. |
| **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 have sufficient permissions to perform the operation.

A **RepositoryException** is thrown if another error occurs. |
| **void** | **logout()**

Releases all resources associated with this **Session**. This method should be called when a **Session** is no |

	longer needed.
boolean	**isLive()** Returns **true** if this **Session** object is usable by the client. A usable **Session** object is one that is neither logged-out, timed-out nor in any other way disconnected from the repository.

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. The following are the level 1 methods of **Workspace**. **Workspace** has other methods as well, though 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 the name of the desired workspace, and receives a new **Session** object.
	A **RepositoryException** is thrown if an error occurs.

6.2.3 Node Read Methods

The following are the level 1 methods of **Node**. They are used for getting the child nodes and properties of a node. The **Node** interface has other methods as well, though in a level 1-only implementation they will either do nothing or throw an exception.

javax.jcr. **Node**	
Node	**getNode(String relPath)**
	Returns the node at **relPath** relative to this node.
	If **relPath** contains a path element that refers to a node with same-name sibling nodes without explicitly including an index using the array-style notation (**[x]**), then the index **[1]** is assumed (See 4.3 *Same-Name Siblings*).
	Within the scope of a single **Session** object, if a node has been acquired with **getNode**, any subsequent call of **getNode** reacquiring the same node must return a **Node** object reflecting the same state as the earlier **Node** object. Whether this object is actually the same **Node** instance, or simply one wrapping the same state, is up to the implementation. See 7.1.3.1 *Re-using Item Objects*.
	If no node exists at **relPath** a **PathNotFoundException** is thrown.

	A **RepositoryException** is thrown if another error occurs.
NodeIterator	**getNodes()** Returns all child nodes of this node. Does *not* include properties of this node. The same reacquisition semantics apply as with **getNode**. If this node has no child nodes, then an empty iterator is returned. A **RepositoryException** is thrown if another error occurs.
NodeIterator	**getNodes(String namePattern)** Gets all child nodes 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, **N.getNodes("jcr:* | myapp:report | my doc")** would return a **NodeIterator** holding all child nodes of **N** that are either called '**myapp:report**', begin with the prefix '**jcr:**' or are called '**my doc**'. Note that leading and trailing whitespace *around* a | character is ignored, but whitespace *within* a disjunct forms part of the pattern to be matched. The EBNF for **namePattern** is: `namePattern ::= disjunct {'|' disjunct}` `disjunct ::= name [':' name]` `name ::= '*' |` `        ['*'] fragment {'*' fragment}['*']` `fragment ::= char {char}` `char ::= nonspace	' '` `nonspace ::= (* Any Unicode character except:` ` '/', ':', '[', ']', '*',` ` ''', '"', '|' or any whitespace` ` character *)` The pattern is matched against the names (not the paths) of the immediate child nodes of this node. If this node has no matching child nodes, then an empty iterator is returned. The same reacquisition semantics apply as with **getNode**. A **RepositoryException** is thrown if an error occurs.

Property	**getProperty(String relPath)**
	Get the property at **relPath** relative to this node. The same reacquisition semantics apply as with **getNode**.
	If no property exists at **relPath** a **PathNotFoundException** is thrown.
	A **RepositoryException** is thrown if another error occurs.
PropertyIterator	**getProperties()**
	Gets all properties of this node. Does *not* include child *nodes* of this node. The same reacquisition semantics apply as with **getNode**. If this node has no properties, then an empty iterator is returned.
	A **RepositoryException** is thrown if an error occurs.
PropertyIterator	**getProperties(String namePattern)**
	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,
	N.getProperties("jcr:* | myapp:name | my doc")
	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**'.
	Note that leading and trailing whitespace *around* a disjunct is ignored, but whitespace *within* a disjunct forms part of the pattern to be matched.
	The EBNF for **namePattern** is:
| | ```
namePattern ::= disjunct {'|' disjunct}
disjunct ::= name [':' name]
name ::= '*' |
         ['*'] fragment {'*' fragment}['*']
fragment ::= char {char}
char ::= nonspace | ' '
nonspace ::= (* Any Unicode character except:
              '/', ':', '[', ']', '*',
              ''', '"', '|' or any whitespace
              character *)
``` |
| | The pattern is matched against the names (not the paths) of the immediate child properties of this node. |
| | If this node has no matching properties, then an |

	empty iterator is returned. The same reacquisition semantics apply as with **getNode**. A **RepositoryException** is thrown if an error occurs.
Item	**getPrimaryItem()** The primary node type (see 6.7 *Node Types*) of this node may specify one child item (child node or property) of this node as the *primary child item*. This method returns that item. In cases where the primary child item specifies the name of a set of same-name sibling child nodes, the node returned will be the one among the same-name siblings with index **[1]**. The same reacquisition semantics apply as with **getNode**. If this node has no primary child item, either because none is declared in the node type or because a declared primary item is not present on this node instance, then this method throws an **ItemNotFoundException**. A **RepositoryException** is thrown if another error occurs.
String	**getUUID()** Returns the UUID of this node as recorded in the node's **jcr:uuid** property. This method only works on nodes of mixin node type **mix:referenceable**. On nonreferenceable nodes, this method throws an **UnsupportedRepositoryOperationException**. To avoid throwing an exception to determine whether a node has a UUID, a call to **isNodeType("mix:referenceable")** can be made. A **RepositoryException** is thrown if another error occurs.
int	**getIndex()** 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., **/a[3]/b[4]**. 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**.
PropertyIterator	**getReferences()** Returns all **REFERENCE** properties that refer to this node. Some level 2 implementations may only return properties that have been saved (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). Other level 2 implementations may additionally return properties that have been added within the current **Session** but are not yet saved. In implementations that support versioning, this method does not return **REFERENCE** properties that are part of the frozen state of a version in version storage. If this node has no references, an empty iterator is returned. A **RepositoryException** is thrown if an error occurs.
boolean	**hasNode(String relPath)** Returns **true** if a node exists at **relPath** and **false** otherwise.
boolean	**hasNodes()** Returns **true** if this node has one or more child nodes. Returns **false** otherwise.
boolean	**hasProperty(String relPath)** Returns **true** if a property exists at **relPath** and **false** otherwise.
boolean	**hasProperties()** Returns **true** if this node has one or more properties. Returns **false** otherwise.

6.2.4 Property Read Methods

The following are the level 1 methods of **Property**. They are used for reading a property. The **Property** interface has other methods as well, though in a level 1-only implementation they will either do nothing or throw an exception.

javax.jcr. **Property**	
`Value`	**getValue()** Returns the value of this property as a **Value** object. If this property is multi-valued, this method throws a **ValueFormatException**. The object returned is a copy of the stored value and is immutable. A **RepositoryException** is thrown if an error occurs.
`Value[]`	**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 **ValueFormatException**. The array returned is a copy of the stored values, so changes to it are not reflected in internal storage. A **RepositoryException** is thrown if an error occurs.
`String`	**getString()** Returns a **String** representation of the value of this property. A shortcut for **Property.getValue().getString()**. See 6.2.7 *Value*. If this property is multi-valued, this method throws a **ValueFormatException**. If the value of this property cannot be converted to a **String**, a **ValueFormatException** is thrown. A **RepositoryException** is thrown if another error occurs.
`InputStream`	**getStream()** Returns an **InputStream** representation of the value of this property. A shortcut for **Property.getValue().getStream()**. See 6.2.7 *Value*. If this property is multi-valued, this method throws a **ValueFormatException**. A **RepositoryException** is thrown if another error

	occurs.
`long`	**`getLong()`** Returns a `long` representation of the value of this property. A shortcut for `Property.getValue().getLong()`. See 6.2.7 *Value*. If this property is multi-valued, this method throws a `ValueFormatException`. If the value of this property cannot be converted to a `long`, a `ValueFormatException` is thrown. A `RepositoryException` is thrown if another error occurs.
`double`	**`getDouble()`** Returns a `double` representation of the value of this property. A shortcut for `Property.getValue().getDouble()`. See 6.2.7 *Value*. If this property is multi-valued, this method throws a `ValueFormatException`. If the value of this property cannot be converted to a `double`, a `ValueFormatException` is thrown. A `RepositoryException` is thrown if another error occurs.
`Calendar`	**`getDate()`** Returns a `Calendar` representation of the value of this property. A shortcut for `Property.getValue().getDate()` See 6.2.7 *Value*. The object returned is a copy of the stored value, so changes to it are not reflected in internal storage. If this property is multi-valued, this method throws a `ValueFormatException`. If the value of this property cannot be converted to a `Calendar`, a `ValueFormatException` is thrown. A `RepositoryException` is thrown if another error occurs.
`boolean`	**`getBoolean()`** Returns a `boolean` representation of the value of this property. A shortcut for `Property.getValue().getBoolean()`. See 6.2.7

	Value. If this property is multi-valued, this method throws a `ValueFormatException`. If the value of this property cannot be converted to a `boolean`, a `ValueFormatException` is thrown. A `RepositoryException` is thrown if another error occurs.
`Item`	`getNode()` If this property is of type `REFERENCE` this method returns the `Node` to which this property refers. If this property is multi-valued, this method throws a `ValueFormatException`. If this property cannot be converted to a reference, then a `ValueFormatException` is thrown. 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`. A `RepositoryException` is thrown if another error occurs.
`long`	`getLength()` 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 to display the value in its string form as defined in 6.2.6 *Property Type Conversion.* Returns **–1** if the implementation cannot determine the length of the value. If this property is multi-valued, this method throws a `ValueFormatException`. A `RepositoryException` is thrown if another error occurs.
`long[]`	`getLengths()` 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 in its string form as defined in 6.2.6 *Property Type Conversion*). The order of the length values

	corresponds to the order of the values in the property.
	Returns a **–1** in the appropriate position if the implementation cannot determine the length of a value.
	If this property is single-valued, this method throws a **ValueFormatException**.
	A **RepositoryException** is thrown if another error occurs.
int	**getType()**
	Returns the type of this **Property**. The type returned is that which was set at property creation. Note that for some property **p**, the type returned by **p.getType()** may differ from the type returned by **p.getDefinition.getRequiredType()** 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.*

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 is used to store strings. It has the same characteristics as the Java **String** class.
int	**BINARY** **BINARY** properties are used to store binary data.
int	**LONG** The **LONG** property type is used to store integers. It has the same characteristics as the Java primitive type **long**.

`int`	**DOUBLE** The **DOUBLE** property type is used to store floating point numbers. It has the same characteristics as the Java primitive type **double**.
`int`	**BOOLEAN** The **BOOLEAN** property type is used to store boolean values. It has the same characteristics as the Java primitive type **boolean**.
`int`	**DATE** The **DATE** property type is used to store time and date information. See 6.2.5.1 *Date*.
`int`	**NAME** A **NAME** is a pairing of a *namespace* and a *local name*. When read, the namespace is mapped to the current prefix. See 6.2.5.2 *Name*.
`int`	**PATH** A **PATH** property is an ordered list of *path elements*. A path element is a **NAME** with an optional index. When read, the **NAME**s within the path are mapped to their current prefix. A path may be absolute or relative. See 6.2.5.3 *Path*.
`int`	**REFERENCE** A **REFERENCE** property stores the UUID of a referenceable node (one having type **mix:referenceable**), which must exist within the same workspace or session as the **REFERENCE** property. A **REFERENCE** property enforces this referential integrity by preventing (in level 2 implementations) the removal of its target node. See 6.2.5.4 *Reference*.
`int`	**UNDEFINED** 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 **Property.getType** and (in level 2 implementations) it cannot be assigned as the type when creating a new property.

String	`TYPENAME_STRING == "String"`
	`TYPENAME_BINARY == "Binary"`
	`TYPENAME_LONG == "Long"`
	`TYPENAME_DOUBLE == "Double"`
	`TYPENAME_DATE == "Date"`
	`TYPENAME_BOOLEAN == "Boolean"`
	`TYPENAME_NAME == "Name"`
	`TYPENAME_PATH == "Path"`
	`TYPENAME_REFERENCE == "Reference"`
	`TYPENAME_UNDEFINED == "Undefined"`
	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*.
String	`nameFromValue(int type)`
	Returns the standard name of the given property type, specified by its integer value.
int	`valueFromName(String name)`
	Returns the integer value of the given property type, specified by its standard name.

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 must be handled internally in such a way that 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 a **NAME** with fully qualified form

> *{http://mycorp.com/myapp}***myItem**

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**

Note however, that how the **NAME** value is stored internally is up to the implementation, as long as dynamic remapping is supported.

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 ::= [prefix ':'] simplename

simplename ::= onecharsimplename |
               twocharsimplename |
               threeormorecharname
```

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

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

threeormorecharname ::= nonspace string nonspace

prefix ::= /* Any valid non-empty XML NCName */

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 **NAME**s 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, given the namespace mapping

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

a **PATH** property value with fully qualified form

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

would be returned as the string

/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]

Note however, that how the **PATH** value is stored internally is up to the implementation, as long as dynamic remapping is supported.

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 ::= abspath | relpath

abspath ::= '/' relpath | '/'

relpath ::= pathelement | relpath '/' pathelement

pathelement ::= name | name '[' number ']' | '..' | '.'

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

name ::= /* see section 6.2.5.2 Name, above */
```

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 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 **getItem**, **getNode** or **getProperty** call.

6.2.5.4 Reference

A **REFERENCE** property stores the UUID of a referenceable node (one having type **mix:referenceable**). The referential integrity of **REFERENCE** properties must be guaranteed.

In level 2 implementations, enforcement of referential integrity means that when a node is **remove**d, a check must be done to ensure that no **REFERENCE** properties in the workspace still refer to nodes in the subtree to be removed. This check is done when an attempt is made to persist the removal of a node (that is, either on **save**, or, if the change was made within a transaction, on *commit*; in any case, the check is not done immediately on **remove**). If any references to a node in the subtree to be removed exist, a **ReferentialIntegrityException** 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. Throw on out-of-range.	Milliseconds since 1970-01-01T00:00:00Z. Throw on out-of- range.	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 represents the value of a 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.
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 **getString**. A **RepositoryException** is thrown if another error occurs.
InputStream	**getStream()** Returns an **InputStream** representation of this value. Uses the standard conversion to binary. If a non-stream **get** method 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 **getStream**. A **RepositoryException** is thrown if another error occurs.
long	**getLong()** Returns a **long** representation of this value. If this value cannot be converted to a **long**, a **ValueFormatException** is thrown. 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 **getLong**. A **RepositoryException** is thrown if another error occurs.
double	**getDouble()** Returns a **double** representation of this value. If this value cannot be converted to a **double**, a **ValueFormatException** is thrown. 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 **getDouble**. A **RepositoryException** is thrown if another error occurs.
Calendar	**getDate()** Returns a **Calendar** representation of this value. The object returned is a copy of the stored value, so

	changes to it are not reflected in internal storage.
	If this value cannot be converted to a **Calendar**, a **ValueFormatException** is thrown.
	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**.
	A **RepositoryException** is thrown if another error occurs.
boolean	**getBoolean()**
	Returns a **boolean** representation of this value.
	If this value cannot be converted to a **boolean**, a **ValueFormatException** is thrown.
	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**.
	A **RepositoryException** is thrown if another error occurs.
int	**getType()**
	Returns the type of this value. See 6.2.5 *Property Types*. The type returned is that which was set at property creation.

Implementations of **Value** must observe the following behavioral restrictions:

- A **Value** object can be read using type-specific **get** methods. These methods are divided into two groups:

    o The non-stream **get** methods **getString()**, **getDate()**, **getLong()**, **getDouble()** and **getBoolean()**.

    o **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()**.

6.2.7.1 Creating New Value Instances

In level 2 repositories, new **Value** instances are created using the **ValueFactory** object acquired through **Session.getValueFactory**. (see 7.1 **Writing Repository Content** and 7.1.5.3 **Creating Value Objects**).

6.2.7.2 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 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.

6.2.7.3 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. In a level 1-only implementation they will either do nothing or throw an exception.

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, **/a/b[3]/c**). A `RepositoryException` 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 `this.getDepth() == 0`), an empty string will be returned. A `RepositoryException` is thrown if an error occurs.
`Item`	**getAncestor(int depth)** Returns the ancestor of the specified depth below the root. An ancestor of depth **x** is the `Item` that is **x** levels down along the path from the root node to *this* `Item`. • **depth = 0** returns the root node. • **depth = 1** returns the child of the root node along the path to *this* `Item`. • **depth = 2** returns the grandchild of the root node along the path to *this* `Item`. • And so on to **depth = n**, where **n** is the depth of *this* `Item`, which returns *this* `Item` itself. If **depth > n** is specified then an `ItemNotFoundException` is thrown. An `ItemNotFoundException` will be thrown if **depth < 0** or **depth > n** where **n** is the is the depth of this item along the path returned by `getPath()`. An `AccessDeniedException` is thrown if the current session does not have sufficient access permissions to retrieve the specified node.

	A **RepositoryException** is thrown if another error occurs.
Node	**getParent()** Returns the parent of this **Item**. An **ItemNotFoundException** is thrown if there is no parent node. This only happens if this item is the root node of a workspace. An **AccessDeniedException** is thrown if the current session does not have sufficient access permissions to retrieve the parent of this item. A **RepositoryException** is thrown if another error occurs.
int	**getDepth()** Returns the depth below the root node of *this* **Item** (counting *this* **Item** itself): • 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 *this* **Item**. A **RepositoryException** is thrown if an error occurs.
Session	**getSession()** Returns the **Session** through which this **Item** was acquired. A **RepositoryException** is thrown if an error occurs.
boolean	**isNode()** Returns **true** if this **Item** is a **Node**; returns **false** if this **Item** is a **Property**.
boolean	**isSame(Item otherItem)** Returns **true** if this **Item** object (the Java object instance) represents the same actual repository item as the object **otherItem**. Two **Item** objects represent the same repository item if all the following are true: • Both objects were acquired through

	Session objects that were created by the same **Repository** object.
	• Both objects were acquired through **Session** objects bound to the same repository workspace.
	• The objects are either both **Node** objects or both **Property** objects.
	• If they are **Node** objects, they have the same correspondence identifier (see 4.10.2 *Multiple Workspaces and Corresponding Nodes*).
	• If they are **Property** objects they have identical names and their parent nodes have the same correspondence identifiers.
	This method does not compare the *states* of the two items. For example, if two **Item** objects representing the same actual repository item have been retrieved through two different sessions and one has been modified, then this method will still return **true** when comparing these two objects. Note that if two **Item** objects representing the same repository item are retrieved through the *same* session they will always reflect the same state (see 7.1.3 *Reflecting Item State*) so comparing state is not an issue.
	A **RepositoryException** is thrown if an error occurs.
void	**accept(ItemVisitor visitor)**
	Accepts an **ItemVisitor** and calls the appropriate **visit** method according to whether *this* **Item** is a **Node** or a **Property**.
	This method provides support for the *visitor* 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.
	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.
	Throws a **RepositoryException** if an error occurs.

6.2.9 Effect of Access Denial on Read

If a particular repository restricts the read access of a particular user (see 6.9 **Access Control**), then the nodes and properties to which that user does not have read access will simply not appear to exist. For example, the nodes returned on `N.getNodes` will not include subnodes of `N` to which the user in question does not have read access. In other words lack of read access to an item blocks access to both information about the *content* of that item and information about the *existence* of the item.

6.2.10 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 = "Geometrixx is proud..."
  │ ├─myapp:paragraph[1]
  │ │ ├─myapp:text = "Geometrixx 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]
```

Assuming 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 RepositoryException {
  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 = "Geometrixx is proud..."
      myapp:paragraph[1]
        myapp:text = "Geometrixx 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

A compliant content repository 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 namespace system is modelled after *XML Namespaces*.

6.3.1 Namespace Registry

Each 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 **RepositoryException** is thrown if an error occurs.
String[]	**getURIs()** Returns an array holding all currently registered URIs. A **RepositoryException** is thrown if an error occurs.
String	**getURI(String prefix)** Returns the URI to which the given **prefix** is mapped. If a mapping with the specified **prefix** does not exist, a **NamespaceException** is thrown. A **RepositoryException** is thrown if another error occurs.
String	**getPrefix(String uri)** Returns the prefix which is mapped to the given **uri**. If a mapping with the specified **uri** does not exist, a **NamespaceException** is thrown A **RepositoryException** 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.

- **`xml -> http://www.w3.org/XML/1998/namespace`**
  Reserved for reasons of compatibility with XML. This prefix should not be used by clients of the API in the names of normal nodes or properties, since doing so will cause problems on export to XML.

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

6.3.3 Session Namespace Remapping

Any registered namespace (other than the empty namespace or one beginning with "xml") can be temporarily remapped to a new prefix within the scope of a particular **Session**.

javax.jcr. **Session**	
void	**setNamespacePrefix(String newPrefix, String existingUri)** Within the scope of this **Session**, gives the existing registered namespace URI **existingUri** the new prefix, **newPrefix**. The remapping only affects operations done through this **Session**. To clear all remappings, the client must acquire a new **Session**. If **existingUri** is not registered in the **NamespaceRegistry** a **NamespaceException** will be thrown. If **newPrefix** is already locally mapped to **existingUri** (i.e., within this **Session**, by virtue of an earlier **setNamespaceRegistry** call) then this method returns silently and has no effect. If **newPrefix** is already locally mapped to a URI other than **existingUri**, then that URI reverts to its globally mapped prefix (as set in the **NamespaceRegistry**) and **newPrefix** is locally mapped to **existingUri**. If **newPrefix** is already assigned in the global **NamespaceRegistry** to *otheruri* (which differs from **existingUri**) and *otherUri* has not been locally mapped to another prefix which differs from **newPrefix**, then a **NamespaceException** will be thrown. In order to successfully locally map **newPrefix** to **existingUri**, *otherUri* must first be locally mapped to another prefix. A **NamespaceException** will be thrown if an attempt is made to remap an existing namespace URI to a prefix beginning with the characters "**xml**" (in any combination of case). A **NamespaceException** will be thrown if an attempt is made to remap either the empty prefix or the empty namespace (i.e., if either **newPrefix** or **existingUri** are the empty string). A **RepositoryException** is thrown if another error occurs.
String[]	**getNamespacePrefixes()** Returns all prefixes currently set for this **Session**. This includes all those registered in the **NamespaceRegistry** but

	not temporarily over-ridden by a `Session.setNamespacePrefix,` plus those currently set locally by `Session.setNamespacePrefix`. A `RepositoryException` is thrown if an error occurs.
`String`	`getNamespaceURI(String prefix)` Returns the URI to which the given `prefix` is mapped as currently set in this `Session`. A `NamespaceException` is thrown if the specified `prefix` is unknown. A `RepositoryException` is thrown if another error occurs.
`String`	`getNamespacePrefix(String uri)` Returns the prefix to which the given URI is mapped as currently set in this `Session`. A `NamespaceException` is thrown if the specified `uri` is unknown. A `RepositoryException` is thrown if another error occurs.

6.3.3.1 Using Session Namespace Remapping

One use case for session-based namespace remapping occurs in the context of an XPath or SQL query (see 6.6 *Searching Repository Content* and 8.58.5 **Searching Repository Content with SQL**).

Queries often include literal names that have namespace prefixes. When attempting to use a stored 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 query 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`.

6.3.3.3 Conflict between Session Remapping and Namespace Registry

There are two circumstances in which a potential conflict may arise between a session namespace mapping and the repository-wide namespace registry.

The first case occurs when an attempt is made to locally map prefix **P** to URI **U** where **P** is already globally mapped to URI **U'** (not equal to **U**) and there is no local mapping of some prefix **P'** (not equal to **P**) to **U'**. As described above, an exception is thrown in this case. To successfully locally map **P** to **U**, **U'** must first be locally mapped to some **P'**.

The second case occurs when a mapping of **P** to **U** is added to the global namespace registry while **P** is locally mapped to some **U'** (not equal to **U**) in at least one active session **S**. How this conflict is handled is left up to the implementation, since in any case, the mechanism for making changes to the global namespace registry is beyond the scope of this specification.

Note that the names of nodes and properties must be stored internally in such a way that when accessed they will reflect the current namespace mapping. One way of achieving this is to store them internally using fully qualified names and, upon access, dynamically produce the correct prefixed name or path based on the current mapping. Other mechanisms may also be used to achieve the same result.

Similarly, all properties of type `NAME` or `PATH` must also dynamically reflect the current mapping. 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*).

6.3.4 Transactions and Namespaces

In repositories that support transactions, both changes to the namespace registry and `Session` namespace remappings must be non-transactional.

6.4 XML Mappings

Level 1 supports two mappings of the content repository 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 workspace, the resulting system view is determined as follows:

1. The relevant namespace mapping from the repository **NamespaceRegistry** is included as XML namespace declarations in the top-most XML element (though the **xml** namespace is excluded, since its presence would be redundant). Additionally a namespace mapping is included that maps to **http://www.jcp.org/jcr/sv/1.0**, for example **xmlns:sv="http://www.jcp.org/jcr/sv/1.0"**. In what follows it is assumed that the prefix used is **sv**, though any prefix is allowed as long as it is mapped to the URI above.

2. Each content repository node becomes an XML element **<sv:node>**.

3. Each content repository property becomes an XML element **<sv:property>**.

4. The name of each content repository node or property becomes the value of the **sv:name** attribute of the corresponding **<sv:node>** or **<sv:property>** element.

5. 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).

6. The property type of each content repository property is recorded in the **sv:type** attribute of the corresponding **<sv:property>** element, using the standard string forms for property type names as returned by the method **PropertyType.nameFromValue** (i.e., "**String**", "**Binary**", "**Date**", "**Boolean**", "**Double**", "**Long**", "**Name**", "**Path**" and "**Reference**").

7. The value of each non-**BINARY** content repository property is converted to string form (according to 6.2.6 *Property Type*

Conversion). `BINARY` values are Base64 encoded. In both cases the resulting string is included as XML text within an `<sv:value>` element within the `<sv:property>` element. Entity references are used to escape characters which cannot be included as literals within XML text (see 6.4.4 *Escaping of Values*). In addition, if the string form of a value contains characters which cannot appear in XML documents at all (neither as literals nor as character references[4]) then the value is also Base64 encoded, the attribute `xsi:type="xsd:base64Binary"` is added to the `<sv:value>` element, and the namespace mappings for `xsi` and `xsd` are added to the topmost XML element (i.e., `xmlns:xsd="http://www.w3.org/2001/XMLSchema"` and `xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"`). Note that since `BINARY` values are always Base64 encoded, the `xsi:type="xsd:base64Binary"` attribute is considered the default in those cases and therefore is omitted.

8. A multi-value property is converted to an `<sv:property>` element containing multiple `<sv:value>` elements. The order of the `<sv:value>` elements reflects the order of the value array returned by `Property.getValues`.

9. The hierarchy of the content repository nodes and properties is reflected in the hierarchy of the corresponding XML elements.

10. Within an `<sv:node>` element all `<sv:property>` subelements must occur before the first `<sv:node>` subelement.

11. The first two `<sv:property>` elements within an `<sv:node>` element must be the `jcr:primaryType` and `jcr:mixinTypes` (in that order) properties of the node in question.

12. In the case of referenceable nodes, the third `<sv:property>` element in the `<sv:node>` element must be `jcr:uuid`.

13. The order of the `<sv:node>` subelements of a parent `<sv:node>` must reflect the order in which the corresponding child nodes are returned by `Node.getNodes()`.

6.4.1.1 Example

A subtree with the following structure:

---

[4] See `http://www.w3.org/TR/REC-xml/#charsets`, `http://www.w3.org/TR/REC-xml/#NT-CharRef`, and `http://www.w3.org/TR/REC-xml/#wf-Legalchar`.

```
Node
Property = "value"

myapp:document
├─jcr:primaryType = "mynt:document"
├─myapp:title = "JSR 170"
├─myapp:lead = "Content Repository"
└─myapp:body
   ├─jcr:primaryType = "mynt:body"
   └─myapp:paragraph
      ├─jcr:primaryType = "mynt:paragraph"
      ├─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

and

mynt -> http://mycorp.com/mynt

would appear in the system view as:

```xml
<?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"
         xmlns:mynt="http://mycorp.com/mynt"
         sv:name="myapp:document">
  <sv:property sv:name="jcr:primaryType" sv:type="Name">
    <sv:value>mynt:document</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="Name">
      <sv:value>mynt:body</sv:value>
    </sv:property>
    <sv:node sv:name="myapp:paragraph">
      <sv:property sv:name="jcr:primaryType" sv:type="Name">
        <sv:value>mynt:paragraph</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 might 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 it is used (export, import or XPath query) and which optional features are supported by the particular implementation in question.

The next section describes the general structure of the document view mapping and then moves 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 workspace, the general form of the document view is determined as follows:

1. The relevant mappings from the repository namespace registry are rendered as a set of namespace declarations in the top-most XML element (though the `xml` namespace is excluded, since its presence would be redundant).

2. Each content repository node `N` becomes an XML element of the same name, `N`.

3. Each child node `C` of `N` becomes a subelement `C` of XML element `N`.

4. The order of the subelements of element `N` must reflect the order in which the corresponding child nodes are returned by `Node.getNodes`.

5. Each property **P** of node **N** becomes an XML attribute **P** of XML element **N**.

6. If **P** is a non-**BINARY** property its value is converted to string form according to the standard conversion (see 6.2.6 *Property Type Conversion*). If **P** is a **BINARY** property its value is Base64 encoded. The resulting string becomes the value of the XML attribute **P**. Entity references are used to escape characters which should not be included as literals within attribute values (see 6.4.4 *Escaping of Values*).

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 workspace root node, by definition, has the empty string as its name.

6.4.2.3 XML Text

In level 2, on document view import XML text is converted to the special node/property structure **jcr:xmltext/jcr:xmlcharacters** (see 7.3.2 *Import from Document View*). When this structure is mapped back to XML the following rules apply.

If a child node of **N** called **jcr:xmltext** is encountered and that **jcr:xmltext** node has one and only one child item and that item is a single-valued property called **jcr:xmlcharacters**, then the 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**. Entity references are used to escape characters which should not be included as literals within XML text (see 6.4.4 *Escaping of Values*) however, escaping of whitespace is not performed (see 6.4.2.5 *Multi-value Properties*). Note also that two or more **jcr:xmltext** nodes adjacent within the ordering of a child node set will have the values of their respective **jcr:xmlcharacters** concatenated into a single resulting XML text node.

- **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. See 6.6.4.12 *text() Node Test* for more details.

6.4.2.4 Invalid Item Names

If the name of a content repository item `I` is not a valid XML element or attribute name (as the case may be) then 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 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 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 `I`.

6.4.2.5 Multi-value Properties

If a multi-value property `P` 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 type[5] (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*. If the multi-value property contains no values, then it is serialized as an empty string.

  - All literal whitespace within each string is escaped, as well as any characters that should not be included as literals in any case, see 6.4.4 *Escaping of Values*.

  - The final attribute value is constructed by concatenating the resulting strings, with the addition of the space delimiter, into a single string. The order of concatenation must be the same as the order in

---

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

which the values appear in the **Value** array returned by **Property.getValues**.

    o   Furthermore, if multi-value property serialization is supported, then a mechanism must be adopted whereby upon re-import the distinction between multi- and single- value properties is not lost, see 6.4.4 *Escaping of Values*.

    o   Note that this escaping of space literals does not apply to the value of **jcr:xmltext/jcr:xmlcharacters** when it is converted to XML text. In that case only the standard XML entity escaping is required, regardless of whether multi-value property serialization is supported (see 6.4.2.3 *XML Text* and 6.4.4 *Escaping of Values*).

- **XPath**: In the context of an XPath query, the value array of property **P** 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 the five special characters (**&**, **<**, **>** ,**'** and **"**) that would normally be escaped using predefined entity references. This is possible because the XML stream in the XPath context is virtual and therefore it need never be actually serialized. However, 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 any of the five special characters occurring within value literals in the XPath statement are not escaped (see 6.6.4.10 *Searching Multi-value Properties*).

6.4.2.6 Invalid Characters in Values

If the string form of the value of property **P** contains characters which cannot appear in XML documents at all (neither as literals nor as character references[6]) then the attribute **P** is simply excluded from the document view serialization and does not appear at all.

6.4.2.7 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

---

[6] See **http://www.w3.org/TR/REC-xml/#charsets**, **http://www.w3.org/TR/REC-xml/#NT-CharRef**, and **http://www.w3.org/TR/REC-xml/#wf-Legalchar**.

multi-value property serialization is not supported and therefore escaping of space literals is not done (see 6.4.4 *Escaping of 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:myapp="http://mycorp.com/myapp"
                xmlns:mynt="http://mycorp.com/mynt"
                jcr:primaryType="mynt:document">
                myapp:title="JSR 170"
                myapp:lead="Content Repository">
  <myapp:body jcr:primaryType="mynt:body">
    <myapp:paragraph jcr:primaryType="mynt:paragraph"
                     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 might not have any extraneous whitespace between elements or attributes.

6.4.3 Escaping of Names

Not every item name is a valid XML name. In particular, even though a content repository prefix is always a valid XML prefix, the content repository 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 content repository name may contain spaces, whereas XML names cannot.

Consequently, for document view serialization, each content repository name is converted to a valid XML name (as defined by XML 1.0) by translating invalid characters into escaped numeric entity encodings[7].

The escape character is the underscore ("_"). Any invalid character is escaped as *_xHHHH_*, where *HHHH* is the four-digit hexadecimal UTF-16 code for the character. When producing escape sequences the implementation should use lowercase letters for the hex digits `a-f`. When unescaping, however, both upper and lowercase alphabetic hexadecimal characters must be recognized.

Escaping and unescaping is done by parsing the name from left to right.

---

[7] This escaping scheme is based on the scheme described in ISO/IEC 9075-14:2003 for converting arbitrary strings into valid XML element and attribute names.

The underscore character ("_"), when appearing as literal, is itself escaped if it is followed by **xHHHH** where **H** is one of the following characters: **0123456789abcdefABCDEF**.

So, for example,

- "**My Documents**" is encoded as "**My_x0020_Documents**".

- "**My_Documents**" is not encoded.

- "**My_x0020Documents**" is encoded as "**My_x005f_x0020Documents**".

- "**My_x0020_Documents**" is encoded as "**My_x005f_x0020_Documents**".

- "**My_x0020 Documents**" is encoded as "**My_x005f_x0020_x0020_Documents**".

6.4.4 Escaping of Values

When a non-**BINARY** value is serialized during either system view or document view export, it is first converted to string form using standard value conversion, see 6.2.6 *Property Type Conversion* (**BINARY** values are encoded using Base64).

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.

In document view serialization, if the property being serialized is multi-valued (or if the implementation chooses to encode spaces in single value properties too, see below) then the value or values must be further encoded by escaping any occurrence of one of the four whitespace characters: space, tab, carriage return and line feed. The scheme used to encode these characters is the same as that described in 6.4.3 *Escaping of Names*. Note that in this restricted context, applying those escaping rules amounts to the following: a space becomes **_x0020_**, a tab becomes **_x0009_**, a carriage return becomes **_x000D_**, a line feed becomes **_x000A_** and any underscore (_) that occurs as the first character of a sequence that could be misinterpreted as an escape sequence becomes **_x005f_**.

Finally, in document view export, the value of the attribute representing a multi-value property is constructed by concatenating the results of the above escaping into a space-delimited list.

In document view export (though not in system view), if multi-value property serialization is supported (see 6.4.2.5 *Multi-value Properties*) then a mechanism must be adopted whereby upon re-

import the distinction between multi- and single- value properties is not lost. One option is that escaping of space literals must be applied to the value of all single-value properties as well, Another option is that when an XML document is imported in document view, each attribute is assumed to be a single-value property unless out-of-band information defines it to be multi-valued (for example, if the applicable node type defines the property as multi-valued or the XML document is associated with a schema definition that indicates that the attribute is a list value).

Note that the value of a `jcr:xmlcharacters` property used to represent XML text (see 6.4.2.3 *XML Text*) is not space-escaped, regardless of the prevailing multi-value property serialization policy.

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`	`exportSystemView(String absPath,`
	`                  ContentHandler contentHandler,`
	`                  boolean skipBinary,`
	`                  boolean noRecurse)`
	
	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 *system view* form. Note that **absPath** must be the path of a node, not a property.
	
	If **skipBinary** is **true** then any properties of **PropertyType.BINARY** will be serialized as if they are empty. That is, the existence of the property will be serialized, but its content will not appear in the serialized output (the **<sv:value>** element will have no content). Note that in the case of multi-value **BINARY** properties, the number of values in the property will be reflected in the serialized output, though they will all be empty. If **skipBinary** is **false** then the actual value(s) of each **BINARY** property is recorded using Base64 encoding.
	
	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.
	
	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.
	
	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.
	
	A **PathNotFoundException** is thrown if no node exists

	at **absPath**.
	A **SAXException** is thrown if an error occurs while feeding events to the **ContentHandler**.
	A **RepositoryException** is thrown if another error occurs.

void	**exportSystemView(String absPath,**
	                               OutputStream out,
	                               boolean skipBinary,
	                               boolean noRecurse)
	
	 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 *system view* form. Note that **absPath** must be the path of a node, not a property.
	
	 If **skipBinary** is **true** then any properties of **PropertyType.BINARY** will be serialized as if they are empty. That is, the existence of the property will be serialized, but its content will not appear in the serialized output (the **<sv:value>** element will have no content). Note that in the case of multi-value **BINARY** properties, the number of values in the property will be reflected in the serialized output, though they will all be empty. If **skipBinary** is **false** then the actual value(s) of each **BINARY** property is recorded using Base64 encoding.
	
	 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.
	
	 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.
	
	 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.
	
	 The output XML will be encoded in UTF-8.
	
	 An **IOException** is thrown if an I/O error occurs.
	
	 A **PathNotFoundException** is thrown if no node exists

	at **absPath**. A **RepositoryException** is thrown if another error occurs.

void	**exportDocumentView(String absPath,**
	                    ContentHandler contentHandler,
	                    boolean skipBinary,
	                    boolean noRecurse)
	
	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 *document view* form. Note that **absPath** must be the path of a node, not a property.
	
	If **skipBinary** is **true** then any properties of **PropertyType.BINARY** will be serialized as if they are empty. That is, the existence of the property will be serialized, but its content will not appear in the serialized output (the value of the attribute will be empty). If **skipBinary** is **false** then the actual value(s) of each **BINARY** property is recorded using Base64 encoding.
	
	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.
	
	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.
	
	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.
	
	A **PathNotFoundException** is thrown if no node exists at **absPath**.
	
	A **SAXException** is thrown if an error occurs while feeding events to the **ContentHandler**.
	
	A **RepositoryException** is thrown if another error occurs.

void	`exportDocumentView(String absPath,`
	`OutputStream out,`
	`boolean skipBinary,`
	`boolean noRecurse)`
	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 *document view* form. Note that `absPath` must be the path of a node, not a property.
	If `skipBinary` is `true` then any properties of `PropertyType.BINARY` will be serialized as if they are empty. That is, the existence of the property will be serialized, but its content will not appear in the serialized output (the value of the attribute will be empty). If `skipBinary` is `false` then the actual value(s) of each `BINARY` property is recorded using Base64 encoding.
	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.
	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.
	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.
	The output XML will be encoded in UTF-8.
	An `IOException` is thrown if an I/O error occurs.
	A `PathNotFoundException` is thrown if no node exists at `absPath`.
	A `RepositoryException` is thrown if another error occurs.

6.5.1 Encoding

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

6.6 Searching Repository Content

In level 1, support for the XPath syntax is required. Optionally, a repository may support the SQL syntax (see 8.5 *Searching Repository Content with SQL*). Implementations may also support additional languages.

XPath is a search language originally designed for selecting elements from an XML document. Since a workspace, like an XML document, can be viewed as a tree structure, XPath provides a convenient syntax for searching workspace content. The main prerequisite for providing XPath querying is to establish an XML mapping of the 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*).

6.6.1 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:myapp="http://mycorp.com/myapp"
          xmlns:mynt="http://mycorp.com/mynt"
          jcr:primaryType="nt:unstructured">
  <myapp:document jcr:primaryType="mynt:document">
                myapp:title="JSR 170"
                myapp:lead="Content Repository">
    <myapp:body jcr:primaryType="mynt:body">
      <myapp:paragraph jcr:primaryType="mynt:paragraph"
                    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 might 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:

//element(*, mynt:paragraph)[@myapp:title="Node Types"]

This query will return the node at the repository workspace path (as opposed to an XPath):

```
/myapp:document/myapp:body/myapp:paragraph
```

6.6.2 XPath and SQL

XPath 2.0 forms the basis of the querying syntax in level 1. All compliant repository implementations must support this search syntax. However, implementations that use a relational database as an underlying datastore will typically be limited in the range of XPath queries that they can efficiently support. Such implementations will find support for the optional SQL syntax a more natural fit (see 8.5 *Searching Repository Content with SQL*).

Therefore, in order to ensure that database-backed implementations are not unnecessarily burdened by compliance requirements, only a subset of XPath is required. This subset is defined as the set of XPath statements that can be translated to and from SQL at parse-time of the query.

This arrangement allows database-backed repositories to implement search natively with SQL but still comply with the minimal XPath requirement by translating XPath queries to SQL.

On the other hand, implementations that are natively hierarchical and therefore capable of supporting XPath functionality beyond the minimum requirement are free to do so.

As mentioned, support for SQL is optional. But because the minimal set of XPath features is driven by the semantic range of SQL, a knowledge of the mapping between the two aids greatly in understanding that minimal feature set XPath. The following sections summarize that mapping.

6.6.3 Structure of a Query

A query, whether XPath or SQL, specifies a subset of nodes within a workspace, called the *result set*. The result set constitutes all the nodes in the workspace that meet the constraints stated in the query. The constraints fall into three categories:

- *Type constraint*: This limits the returned nodes to a particular primary node type (and possibly, additionally limits the nodes to those with particular mixin node types).

- *Property constraint*: This limits the returned nodes to those with particular properties having particular values.

- *Path constraint*: This limits the returned nodes to those within certain subtrees in the workspace.

A query result is returned in two parallel forms: an iterator over the result set of nodes and a table where each row corresponds to a node in the result set. The query statement also defines aspects of

how these two return objects are structured through two presentation specifiers:

- *Column specifier*: This specifies the set of properties that will form the columns of the returned table.

- *Ordering specifier*: This defines the order of the nodes in the iterator and rows in the table.

The following sections describe in more detail how each of these five elements are expressed, both in XPath and in SQL, and how these affect the content and presentation of the query result.

6.6.3.1 Column Specifier

The column specifier of a query is the part of the statement that specifies which properties are to be returned as columns in the result table. Support is only required for single-value, non-residual properties that are declared in or inherited by the node types specified in the *type constraint*. It is optional to allow specification of residual properties as columns.

If no column specifier is given, then at least the default set of columns will be returned. The default set is defined as all single-value, non-residual properties declared in or inherited by the node types specified in the type constraint. It is optional to return columns for residual properties.

In both cases (an explicitly specified set of columns, or the default set) the pseudo-property `jcr:path` will always be returned as a column. `jcr:path` is a special column that does not correspond to any actual property, it holds the normalized absolute path for the node represented by each row (see 8.5.2.2 **Pseudo-property jcr:path**).

As well, a score column will also be included, though it is not required that its contents always be meaningful. Note also that this column may be labeled simply `jcr:score` or it may be labeled with the signature and parameters of the `jcr:score(...)` function used in XPath. Additional score-related columns may be also returned by implementations that support multiple `jcr:score(...)` functions with varying parameters (see 8.5.2.4 **Pseudo-property**, 6.6.5.2 *jcr:contains Function*, and 8.5.4.5 **CONTAINS**).

If columns are explicitly specified then the order in which they are specified in the query is the order in which they will appear in the table. If no columns are explicitly specified then the order in which they appear is implementation-specific.

The *column specifier* has no effect on the content or form of the `NodeIterator` view of the query result.

XPath: In XPath the mechanism of the column specifier is not specified, though one possible approach is to interpret as the column specifier the last location step when it uses the attribute axis; in other words, when content repository properties (XML attributes in document view) are selected in the last location step. If this approach is taken then, for example, multiple properties are selected with a union.

Another possible approach is to define an XPath function that specifies the desired columns. This specification, however, does not attempt to define or limit the possible options.

SQL: In SQL the column specifier is the `SELECT` clause. To select the default column set the `*` is used.

Examples:

SQL	**XPath** (one suggested approach)
SELECT * FROM nt:base	//*
SELECT * FROM my:type	//element(*, my:type)
SELECT my:title FROM my:type	//element(*, my:type)/@my:title
SELECT my:title, my:text FROM my:type	//element(*, my:type)/ (@my:title | @my:text)

6.6.3.2 Type Constraint

A type constraint specifies the common primary node type of the returned nodes, plus, possibly, additional mixin types that they also must have. Type constraints are inheritance-sensitive in that specifying a constraint of node type **x** will include all nodes explicitly declared to be type **x**, as well as all nodes of subtypes of **X**.

Implementations are required to support constraints of one primary type. It is optional to support constraints based on multiple primary node types (this would, in any case, only be applicable to implementations that supported multiple inheritance of node types). It is also optional to support constraints on (one or more) mixin node types.

Note however, that property constraints can always be used to test *declared* types (that is, a non-inheritance-sensitive test), by testing the values of the properties `jcr:primaryType` and `jcr:mixinType`.(see 6.6.3.3 *Property Constraint*).

XPath: In XPath the `element` test is used to test against node type. It is optional to support `element` tests on location steps other than the last.

SQL: In SQL the type constraint is expressed in the `FROM` clause.

Examples:

SQL	XPath
SELECT * FROM my:type	//element(*, my:type)
SELECT * FROM my:type WHERE jcr:path LIKE '/nodes[%]/%'	/jcr:root/nodes// element(*, my:type)

6.6.3.3 Property Constraint

A query may specify further constraints on the result nodes by way of property constraints.

XPath: In XPath a predicate that tests attributes on the last location step forms the property constraint expression. Predicates on any other location step are optional.

SQL: In SQL the `WHERE` clause forms the constraint expression.

Examples:

SQL	XPath
SELECT * FROM my:type WHERE my:title='JSR 170'	//element(*, my:type) [@my:title = 'JSR 170']

In order to ensure mutual translatability between XPath and SQL we only require support for the XPath general comparison operators (**=**, **!=**, **<**, **<=**, **>**, **>=**). In SQL the semantics of these operators must be the same as they are for XPath. The only difference is that in XPath the not-equal operator is **!=**, while in SQL it is **<>**.

The term "general comparison" comes from XPath terminology. The significance of requiring support for XPath general comparison, and their equivalents in SQL, lies in the way that these operators behave with multi-value properties. See 6.6.4.10 *Searching Multi-value Properties* for details.

Additionally, support for `jcr:like()` (`LIKE` in SQL) and `jcr:contains` (`CONTAINS` in SQL) is required (though the range of this requirement is qualified below).

Since not all property types can be meaningfully compared using all operators the following describes the minimal set of comparison support required for each property type:

STRING: **=**, **!= (<>)**, **<**, **<=**, **>**, **>=**, **jcr:like()** (**LIKE**)

LONG: **=**, **!= (<>)**, **<**, **<=**, **>**, **>=**

DOUBLE: **=**, **!= (<>)**, **<**, **<=**, **>**, **>=**

DATE: **=**, **!= (<>)**, **<**, **<=**, **>**, **>=**

NAME: **=**, **!= (<>)**

PATH: **=**, **!= (<>)**, (additionally, in SQL, **LIKE** is used against the **jcr:path** pseudo-property to define path constraints, see 6.6.3.4 *Path Constraint*)

REFERENCE: **=**, **!= (<>)**

BOOLEAN: **=**, **!= (<>)**

BINARY: *(none)*

The **jcr:like** function in XPath corresponds to the **LIKE** operator in SQL. See 6.6.5.1 *jcr:like Function* and 8.5.4.4 *LIKE.*

Support for the **jcr:contains()** (**CONTAINS()** in SQL) function is not required for any property types in particular. It is however required to work *at the node level*. In that case it applies to those properties of the node for which the implementation maintains an index. Which properties those are is an implementation issue. See 6.6.5.2 *jcr:contains Function* and **8.5.4.5 CONTAINS**.

Support for comparing **jcr:score** in a SQL **WHERE** clause or **jcr:score(...)** in a XPath predicate is not required.

In 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. Support for **[@p = @q]** (and so forth for each operator) is not required.

Examples:

SQL	XPath
my:title = 'JSR 170'	@my:title = 'JSR 170'
my:title <> 'JSR 170'	@my:title != 'JSR 170'
my:title < 'JSR 170'	@my:title < 'JSR 170'
my:title <= 'JSR 170'	@my:title <= 'JSR 170'

`my:title > 'JSR 170'`	`@my:title > 'JSR 170'`
`my:title >= 'JSR 170'`	`@my:title >= 'JSR 170'`
`my:title = 'JSR 170' AND my:author = 'David'`	`@my:title = 'JSR 170' and @my:author = 'David'`
`my:title = 'JSR 170' OR my:title = 'JSR-170'`	`@my:title = 'JSR 170' or @my:title = 'JSR-170'`
`NOT (my:title = 'JSR 170')`	`not(@my:title >= 'JSR 170')`
`my:title IS NOT NULL`	`@my:title`
`my:title IS NULL`	`not(@my:title)`
`my:title LIKE 'JSR 170%'`	`jcr:like(@my:title, 'JSR 170%')`
`CONTAINS(*, 'JSR 170')`	`jcr:contains(., 'JSR 170')`

6.6.3.4 Path Constraint

The path constraint restricts the result node to a scope specified by a path expression. The following path constraints must be supported:

- Exact
- Child nodes
- Descendants
- Descendants or self

XPath: In XPath the location steps specify the path constraint.

SQL: In SQL the path constraint occurs as an ANDed test within the WHERE clause of the pseudo-property `jcr:path` using either the = operator or the LIKE operator.

Exact path constraint examples:

SQL	XPath
`SELECT * FROM my:type WHERE jcr:path LIKE '/some[%]/nodes[%]'`	`/jcr:root/some/ element(nodes, my:type)`
`SELECT * FROM my:type WHERE jcr:path = '/some/nodes'`	`/jcr:root/some[1]/element(nodes, my:type)[1]`

Child nodes path constraint examples:

SQL	XPath
SELECT * FROM my:type WHERE jcr:path LIKE '/some[%]/nodes[%]/%' AND NOT jcr:path LIKE '/some[%]/nodes[%]/%/%'	/jcr:root/some/nodes/ element(*, my:type)
SELECT * FROM my:type WHERE jcr:path LIKE '/some/nodes/%' AND NOT jcr:path LIKE '/some/nodes/%/%'	/jcr:root/some[1]/nodes[1]/ element(*, my:type)

Descendants path constraint examples:

SQL	XPath
SELECT * FROM my:type WHERE jcr:path LIKE '/some[%]/nodes[%]/%'	/jcr:root/some/nodes// element(*, my:type)
SELECT * FROM my:type WHERE jcr:path LIKE '/some/nodes/%'	/jcr:root/some[1]/nodes[1]// element(*, my:type)

Descendants or self path constraint examples:

SQL	XPath
SELECT * FROM my:type WHERE jcr:path LIKE '/some[%]/nodes[%]' OR jcr:path LIKE '/some[%]/nodes[%]/%'	/jcr:root/some/nodes// element(*, my:type)
SELECT * FROM my:type WHERE jcr:path = '/some/nodes' OR jcr:path LIKE '/some/nodes/%'	/jcr:root/some[1]/nodes[1]// element(*, my:type)

6.6.3.5 Ordering Specifier

This part of the query statement specifies the ordering of the result nodes according to the natural ordering of the values of one or more properties of the result nodes. If no order specification is supplied in the query statement, implementations may support document order on the result nodes (see 6.6.4.2 *Document Order*).

In both XPath and SQL, ordering is specified by a special clause that lists one or more property names and, for each, whether the order is to be ascending or descending. If neither ascending nor descending is specified after a property name (or `jcr:score(...)` function), the default is ascending.

Implementations must support ordering on `jcr:score` (in SQL just the name is used, see 8.5.2.4 **Pseudo-property**) or `jcr:score(...)` (in XPath, the function form is used, though the number of parameters depends on the implementation, see 6.6.5.3 *jcr:score function*).

Support for ordering on `PATH` and `NAME` properties is not required. If it is supported then the collation sequence for these types is implementation specific.

XPath: A subset of the order by clause specified in XQuery 1.0 is used to implement the order specification in XPath. If the order modifier (`ascending` or `descending`) is missing, the ordering defaults to `ascending`. See 6.6.5.5 *order by Clause*.

SQL: The order by clause implements the order specification in SQL. If the order modifier (`ASC` meaning ascending or `DESC`, meaning descending) is missing, the ordering defaults to `ASC`. See 8.5.4.6 **ORDER BY**.

Examples:

SQL	XPath
SELECT * FROM my:type ORDER BY my:title	//element(*, my:type) order by @my:title
SELECT * FROM my:type ORDER BY my:date DESC, my:title ASC	//element(*, my:type) order by @my:date descending, @my:title ascending
SELECT * FROM my:type WHERE CONTAINS(., 'jcr')	//element(*, my:type) [jcr:contains(., 'jcr')] order by jcr:score() descending

ORDER BY jcr:score DESC	

6.6.4 Adapting XPath to the Content Repository

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 value ("**true**" or "**false**") indicates whether the feature in question is supported (see 6.1.1.1 *Repository Descriptors*).

6.6.4.1 Same-Name Siblings

The syntax used to address same-name sibling nodes in a workspace 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. **Repository.getDescriptor(QUERY_XPATH_POS_INDEX)** returning "**true**" indicates that the index position notation for same-name siblings is supported for XPath query.

6.6.4.2 Document Order

The results returned by an XPath query *may* reflect the document order of the elements returned if no **order by** clause is specified (see 6.6.3.5 *Ordering Specifier*, above).

Support for this is optional, since some implementations (notably, database-backed ones) will not have a notion of document order.

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

Repository.getDescriptor(QUERY_XPATH_DOC_ORDER) returning "**true**" indicates that document order searching is supported.

6.6.4.3 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.

6.6.4.4 Mapping Property Types to XML Types

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

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

6.6.4.5 Abbreviated Syntax

Only support for the abbreviated syntax of XPath is required.

6.6.4.6 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.

See 6.6.3.3 *Property Constraint* and 6.6.3.4 *Path Constraint*, above.

6.6.4.7 Predicates

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

Find all employees who have a secretary and an assistant property:
```
//element(*, employee)[@secretary and @assistant]
```

See 6.6.3.3 *Property Constraint*, above.

6.6.4.8 Boolean Functions

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

6.6.4.9 Escaping

The names of elements and attributes (corresponding to nodes and properties, respectively) within an XPath statement must correspond to the form in which they (notionally) appear in the document view. This means that spaces (and any other non-XML characters) within names must be encoded according to the rules described in 6.4.3 *Escaping of Names*.

The values of attributes, on the other hand, are not escaped in the XPath usage of document view (as opposed to the export usage, see 6.4.2.5 **Multi-value Properties**, 6.4.4 **Escaping of Values** and *6.6.4.10 Searching Multi-value Properties*). As a result, string literals in XPath predicates that test those attribute values are not escaped either. They may contain whitespace as well as the characters ampersand (`&`), less-than (`<`), greater-than (`>`), quotation mark (") and apostrophe ('). In XPath the entity references that would be used within an XML document (`<`, `>` etc..) are not used. However, the apostrophe (') and quotation mark(`"`) must be escaped according to the standard rules of XPath with regard to string literals: If the literal is delimited by apostrophes, two adjacent apostrophes within the literal are interpreted as a single apostrophe. Similarly, if the literal is delimited by quotation marks, two adjacent quotation marks within the literal are interpreted as one quotation mark.[8]

6.6.4.10 Searching Multi-value Properties

Searching of multi-value properties with XPath is supported by mapping multi-value properties to XML attributes of the *list type* and employing the XPath feature of *general comparisons*.

---

[8] See http://www.w3.org/TR/2005/WD-xpath20-20050404/#doc-xpath-StringLiteral for details.

XML Schema supports attributes having types. Among those is one called the list type[9]. This type specifies that white space within an attribute value serves as the delimiter between individual list items.

Additionally, XPath 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, will return **true** if the specified relation (equal, not equal, greater-than, etc.) evaluates to **true** for *at least one* of the values of 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']
```

would return all nodes with path **/x/y** that have a property **p** which has *at least one* value **"hello"**.

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

6.6.4.11 Comparison Operators

In XPath only support for the general comparison operators (**=**, **!=**, **<**, **<=**, **>**, **>=**) is required. See 6.6.3.3 *Property Constraint*, above.

---

[9] See http://www.w3.org/TR/xmlschema-0/#ListDt for more information about the XML Schema list type.

6.6.4.12 *text()* Node Test

As discussed in 7.3.2 *Import from Document View* and 6.4.2.3 *XML Text*, document view import converts XML text nodes into the special structure `jcr:xmltext/jcr:xmlcharacters`, importing the actual text into the value of the `jcr:xmlcharacters` property.

In the virtual XML document against which XPath is run, this structure appears as an element and attribute. For example:

```
<limerick>
<jcr:xmltext jcr:xmlcharacters="There once was a..."/>
</limerick>
```

However, as a convenience, the XPath `text()` node test *may* be supported in such a way as to make the text also simultaneously visible to XPath in its original form, as an XML text node:

```
<limerick>There once was a...</limerick>
```

If the `text()` node test is supported, the result is simply that `text()` becomes equivalent to `jcr:xmltext` as a node test within an XPath statement. For example, the XPaths `/jcr:root/limerick/jcr:xmltext` and `/jcr:root limerick/text()` would be equivalent.

6.6.4.13 *element()* Node Test

The `element()` node test defined in XPath 2.0 is used to select nodes of a particular primary node type. For example:

```
//element(*, nt:file)
```

would select all nodes of primary node type `nt:file`. This includes all node of subtypes of `nt:file` as well.

6.6.5 XPath Extensions

XPath in content repositories also defines a small number of functions and one syntactic addition (the order-by clause). Content repository-related functions are prefixed with `jcr:`.

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 repository node. Similarly, the type *attribute()* refers to an XML attribute, which corresponds to a repository property and *node()* refers to an XML node which corresponds to a repository item (that is, a repository node or property).

6.6.5.1 *jcr:like* Function

This function is based on the **LIKE** predicate found in SQL. Support for this function is required, as described in 6.6.3.3 *Property Constraint*.

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

For example, the query "Find all **para**s in **document** whose title property includes the substring 'Java' ", is expressed as:

```
/jcr:root/document/para[jcr: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 ("****").

6.6.5.2 *jcr:contains* Function

```
jcr:contains($scope as node(),
             $exp 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 8.5.4.5 *CONTAINS.*

The first parameter defines the scope of the contains predicate. It can be either "**.**" (meaning this node, i.e., the node-set defined by the current location step) or it can be an XML attribute name (and therefore a content repository property), for example **@my:property**, specifying a particular property of the node-set defined by the current location step). If the scope is "**.**" then all properties of the current node set for which the implementation maintains an index are searched. If a specific property is specified then only the value of that property is searched (if the property is not indexed then the function will return false).

As described in 6.6.3.3 *Property Constraint*, support for the **jcr:contains()** function is required to work *at the node level* in those repositories which support full text searching. In other words only support for "**.**" is required. Support for property specific full-text search is optional.

The EBNF for the second parameter is:

```
searchexp ::= [-]term {whitespace [OR]
              whitespace [-]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 *exp* 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 double quotes (**"**).

- The entire text search expression (**searchexp** in the EBNF, above) *must be* delimited by single 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 ("****").

For example, the query "Find all nodes with some property that contains the text 'JSR 170' " is expressed as:

//*[jcr:contains(., 'JSR 170')]

the optionally supported query "Find all nodes with a property called **myapp:title** that contain the text 'JSR 170' " is expressed as:

//*[jcr:contains(@myapp:title, 'JSR 170')]

The relevance score for each node may be returned in (one or more) score columns (**jcr:score** or **jcr:score(...)**) however the details of how the score is calculated are implementation-specific (see 8.5.2.4 **Pseudo-property**, 6.6.5.2 *jcr:contains Function* and 8.5.4.5 **CONTAINS**).

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

6.6.5.3 *jcr:score* function

jcr:score(...) *as xs:decimal*

As described in 6.6.3.1 *Column Specifier*, a score value is returned for each row the result table. However, how this value is calculated is left up to the implementation. It is not required that its contents always be meaningful.

The XPath function `jcr:score(...)` is provided to enable queries to specify score calculation parameters in those implementations that support it.

The `jcr:score(...)` function must therefore be supported, but the number and meaning of its parameters is left up to the implementation. The `jcr:score(...)` function can be used in either the columns specifier of the query, or the order specifier. It is also possible (though not required) that implementations support multiple `jcr:score(...)` functions within a single query.

The column within which the score information is returned may be labeled simply `jcr:score` or it may be labeled with the signature and parameters of the `jcr:score(...)` function used. Additional score-related columns may be also returned by implementations that support multiple `jcr:score(...)` functions with varying parameters (see 8.5.2.4 **Pseudo-property**, 6.6.5.2 *jcr:contains Function*, and 8.5.4.5 **CONTAINS**).

Support for comparing `jcr:score(...)` in a predicate is not required.

6.6.5.4 *jcr:deref* Function

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

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

The first argument is an XML attribute that represents a **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:

```
/jcr:root/myapp:myDoc/
    jcr:deref(@myapp:author, 'myapp:person')/address
```

The dereference expression above evaluates to one or more nodes which has the name **myapp:person**. Subsequently a child node of each, representing the person's address, is selected.

6.6.5.5 *order by* Clause

As described in 6.6.3.5 *Ordering Specifier*, 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 is done by adding the order by clause to a location path expression. The additions to the syntax of the XPath location clause are as follows:

```
JCRXPathExpr      ::=   (XPath OrderByClause?)?

OrderByClause     ::=   "order by" OrderSpecList

OrderSpecList     ::=   OrderSpec ("," OrderSpec)*

OrderSpec         ::=   ("@" AttributeName OrderModifier) |
                            (ScoreFunction OrderModifer)

OrderModifier     ::=   ("ascending" | "descending")?

ScoreFunction     ::=   "jcr:score(" ParamList ")"

ParamList         ::=   /* zero or more comma separated
                        parameters */
```

For example, the query to find all nodes of type **car** where the **color** is "**green**" and sort them by price in ascending order, the following query might be used:

//element(*, car)[@brand='green'] order by @price ascending

Note that if neither **ascending** nor **descending** are explicitly specified the default behavior is **ascending**.

6.6.6 XPath Grammar

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

6.6.6.1 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]

6.6.6.2 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)*

    /* Note that support for a UnionExpr
    of attributes in the last location
    step is optional*/

[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~~        ~~::=~~  ~~(ItemType OccurrenceIndicator?)~~
                                 ~~| <"empty" "(" ")">~~

[63]  ~~AtomicType~~          ~~::=~~  ~~QName~~

[64]  ~~ItemType~~            ~~::=~~  ~~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)~~
                                 ~~| (AttribNameOrWildcard (","~~
                                 ~~TypeNameOrWildcard)?))? ")"~~

[68]  ElementName           ::=  QName

[69]  AttributeName         ::=  QName

[70]  TypeName              ::=  QName

[71]  ElementNameOrWildca   ::=  ElementName | "*"
      rd

[72]  ~~AttribNameOrWildcar~~   ~~::=~~  ~~AttributeName | "*"~~
```

~~d~~

[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~~   ~~::=   "?" | "*" | "+"~~

6.6.6.3 Notes on the Grammar

Required function support ([42] and [60] above) is limited to the functions described in 6.6.5 *XPath Extensions.*

6.6.7 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.

6.6.8 Query API

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

javax.jcr. **Workspace**	
QueryManager	**getQueryManager()** Returns the **QueryManager**, through which search methods are accessed. A **RepositoryException** is thrown if an error occurs.

6.6.9 QueryManager

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

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

6.6.10 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 strings returned by **QueryManager.getSupportedQueryLanguages()**.
String	**getStoredQueryPath()** If this is a **Query** that has been stored using **Query.storeAsNode** (regardless of whether it has been **save**d yet) or retrieved using **QueryManager.getQuery**, then this method returns the path of the **nt:query** node that stores the query. If this is a transient query (that is, a **Query** object created with **QueryManager.createQuery** and not yet stored), then this method throws an **ItemNotFoundException**. Throws a **RepositoryException** if another error occurs.
void	**storeAsNode(String absPath)** Creates a node representing this **Query** in content. In a level 1 repository this method throws an **UnsupportedRepositoryOperationException**. In a level 2 repository it creates a node of type **nt:query** at **absPath** and returns that node. In order to persist the newly created node, a **save** must be performed that includes *the parent* of this new node within its scope. In other words, either a **Session.save** or an **Item.save** on the parent or higher-degree ancestor of **absPath** must be performed. In the context of this method the **absPath** provided must not have an index on its final element. If it does then a **RepositoryException** is thrown. Strictly speaking, the parameter is actually a absolute path to the parent node of the node to be added, appended with the name desired for the new node. 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.
	An **ItemExistsException** will be thrown either immediately (by this method), or on **save**, if an item at the specified path already exists and same-name siblings are not allowed. Implementations may differ on when this validation is performed.
	A **PathNotFoundException** will be thrown either immediately (by this method), or on **save**, if the specified path implies intermediary nodes that do not exist. Implementations may differ on when this validation is performed.
	A **ConstraintViolationException** will be thrown either immediately (by this method), or on **save**, if adding the node would violate a node type or implementation-specific constraint or if an attempt is made to add a node as the child of a property. Implementations may differ on when this validation is performed.
	A **VersionException** will be thrown either immediately (by this method), or on **save**, 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. Implementations may differ on when this validation is performed.
	A **LockException** will be thrown either immediately (by this method), or on **save**, if a lock prevents the addition of the node. Implementations may differ on when this validation is performed.
	A **RepositoryException** is thrown if another error occurs.
`String`	**XPATH** A string constant representing the XPath query language applied to the *document* view XML mapping of the workspace.
`String`	**SQL** A string constant representing the SQL query language applied to the *database view* of the workspace. Support for this language is optional. See 8.5 *Searching Repository Content with SQL*.

6.6.11 Persistent vs. Transient Queries

When a new `Query` object is first created with
`QueryManager.createQuery` it is a *transient query*. If the
repository is level 2 compliant and supports the node type
`nt:query`, then a transient query can be stored in content by
calling `Query.storeAsNode(String absPath)`. This creates an
`nt:query` node at the specified path (a `save` on the parent of the
new node is required to persist the stored query). Retrieving a
stored query is done by passing the `nt:query` node to
`QueryManager.getQuery(Node node)`.

Note that the actual query statement stored within a persistent
query (that is, the value of the property `jcr:statement`, for
example, "`//*[@jcr:primaryType='nt:file']`" or "`SELECT *
FROM nt:base WHERE jcr:primaryType='nt:file'`") is
namespace-fragile in that it is stored as a literal string with the
namespaces in prefix form. As a result, if the stored query is run in
a context where a prefix it references has been remapped, the
query will not produce the same result as it would have before the
remapping. It is left up to the application to ensure that appropriate
mappings are in place (either using temporary `Session` remapping
or persistent `NamespaceRegistry` changes) when a stored query is
executed.

6.6.12 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 can clear the `Session` (either through `save` or
`refresh(false)`) before running a query in order to avoid such
discrepancies.

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. See 6.6.3 *Structure of a Query* for details of

how the various aspects of these two views are governed by the query. The methods below provide access to the two views:

javax.jcr.query. **QueryResult**	
`String[]`	**getColumnNames()** Returns an array of all the column names in the table view of this result set. Throws a **RepositoryException** if an error occurs.
`RowIterator`	**getRows()** Returns an iterator over the **Row**s of the result table. If an **ORDER BY** 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 **RepositoryException** if an error occurs.
`NodeIterator`	**getNodes()** Returns an iterator over all nodes that match the query. If an **ORDER BY** 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 **RepositoryException** if an error occurs.

javax.jcr.query. **Row**	
`Value[]`	**getValues()** Returns an array of all the values in the same order as the column names returned by **QueryResult.getColumnNames()**. Throws a **RepositoryException** if an error occurs.
`Value`	**getValue(String propertyName)** Returns the value of the indicated property in this **Row**. If **propertyName** is not among the column names of the query result table, an **ItemNotFoundException** is thrown. Throws a **RepositoryException** if another error

	occurs.

6.6.13 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 content 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 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 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**. Predefined mixin types are prefixed with **mix**. See 6.7.19 *Predefined Node Types.*

- **Supertypes**: A primary node type (with the exception of **nt:base**) *must* extend another node type (or more than one node type, if the implementation supports multiple inheritance). A mixin node type *may* extend another node type. 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 through this API.

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 content repository, every node has one and only one declared primary node type. This node type defines, as mentioned, a set of restrictions on the child items of the node. Note that because node types may inherit characteristics from supertypes, a particular node may *be* of more than one primary node type by virtue of type inheritance. For example if **X** is a supertype of **Y** and node **N** is of type **Y**, then **N** is also of type **X**. Nonetheless, this does not change the fact that any particular node still has exactly one declared node type.

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`.

Note that the mixin node types listed in the `jcr:mixinTypes` property are those that have been *explicitly assigned* (using `Node.addMixin`) to a node. It does not include mixin types that may be among the supertypes of a node's primary type.

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 application using 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 defined by this specification or implementation or 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 (though it may be specified as `UNDEFINED`).

- 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. Only properties with a default value can be auto-created.

- Whether the property is **mandatory**. A mandatory property is one that must exist. If a node of a type that specifies a mandatory property is created then any attempt to `save` that node without adding the mandatory property will fail. Since single-value properties either have a value or do not exist (there being no concept of the null value) this implies that a mandatory single-value property must have a value. A mandatory multi-value property on the other hand may have zero or more values.

- The **onParentVersion** status of the property. This specifies what happens to 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) directly through this API but which may be modified or removed by the repository implementation itself.

- 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 is capable of listing more than one node type to accommodate those implementations that support multiple inheritance of primary node types. A child node definition may, for example, specify required node types **X** and **Y**. *This does not mean that the node specified will have more than one declared primary node type*, but rather that its primary node type (whatever else it may be) must be *at least* a subtype of both node types **X** and **Y**. Of course, inheritance is also respected in the simpler case where this attribute specifies only one primary node type. If it specifies, for example, type **T**, this means that the child node must be of type **T** *or a subtype of type **T**.* Finally, it should be clear that the subtype relationship between the required type (or types) and the actual type of the child node must be an explicit one, that is, it must arise be by virtue of a chain of declared superclass attributes (see 6.7.8 *Inheritance Among 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.

- 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.

- 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 possibly other attributes) and may declare further property or child node definitions.

The semantics of inheritance follow the usual rules:

- The **supertype** relation is, as one would expect, transitive. In other words if **T1** is a supertype of **T2** and **T2** is a supertype of **T3** then **T1** is a supertype of **T3**.

- The **subtype** relation is, of course, the converse of supertype: **T1** is a subtype of **T2** if and only if **T2** is a supertype of **T1**. Hence, subtype is also a transitive relation.

- The '**is of type**' relation which holds between node instances and node types (as in, node **N** is of type **T**) is itself transitive across the **subtype** relation. In other words, if **T2** is a subtype of **T1** and **N** is of type **T2** then **N** is also of type **T1**. This predicate appears in the API as the method `Node.isNodeType`. Note that this relation is also the one that is relevant in the child node definition attribute *required primary node types*.

- The supertype relation always and only stems from explicit *Supertypes* attribute declarations within the set of node types. For example, just because the property and child node definitions of **T2** happen to be a superset of those of **T1** does not make **T1** a supertype of **T2**. For that to be the case, **T2** must declare **T1** as its supertype.

- Similarly, the '**is of type**' relation always and only stems from an explicit assignment of a node type to a node. Just because node **N** happens to have the properties and child nodes declared by node type **T** does not necessarily mean that **N** is of type **T**. For that to be the case, **N** must have been explicitly assigned the type **T**, or a subtype of **T**.

Management of the hierarchy of node types available within a particular repository is outside the scope of this specification. However, the requirement of preserving the '**is of type**' relation across subtyping, as mentioned above, does imply certain things about inheritance. The requirement can be restated as:

- If **T2** is a subtype of **T1**, then any instance of **T2** must also be a valid instance of **T1**.

Note that an implementation can meet this requirement in a number of ways, ranging from the most coarse-grained to the most fine-grained. A coarse-grained approach would be to say that a subtype can never override the property of child node definition of a supertype (that is, declare a definition with the same name as one in the supertype). A more fine-grained approach would allow such overrides, but only in cases where an instance of the subtype would still be a valid instance of the supertype. For example, if a

supertype declares a property definition called **A** of type `UNDEFINED`, a subtype would may override that with a property definition **A** of type `STRING`. However, the reverse would not be allowed.

For purposes of the above, the notion of two definitions having the same name *does not apply* to two residual definitions. Two (or more) residual property or child node definitions with differing sub-attributes must be permitted to co-exist in the same effective node type. They are interpreted as disjunctive (ORed) options.

Apart from the issue of how inheritance affects the set of property and child node definitions, there is also the issue of the top-level node type attributes **mixin-status**, **orderable-status** and **primary item**. The specification implies only one requirement with regard to these attributes: that a mixin node-type is capable of being a supertype of a primary node type, and therefore that a mixin-status of *primary* in the subtype overrides a mixin-status of *mixin* in the supertype (See, 6.7.22.2 *Additions to the Hierarchy*).

Other than this, the specification does not define how conflicts between multiple supertypes are resolved or how these three top level attributes are inherited. For example, the question of whether the orderable child nodes setting of a node type is inherited by its subtypes is left up to the particular implementation.

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 6.7.11 *Discovering the Definition of a Node Type*.

6.7.9 Discovering available Node Types

Discovery of which node types are available in a content repository is done through 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 nodeTypeName)** Returns the **NodeType** specified by **nodeTypeName**. 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**	
NodeType	**getPrimaryNodeType()** Returns the primary node type assigned to this node. Which **NodeType** 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. A **RepositoryException** is thrown if an error occurs.
NodeType[]	**getMixinNodeTypes()** Returns an array of **NodeType** 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 **jcr:mixinTypes**. 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*.
	A **RepositoryException** is thrown if an error occurs.
boolean	**isNodeType(String nodeTypeName)**
	Returns true if this node is of the specified primary node type or mixin type, or a subtype thereof. Returns **false** otherwise.
	A **RepositoryException** is thrown if an error occurs.

6.7.11 Discovering the Definition of a Node Type

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

javax.jcr.nodetype. **NodeType**	
String	**getName()**
	Returns the name of the node type.
boolean	**isMixin()**
	Returns **true** if this is a mixin type; returns **false** if it is primary.
boolean	**hasOrderableChildNodes()**
	Returns **true** if nodes of this type must support orderable child nodes; returns **false** otherwise. If a node type returns **true** on a call to this method, then all nodes of that node type *must* support the method **Node.orderBefore**. If a node type returns **false** on a call to this method, then nodes of that node type *may* support **Node.orderBefore**. 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 7.1.11 *Ordering Child Nodes*.
String	**getPrimaryItemName()**
	Returns the name of the primary item (one of the child items of the nodes of this node type). If this node has no primary item, then this method returns **null**. This indicator is used by the method **Node.getPrimaryItem()**. See 6.2.3 *Node Read Methods*.

`NodeType[]`	**`getSupertypes()`** Returns all supertypes of this node type in the node type inheritance hierarchy. For primary types apart from **`nt:base`**, this list will always include at least **`nt:base`**. For mixin types, there is no required supertype.
`NodeType[]`	**`getDeclaredSupertypes()`** Returns the *direct* 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.
`boolean`	**`isNodeType(String nodeTypeName)`** Returns **`true`** if this node type is **`nodeTypeName`** or a subtype of **`nodeTypeName`**, otherwise returns **`false`**.
`PropertyDefinition[]`	**`getPropertyDefinitions()`** 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.
`PropertyDefinition[]`	**`getDeclaredPropertyDefinitions()`** Returns an array containing the property definitions actually declared in this node type.
`NodeDefinition[]`	**`getChildNodeDefinitions()`** 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 inherited from the supertypes of this node type.
`NodeDefinition[]`	**`getDeclaredChildNodeDefinitions()`** Returns an array containing the child node definitions actually declared in this node type.
`boolean`	**`canSetProperty(String propertyName,`** **`Value value)`** Returns **`true`** if setting **`propertyName`** to **`value`** is

	allowed by this node type. Otherwise returns **false**.
boolean	**canSetProperty(String propertyName,** **Value[] values)** Returns **true** if setting **propertyName** to **values** is allowed by this node type. Otherwise returns **false**.
boolean	**canAddChildNode(String childNodeName)** Returns **true** if this node type allows the addition of a child node called **childNodeName** without specific node type information (that is, given the definition of this parent node type, the child node name is sufficient to determine the intended child node type). Returns **false** otherwise.
boolean	**canAddChildNode(String childNodeName,** **String nodeTypeName)** Returns **true** if this node type allows the addition of a child node called **childNodeName** of node type **nodeTypeName**. Returns **false** otherwise.
boolean	**canRemoveItem(String itemName)** Returns **true** if removing the child item called **itemName** is allowed by this node type. Returns **false** otherwise.

6.7.12 ItemDefinition

The **ItemDefinition** is the super-interface of **PropertyDefinition** and **NodeDefinition**. It encapsulates the methods common to both.

javax.jcr.nodetype. **ItemDefinition**	
NodeType	**getDeclaringNodeType()** Gets the node type that contains the declaration of this **ItemDefinition**.
String	**getName()** Gets the name of the item to which this definition applies. If "*****", then this **ItemDefinition** defines a *residual set* of child items. That is, it defines the characteristics of all those child items with names *apart from the names explicitly used in other item definitions*.

	See 6.7.15 *Residual Definitions*.
`boolean`	**`isAutoCreated()`** Reports whether the item is to be automatically created when its parent node is created. If `true` then this `ItemDefinition` will necessarily *not* 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*).
`boolean`	**`isMandatory()`** Reports whether the item is mandatory. A mandatory item is one that, if its parent node exists, must also exist. This means that a mandatory single-value property must have a value (since there is no such thing a `null` value). In the case of multi-value properties this means that the property must exist, though it can have zero or more values. An attempt (in a level 2 implementation) to `save` a node that has a mandatory child item without first creating that child item will throw a `ConstraintViolationException` on `save`.
`int`	**`getOnParentVersion()`** 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*.
`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 directly through 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.

6.7.13 PropertyDefinition

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

javax.jcr.nodetype. **PropertyDefinition** extends **ItemDefinition**	
`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 *optional*. 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 *Node Type Definitions in Content*). 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. 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.
`Value[]`	**getDefaultValues()** Gets the default value(s) of the property. These are the values (or value) that the property defined by this **PropertyDefinition** will be assigned if it is automatically created (that is, if **isAutoCreated()** returns **true**). 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. 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 *must* be the case if **isAutoCreated** returns **true** and this method returns **null**. 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 *Node Type Definitions in Content*).
boolean	**isMultiple()** 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 the 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*).

6.7.14 NodeDefinition

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

javax.jcr.nodetype. **NodeDefinition** extends ItemDefinition	
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 any particular node instance still has only one assigned primary node type, but in multiple-inheritance-supporting implementations the **RequiredPrimaryTypes** attribute can be used to restrict that assigned node type to be a subtype of *all* of a specified set of node types.
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 *Node Type Definitions in Content*).
`boolean`	`allowsSameNameSiblings()` 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 `PropertyDefinition` or `NodeDefinition` is "`*`", this indicates that the definition is a *residual definition*.

A residual definition defines the characteristics of all properties (if it is a `PropertyDefinition`) or child nodes (if it is a `NodeDefinition`) *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 `PropertyDefinition.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 regular expression-specific meta-characters) like "**banana**" matches only the string "**banana**" (see 6.7.16.1 *Choice Lists*, below).

- **PATH**: The constraint string is a path terminating with either no final "**/**", a single "**/**" or the substring "**/***". 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 constraint 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 *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 *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 type[10]. 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

---

[10] 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.

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.

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.

- `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 **NodeDefinition** or **PropertyDefinition** applicable to a particular node or property.

javax.jcr. **Node**	
NodeDefinition	**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 *that* definition that 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 **NodeDefinition** returned when this method is called on the root node of a workspace is also up to the implementation. Throws a **RepositoryException** if an error occurs.

javax.jcr. **Property**	
PropertyDefinition	**getDefinition()** 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 *that* 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. Throws a **RepositoryException** if an error occurs.

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.

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
  8.4 *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
  ...
ChildNodeDefinition
  Name ...
  RequiredPrimaryTypes ...
  DefaultPrimaryType ...
  AutoCreated ...
  Mandatory ...
  OnParentVersion ...
  Protected ...
  SameNameSiblings ...
.
. (more ChildNodeDefinitions)
.
PropertyDefinition
  Name ...
```

```
        RequiredType ...
        ValueConstraints ...
        DefaultValues ...
        AutoCreated ...
        Mandatory ...
        OnParentVersion ...
        Protected ...
        Multiple ...
.
. (more PropertyDefinitions)
.
```

6.7.20 Node Type Definitions in Content

It is *optional* for the repository to expose the definitions of its available node types as content. However, if it does expose these definitions then it should expose them using the built-in node type **nt:nodeType** (and its associated node types **nt:propertyDefinition** and **nt:childNodeDefinition**). These node types are defined to store node type definitions themselves. For example, to store a *PropertyDefinition* a node of type **nt:propertyDefinition** 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 *ChildNodeDefinitions* 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:childNodeDefinition** or **nt:propertyDefinition** node the property representing that attribute must simply be not present, since null values for properties are not allowed (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**,

```
NodeTypeName
  nt:file
Supertypes
  nt:hierarchyNode
IsMixin
  false
HasOrderableChildNodes
  false
PrimaryItemName
  jcr:content
ChildNodeDefinition
  Name jcr:content
```

```
RequiredPrimaryTypes [nt:base]
DefaultPrimaryType null
AutoCreated false
Mandatory true
OnParentVersion COPY
Protected false
SameNameSiblings false
```

the child node `jcr:content` must exist (hence `Mandatory` is **true**, but it must be added by the client, not automatically (hence `AutoCreated` 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 `ItemDefinition.getName()` is "*****". However, "*****" is not a valid value for the property `jcr:name` in a `nt:propertyDefinition` or `nt:childNodeDefinition` node (because `jcr:name` it is a **NAME** property, not a **STRING**). As a result, an in-content definition of a residual item will simply not have a `jcr:name` property. 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 supertype for mixin types.

6.7.21.1 mix:lockable

```
NodeTypeName
  mix:lockable
Supertypes
  []
IsMixin
  true
HasOrderableChildNodes
  false
PrimaryItemName
  null
PropertyDefinition
  Name jcr:lockOwner
  RequiredType STRING
  ValueConstraints []
```

```
        DefaultValues null
        AutoCreated false
        Mandatory false
        OnParentVersion IGNORE
        Protected true
        Multiple false
   PropertyDefinition
        Name jcr:lockIsDeep
        RequiredType BOOLEAN
        ValueConstraints []
        DefaultValues null
        AutoCreated 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 8.4 *Locking*.

6.7.21.2 mix:referenceable

```
   NodeTypeName
      mix:referenceable
   Supertypes
      []
   IsMixin
      true
   HasOrderableChildNodes
      false
   PrimaryItemName
      null
   PropertyDefinition
      Name jcr:uuid
      RequiredType STRING
      ValueConstraints []
      DefaultValues null
      AutoCreated 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
PropertyDefinition
  Name **jcr:versionHistory**
  RequiredType **REFERENCE**
  ValueConstraints **["nt:versionHistory"]**
  DefaultValues **null**
  AutoCreated **false**
  Mandatory **true**
  OnParentVersion **COPY**
  Protected **true**
  Multiple **false**
PropertyDefinition
  Name **jcr:baseVersion**
  RequiredType **REFERENCE**
  ValueConstraints **["nt:version"]**
  DefaultValues **null**
  AutoCreated **false**
  Mandatory **true**
  OnParentVersion **IGNORE**
  Protected **true**
  Multiple **false**
PropertyDefinition
  Name **jcr:isCheckedOut**
  RequiredType **BOOLEAN**
  ValueConstraints **[]**
  DefaultValues **[true]**
  AutoCreated **true**
  Mandatory **true**
  OnParentVersion **IGNORE**
  Protected **true**
  Multiple **false**
PropertyDefinition
  Name **jcr:predecessors**
  RequiredType **REFERENCE**
  ValueConstraints **["nt:version"]**
  DefaultValues **null**
  AutoCreated **false**
  Mandatory **true**
  OnParentVersion **COPY**
  Protected **true**
  Multiple **true**
PropertyDefinition
  Name **jcr:mergeFailed**
  RequiredType **REFERENCE**
  ValueConstraints **["nt:version"]**
  DefaultValues **null**
  AutoCreated **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:propertyDefinition** and **nt:childNodeDefinition** 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:propertyDefinition
|
|--nt:childNodeDefinition
|
|--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
PropertyDefinition
  Name jcr:primaryType
  RequiredType NAME
  ValueConstraints []
  DefaultValues null
  AutoCreated true
  Mandatory true
  OnParentVersion COMPUTE
  Protected true
  Multiple false
PropertyDefinition
  Name jcr:mixinTypes
  RequiredType NAME
  ValueConstraints []
  DefaultValues null
  AutoCreated 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. An application using the API can read the properties but cannot remove or alter them.

6.7.22.4 nt:unstructured

```
NodeTypeName
  nt:unstructured
Supertypes
  nt:base
IsMixin
  false
HasOrderableChildNodes
  true
PrimaryItemName
  null
ChildNodeDefinition
  Name *
  RequiredPrimaryTypes [nt:base]
  DefaultPrimaryType nt:unstructured
  AutoCreated false
  Mandatory false
  OnParentVersion VERSION
  Protected false
  SameNameSiblings true
PropertyDefinition
  Name *
  RequiredType UNDEFINED
  ValueConstraints []
  DefaultValues null
  AutoCreated false
  Mandatory false
  OnParentVersion COPY
  Protected false
  Multiple true
PropertyDefinition
  Name *
  RequiredType UNDEFINED
  ValueConstraints []
  DefaultValues null
  AutoCreated false
  Mandatory false
  OnParentVersion COPY
  Protected false
  Multiple false
```

This node type 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

NodeTypeName
  nt:hierarchyNode
Supertypes
  nt:base
IsMixin
  false
HasOrderableChildNodes
  false
PrimaryItemName
  null
PropertyDefinition
  Name **jcr:created**
  RequiredType **DATE**
  ValueConstraints **[]**
  DefaultValues **null**
  AutoCreated **true**
  Mandatory **false**
  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

NodeTypeName
  nt:file
Supertypes
  nt:hierarchyNode
IsMixin
  false
HasOrderableChildNodes
  false
PrimaryItemName
  jcr:content
ChildNodeDefinition
  Name **jcr:content**
  RequiredPrimaryTypes **[nt:base]**
  DefaultPrimaryType **null**
  AutoCreated **false**
  Mandatory **true**
  OnParentVersion **COPY**
  Protected **false**
  SameNameSiblings **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
PropertyDefinition
  Name jcr:content
  RequiredType REFERENCE
  ValueConstraints []
  DefaultValues null
  AutoCreated 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
ChildNodeDefinition
  Name *
  RequiredPrimaryTypes [nt:hierarchyNode]
  DefaultPrimaryType null
  AutoCreated false
  Mandatory false
  OnParentVersion VERSION
  Protected false
  SameNameSiblings 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
PropertyDefinition
  Name jcr:encoding
  RequiredType STRING
  ValueConstraints []
  DefaultValues null
  AutoCreated false
  Mandatory false
  OnParentVersion COPY
  Protected false
  Multiple false
PropertyDefinition
  Name jcr:mimeType
  RequiredType STRING
  ValueConstraints []
  DefaultValues null
  AutoCreated false
  Mandatory true
  OnParentVersion COPY
  Protected false
  Multiple false
PropertyDefinition
  Name jcr:data
```

161

```
          RequiredType BINARY
          ValueConstraints []
          DefaultValues null
          AutoCreated false
          Mandatory true
          OnParentVersion COPY
          Protected false
          Multiple false
        PropertyDefinition
          Name jcr:lastModified
          RequiredType DATE
          ValueConstraints []
          DefaultValues null
          AutoCreated 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`. The `jcr:encoding` property indicates the character set encoding used. If this resource is does not contain character data then this property will not be present. If the resource does hold character data then this property should hold one of the character set names defined in `http://www.iana.org/assignments/character-sets`. The `jcr:mimeType` should contain the name of the media type of this resource as defined in `http://www.iana.org/assignments/media-types/`.

6.7.22.10 nt:nodeType

```
        NodeTypeName
          nt:nodeType
        Supertypes
          nt:base
        IsMixin
          false
        HasOrderableChildNodes
          false
        PrimaryItemName
          null
        PropertyDefinition
          Name jcr:nodeTypeName
          RequiredType NAME
          ValueConstraints []
          DefaultValues null
          AutoCreated false
          Mandatory true
          OnParentVersion COPY
          Protected false
          Multiple false
        PropertyDefinition
          Name jcr:supertypes
          RequiredType NAME
```

```
          ValueConstraints []
          DefaultValues null
          AutoCreated false
          Mandatory false
          OnParentVersion COPY
          Protected false
          Multiple true
PropertyDefinition
          Name jcr:isMixin
          RequiredType BOOLEAN
          ValueConstraints []
          DefaultValues null
          AutoCreated false
          Mandatory true
          OnParentVersion COPY
          Protected false
          Multiple false
PropertyDefinition
          Name jcr:hasOrderableChildNodes
          RequiredType BOOLEAN
          ValueConstraints []
          DefaultValues null
          AutoCreated false
          Mandatory true
          OnParentVersion COPY
          Protected false
          Multiple false
PropertyDefinition
          Name jcr:primaryItemName
          RequiredType NAME
          ValueConstraints []
          DefaultValues null
          AutoCreated false
          Mandatory false
          OnParentVersion COPY
          Protected false
          Multiple false
ChildNodeDefinition
          Name jcr:propertyDefinition
          RequiredPrimaryTypes [nt:propertyDefinition]
          DefaultPrimaryType nt:propertyDefinition
          AutoCreated false
          Mandatory false
          OnParentVersion VERSION
          Protected false
          SameNameSiblings true
ChildNodeDefinition
          Name jcr:childNodeDefinition
          RequiredPrimaryTypes [nt:childNodeDefinition]
          DefaultPrimaryType nt:childNodeDefinition
          AutoCreated false
          Mandatory false
          OnParentVersion VERSION
          Protected false
          SameNameSiblings 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:propertyDefinition

NodeTypeName
  nt:propertyDefinition
Supertypes
  nt:base
IsMixin
  false
HasOrderableChildNodes
  false
PrimaryItemName
  null
PropertyDefinition
  Name **jcr:name**
  RequiredType **NAME**
  ValueConstraints **[]**
  DefaultValues **null**
  AutoCreated **false**
  Mandatory **false**
  OnParentVersion **COPY**
  Protected **false**
  Multiple **false**
PropertyDefinition
  Name **jcr:autoCreated**
  RequiredType **BOOLEAN**
  ValueConstraints **[]**
  DefaultValues **null**
  AutoCreated **false**
  Mandatory **true**
  OnParentVersion **COPY**
  Protected **false**
  Multiple **false**
PropertyDefinition
  Name **jcr:mandatory**
  RequiredType **BOOLEAN**
  ValueConstraints **[]**
  DefaultValues **null**
  AutoCreated **false**
  Mandatory **true**
  OnParentVersion **COPY**
  Protected **false**
  Multiple **false**
PropertyDefinition
  Name **jcr:onParentVersion**
  RequiredType **STRING**
  ValueConstraints **["COPY", "VERSION", "INITIALIZE",**
                    "COMPUTE", "IGNORE", "ABORT"]
  DefaultValues **null**
  AutoCreated **false**
  Mandatory **true**
  OnParentVersion **COPY**
  Protected **false**
  Multiple **false**
PropertyDefinition
  Name **jcr:protected**
  RequiredType **BOOLEAN**
  ValueConstraints **[]**
  DefaultValues **null**
  AutoCreated **false**
  Mandatory **true**
  OnParentVersion **COPY**

```
        Protected false
        Multiple false
    PropertyDefinition
      Name jcr:requiredType
      RequiredType STRING
      ValueConstraints ["STRING", "BINARY", "LONG", "DOUBLE",
                        "BOOLEAN", "DATE", "NAME", "PATH",
                        "REFERENCE", "UNDEFINED"]
        DefaultValues null
        AutoCreated false
        Mandatory true
        OnParentVersion COPY
        Protected false
        Multiple false
    PropertyDefinition
      Name jcr:valueConstraints
      RequiredType STRING
      ValueConstraints []
        DefaultValues null
        AutoCreated false
        Mandatory false
        OnParentVersion COPY
        Protected false
        Multiple true
    PropertyDefinition
      Name jcr:defaultValues
      RequiredType UNDEFINED
      ValueConstraints []
        DefaultValues null
        AutoCreated false
        Mandatory false
        OnParentVersion COPY
        Protected false
        Multiple true
    PropertyDefinition
      Name jcr:multiple
      RequiredType BOOLEAN
      ValueConstraints []
        DefaultValues null
        AutoCreated 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:childNodeDefinition`.

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:propertyDefinition` node.

6.7.22.12 nt:childNodeDefinition

```
NodeTypeName
  nt:childNodeDefinition
Supertypes
  nt:base
```

IsMixin
  false
HasOrderableChildNodes
  false
PrimaryItemName
  null
PropertyDefinition
  Name **jcr:name**
  RequiredType **NAME**
  ValueConstraints **[]**
  DefaultValues **null**
  AutoCreated **false**
  Mandatory **false**
  OnParentVersion **COPY**
  Protected **false**
  Multiple **false**
PropertyDefinition
  Name **jcr:autoCreated**
  RequiredType **BOOLEAN**
  ValueConstraints **[]**
  DefaultValues **null**
  AutoCreated **false**
  Mandatory **true**
  OnParentVersion **COPY**
  Protected **false**
  Multiple **false**
PropertyDefinition
  Name **jcr:mandatory**
  RequiredType **BOOLEAN**
  ValueConstraints **[]**
  DefaultValues **null**
  AutoCreated **false**
  Mandatory **true**
  OnParentVersion **COPY**
  Protected **false**
  Multiple **false**
PropertyDefinition
  Name **jcr:onParentVersion**
  RequiredType **STRING**
  ValueConstraints **["COPY", "VERSION", "INITIALIZE",**
                  "COMPUTE", "IGNORE", "ABORT"]
  DefaultValues **null**
  AutoCreated **false**
  Mandatory **true**
  OnParentVersion **COPY**
  Protected **false**
  Multiple **false**
PropertyDefinition
  Name **jcr:protected**
  RequiredType **BOOLEAN**
  ValueConstraints **[]**
  DefaultValues **null**
  AutoCreated **false**
  Mandatory **true**
  OnParentVersion **COPY**
  Protected **false**
  Multiple **false**
PropertyDefinition
  Name **jcr:requiredPrimaryTypes**
  RequiredType **NAME**
  ValueConstraints **[]**

```
        DefaultValues [nt:base]
        AutoCreated false
        Mandatory true
        OnParentVersion COPY
        Protected false
        Multiple true
  PropertyDefinition
        Name jcr:defaultPrimaryType
        RequiredType NAME
        ValueConstraints []
        DefaultValues null
        AutoCreated false
        Mandatory false
        OnParentVersion COPY
        Protected false
        Multiple false
  PropertyDefinition
        Name jcr:sameNameSiblings
        RequiredType BOOLEAN
        ValueConstraints []
        DefaultValues null
        AutoCreated 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:propertyDefinition**.

6.7.22.13 nt:versionHistory

```
  NodeTypeName
        nt:versionHistory
  Supertypes
        nt:base
        mix:referenceable
  IsMixin
        false
  HasOrderableChildNodes
        false
  PrimaryItemName
        null
  PropertyDefinition
        Name jcr:versionableUuid
        RequiredType STRING
        ValueConstraints []
        DefaultValues null
        AutoCreated true
        Mandatory true
        OnParentVersion ABORT
        Protected true
        Multiple false
  ChildNodeDefinition
        Name jcr:rootVersion
        RequiredPrimaryTypes [nt:version]
        DefaultPrimaryType nt:version
        AutoCreated true
        Mandatory true
```

```
      OnParentVersion ABORT
      Protected true
      SameNameSiblings false
ChildNodeDefinition
      Name jcr:versionLabels
      RequiredPrimaryTypes [nt:versionLabels]
      DefaultPrimaryType nt:versionLabels
      AutoCreated true
      Mandatory true
      OnParentVersion ABORT
      Protected true
      SameNameSiblings false
ChildNodeDefinition
      Name *
      RequiredPrimaryTypes [nt:version]
      DefaultPrimaryType nt:version
      AutoCreated false
      Mandatory false
      OnParentVersion ABORT
      Protected true
      SameNameSiblings 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
PropertyDefinition
      Name *
      RequiredType REFERENCE
      ValueConstraints ["nt:version"]
      DefaultValues null
      AutoCreated 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
PropertyDefinition
  Name **jcr:created**
  RequiredType **DATE**
  ValueConstraints **[]**
  DefaultValues **null**
  AutoCreated **true**
  Mandatory **true**
  OnParentVersion **ABORT**
  Protected **true**
  Multiple **false**
PropertyDefinition
  Name **jcr:predecessors**
  RequiredType **REFERENCE**
  ValueConstraints **["nt:version"]**
  DefaultValues **null**
  AutoCreated **false**
  Mandatory **false**
  OnParentVersion **ABORT**
  Protected **true**
  Multiple **true**
PropertyDefinition
  Name **jcr:successors**
  RequiredType **REFERENCE**
  ValueConstraints **["nt:version"]**
  DefaultValues **null**
  AutoCreated **false**
  Mandatory **false**
  OnParentVersion **ABORT**
  Protected **true**
  Multiple **true**
ChildNodeDefinition
  Name **jcr:frozenNode**
  RequiredPrimaryTypes **[nt:frozenNode]**
  DefaultPrimaryType **null**
  AutoCreated **false**
  Mandatory **false**
  OnParentVersion **ABORT**
  Protected **true**
  SameNameSiblings **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
PropertyDefinition
  Name **jcr:frozenPrimaryType**
  RequiredType **NAME**
  ValueConstraints **[]**
  DefaultValues **null**
  AutoCreated **true**
  Mandatory **true**
  OnParentVersion **ABORT**
  Protected **true**
  Multiple **false**
PropertyDefinition
  Name **jcr:frozenMixinTypes**
  RequiredType **NAME**
  ValueConstraints **[]**
  DefaultValues **null**
  AutoCreated **false**
  Mandatory **false**
  OnParentVersion **ABORT**
  Protected **true**
  Multiple **true**
PropertyDefinition
  Name **jcr:frozenUuid**
  RequiredType **STRING**
  ValueConstraints **[]**
  DefaultValues **null**
  AutoCreated **true**
  Mandatory **true**
  OnParentVersion **ABORT**
  Protected **true**
  Multiple **false**
PropertyDefinition
  Name *****
  RequiredType **UNDEFINED**
  ValueConstraints **[]**
  DefaultValues **null**
  AutoCreated **false**
  Mandatory **false**
  OnParentVersion **ABORT**
  Protected **true**
  Multiple **false**
PropertyDefinition
  Name *****
  RequiredType **UNDEFINED**
  ValueConstraints **[]**
  DefaultValues **null**
  AutoCreated **false**
  Mandatory **false**
  OnParentVersion **ABORT**

```
    Protected true
    Multiple true
ChildNodeDefinition
    Name *
    RequiredPrimaryTypes [nt:base]
    DefaultPrimaryType null
    AutoCreated false
    Mandatory false
    OnParentVersion ABORT
    Protected true
    SameNameSiblings 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
PropertyDefinition
    Name jcr:childVersionHistory
    RequiredType REFERENCE
    ValueConstraints ["nt:versionHistory"]
    DefaultValues null
    AutoCreated 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
```

```
PropertyDefinition
  Name jcr:statement
  RequiredType STRING
  ValueConstraints []
  DefaultValues null
  AutoCreated false
  Mandatory false
  OnParentVersion COPY
  Protected false
  Multiple false
PropertyDefinition
  Name jcr:language
  RequiredType STRING
  ValueConstraints []
  DefaultValues null
  AutoCreated 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 data 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 its node type is left up to the implementation.

6.9 Access Control

A level 1 compliant implementation must support the access control discovery method **Session.checkPermission** (see below).

In the simplest cases, where an implementation does not actually support access control, the behavior of this method can be hardcoded.

In repositories that do support access control, this method reports whether a particular **Session** has permission to perform a particular action according to the relevant access control policies. 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.

6.9.1 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**, the content repository allows for authorization and authentication to be handled by JAAS (or another external mechanism) if the implementer so chooses.

To use JAAS authentication to create **Session**s with end-user identity, invocations of the **Repository.login** method that do not specify **Credentials** (i.e., either a **null Credentials** is passed or a signature without the **Credentials** parameter is used) *should* obtain the identity of the already-authenticated user by calling the static **getSubject** method of **javax.security.auth.Subject**.

The discovery mechanism for finding what permissions apply is also JAAS-compatible since it uses the JAAS-like concept of *actions*.

6.9.2 Checking Permissions

Permission checking is done through the method **Session.checkPermission**:

javax.jcr. **Session**	
void	checkPermission(String absPath, String actions)
	Determines whether this **Session** has permission to

perform the specified **actions** at the specified **absPath**. This method quietly returns if the access request is permitted, or throws a suitable **java.security.AccessControlException** otherwise.

The actions parameter is a comma separated list of action strings. The following action strings are defined:

add_node
If **checkPermission(path, "add_node")** returns quietly, then this **Session** has permission to add a node at **path**, otherwise permission is denied.

set_property
If **checkPermission(path, "set_property")** returns quietly, then this **Session** has permission to set (add or change) a property at **path**, otherwise permission is denied.

remove
If **checkPermission(path, "remove")** returns quietly, then this **Session** has permission to remove an item at **path**, otherwise permission is denied.

read
If **checkPermission(path, "read")** returns quietly, then this **Session** has permission to retrieve (and read the value of, in the case of a property) an item at **path**, otherwise permission is denied.

When more than one action is specified in the **actions** parameter, this method will only return quietly if this **Session** has permission to perform *all* of the listed actions at the specified **path**.

The information returned through this method will only reflect access control policies and not other restrictions that may exist. For example, even though **checkPermission** may indicate that a particular **Session** may add a property at **/A/B/C**, the node type of the node at **/A/B** may prevent the addition of a property called **C**.

A **RepositoryException** is thrown if an error occurs.

7 Level 2 Repository Features

The following section explains level 2 of the content repository 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

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

7.1 Writing Repository Content

Methods for writing content to the repository fall into two categories: those that write directly to the workspace and those that write to the transient storage associated with the session. The latter require a **save** for their changes to be written to the workspace, the former do not.

7.1.1.1 Writing to Transient Storage

The purpose behind the transient storage in the session is to provide a space in which complex changes can be made to content without having these changes validated at every step. When a node/property structure is fully assembled it can be saved and validated against node type and other constraints. This allows structures of nodes and properties to be temporarily invalid while they are being built.

The methods that write to the transient layer are:

- **Node.addNode**, **setProperty**, **orderBefore**, **addMixin** and **removeMixin**.

- **Property**.**setValue**.

- **Item**.**remove**.

- **Session.move** and **importXML**.

- **Query.storeAsNode**.

Changes made though these methods will only be pushed to the workspace when a **save** is called that includes the change within its scope.

Session.save persists all pending changes currently stored in the **Session** object. 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(false)** 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.

To persist any change that involves the addition, removal or re-ordering of nodes or the addition or removal of properties, the scope of the **save** must include the parent node affected.

To persist a change that involves only a change to the value of an existing property, only that property itself needs to be within the scope of the **save**.

Note that this means that the minimal scope required to persist the effect of **Node.setProperty** depends on whether the property in question already exists or not. If it does not, then the parent node must be saved. If it does, then only the property itself needs to be saved.

7.1.1.2 Writing Directly to the Workspace

The methods that write changes directly to the workspace are :

- **Node.checkin**, **checkout**, **restore**, **restoreByLabel**, **merge**, **cancelMerge**, **doneMerge**, **update**, **lock** and **unlock**.

- **Workspace.move**, **copy**, **clone**, **restore** and **importXML**

- **VersionHistory.addVersionLabel**, **removeVersionLabel**, **removeVersion**.

(some of these are relevant only in repositories that support the relevant feature: locking or versioning)

7.1.1.3 Effect of Transactions

In repositories that support transactions, all changes, whether workspace-direct or session-mediated, may be further insulated from persistent storage by their transaction context. If a set of write methods is within the scope of a transaction then the changes they make will only be reflected in persistent storage upon commit of that transaction.

7.1.1.4 Invalid States

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**.

7.1.1.5 Timing of Validation

For those write methods that require a **save**, implementers have considerable flexibility in deciding whether a particular validation is to be performed immediately during the invocation of the write method or later on **save**. For example, in the case **Node.addNode**, an implementer might immediately check that the path given is valid while postponing validation of node type constraints until **save**-time.

The *suggestion* is that an implementation should perform each validation as soon as possible, given the underlying design of the repository.

The *requirement* is that an implementation must prohibit the emergence of a persistent state in violation the validation rules defined by this specification. Therefore at the latest, all validation must be done on **save**.

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.

7.1.1.6 Session

Session has the following **save-** and **Value-**related methods:

javax.jcr. **Session**	
void	**save()** Validates all pending changes currently recorded in this **Session**. If validation of *all* 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 **Session**s. 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 7.1.2 *Saving by UUID and Path*, for details. An **AccessDeniedException** will be thrown if any of the changes to be persisted would violate the access permissions of the **Session**. If any of the changes to be persisted would cause the removal of a node that is currently the target of a **REFERENCE** property then a **ReferentialIntegrityException** is thrown, provided that this **Session** has read access to that **REFERENCE** property. If, on the other hand, this **Session** does not have read access to the **REFERENCE** property in question, then an **AccessDeniedException** is thrown instead.

	An **ItemExistsException** will be thrown if any of the changes to be persisted would be prevented by the presence of an already existing item in the workspace. 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. 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. A **LockException** is thrown if the **save** would result in a change to persistent storage that would violate a lock. A **NoSuchNodeTypeException** is thrown if the **save** would result in the addition of a node with an unrecognized node type. A **RepositoryException** will be thrown if another error occurs.
void	**refresh(boolean keepChanges)** 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. 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 7.1.3.4 *Seeing Changes Made by Other Sessions*, for more details. A **RepositoryException** is thrown if another error occurs.
boolean	**hasPendingChanges()** Returns **true** if this **Session** holds pending (that is,

	unsaved) changes; otherwise returns **false**.
	A **RepositoryException** is thrown if an error occurs.
ValueFactory	**getValueFactory()**
	This method returns a **ValueFactory** that is used to create **Value** objects for use when setting repository properties (see 7.1.5 *Adding and Writing Properties* and 7.1.5.3 *Creating Value Objects*).
	If writing to the repository is not supported (because this is a level 1-only implementation, for example) an **UnsupportedRepositoryOperationException** will be thrown.

7.1.1.7 Item

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 *all* 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 **Session**s. 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 7.1.2 *Saving by UUID and Path*, for details.
	An **AccessDeniedException** will be thrown if any of the changes to be persisted would violate the access permissions of the **Session**.
	If any of the changes to be persisted would cause the removal of a node that is currently the target of a **REFERENCE** property then a **ReferentialIntegrityException** is thrown, provided that this **Session** has read access to that **REFERENCE** property. If, on the other hand, this **Session** does not have

	read access to the **REFERENCE** property in question, then an **AccessDeniedException** is thrown instead.
	An **ItemExistsException** will be thrown if any of the changes to be persisted would be prevented by the presence of an already existing item in the workspace.
	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.
	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.
	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.
	A **LockException** is thrown if the **save** would result in a change to persistent storage that would violate a lock.
	A **NoSuchNodeTypeException** is thrown if the **save** would result in the addition of a node with an unrecognized node type.
	A **RepositoryException** will be thrown if another error occurs.
void	**refresh(boolean keepChanges)** 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 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. 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 7.1.3.4 *Seeing Changes Made by Other Sessions*, for more details.

An **InvalidItemStateException** is thrown if this **Item** object represents a workspace item that has been removed (either by this session or another).

A **RepositoryException** is thrown if another error occurs.

7.1.2 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 using the same principles as those that govern correspondence of nodes between workspaces (see 4.10.2 **Multiple Workspaces and Corresponding Nodes**). The difference in this case is that the correspondence is not between two workspaces but between the transient storage of the session and the persistent storage of its associated workspace. In the context of saving a node, those principles amount to the following:

- 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

  o node **/a/b/c** and node **/a/b** do not have UUIDs;

  o node **/a** has UUID **u**;

  o 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**.

7.1.3 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.

7.1.3.1 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 7.1.2 *Saving by UUID and Path*.

7.1.3.2 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 application, 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(false)` 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.

7.1.3.3 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(false)` 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.

7.1.3.4 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 acquired, 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. The copy-on-read approach also has some repercussions in the context of transactions (in those implementations that support this feature). See 8.1.4 *Single Session Across Multiple Transactions*.

- **Copy-on-Write**: An alternative approach is *not* to immediately copy the state of a newly acquired **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.

7.1.3.5 Resolving Conflicts with Persistent State

When an **InvalidItemStateException** is thrown (either at write-time or **save**-time) an application may 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(false)** 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, applications may avoid such conflicts by using the locking mechanism (see 8.4 *Locking*).

7.1.3.6 Item Status

This specification 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**.

7.1.3.7 Save vs Direct

The direct-to-workspace methods should not be thought of automatically "doing a save". The effect of a directly-persistent method happens one level deeper and is therefore not necessarily equivalent to making the same change transiently and then immediately calling **save**. For example:

Following **A.addNode("B")** we have:

```
A (transient state differs from persistent because of B)
|
|--B (transient)
```

If we now import **C** as child of **A** (using the **importXML** method, though this would apply to any direct-to-workspace child addition) we have:

```
A (transient state still differs from persistent because
|   of B)
|
|--B (transient)
|
|--C (persistent)
```

If we now do an **A.save** we get an **InvalidItemStateException** because **A**'s state on the persistent layer has changed due to the import. We would now have to do an **A.refresh(false)**, discarding the effect of the **A.addNode("B")** and reverting the situation to:

```
A (persistent)
|
|--C (persistent)
```

At this point we could do the **A.addNode("B")** again and, as long as we saved it before making an further direct-to-workspace changes to **A**, the **save** would succeed.

7.1.4 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 either on the **Session**, or on the new node's parent or higher-order ancestor (grandparent, etc.). An attempt to call **save** *only* on the newly added node will throw a **RepositoryException**. In the context of this method the **relPath** provided must not have an index on its final element. If it does then a **RepositoryException** is thrown. Strictly speaking, the parameter is actually a relative path to the parent node of the node to be added, appended with the name desired for the new node (if the node is being added directly below **this** 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. Since this signature does not allow explicit node type assignment, the new node's primary node type will be determined (either immediately or on **save** depending on the implementation) by the child node definitions in the node types of its parent. See 7.4.2 *Assigning a Primary Node Type*. An **ItemExistsException** will be thrown either immediately (by this method), or on **save**, if an item at the specified path already exists and same-name siblings are not allowed. Implementations may differ on when this validation is performed. A **PathNotFoundException** will be thrown either immediately (by this method), or on **save**, if the specified path implies intermediary nodes that do not exist. Implementations may differ on when this validation is performed. A **ConstraintViolationException** will be thrown either immediately (by this method), or on **save**, if adding the node would violate a node type or implementation-specific constraint or if an attempt is made to add a node as the

	child of a property. Implementations may differ on when this validation is performed. A **VersionException** will be thrown either immediately (by this method), or on **save**, 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. Implementations may differ on when this validation is performed. A **LockException** will be thrown either immediately (by this method), or on **save**, if a lock prevents the addition of the node. Implementations may differ on when this validation is performed. A **RepositoryException** is thrown if another error occurs.
Node	**addNode(String relPath,** **String primaryNodeTypeName)** Creates a new node at **relPath** of the specified primary node type. The same as **addNode(String relPath)** except that the primary node type of the new node is explicitly specified. An **ItemExistsException** will be thrown either immediately (by this method), or on **save**, if an item at the specified path already exists and same-name siblings are not allowed. Implementations may differ on when this validation is performed. A **PathNotFoundException** will be thrown either immediately (by this method), or on **save**, if the specified path implies intermediary nodes that do not exist. Implementations may differ on when this validation is performed. A **NoSuchNodeTypeException** will be thrown either immediately (by this method), or on **save**, if the specified node type is not recognized. Implementations may differ on when this validation is performed. A **ConstraintViolationException** will be thrown either immediately (by this method), or on **save**, if adding the node would violate a node type or implementation-specific constraint or if an attempt is made to add a node as the child of a property. Implementations may differ on when this validation is performed. A **VersionException** will be thrown either immediately (by this method), or on **save**, 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. Implementations may differ on when this validation is performed.

A **LockException** will be thrown either immediately (by this method), or on **save**, if a lock prevents the addition of the node. Implementations may differ on when this validation is performed.

A **RepositoryException** is thrown if another error occurs.

7.1.4.1 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.

7.1.5 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`	`setProperty(String name,`
	`              Value value)`
	
	Sets the specified (single value) property of this node to the specified **value**. If the property does not yet exist, it is created. 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** object is different from that required, then a best-effort conversion is attempted. If the conversion fails, a **ValueFormatException** is thrown. If another error occurs, a **RepositoryException** is thrown.
	
	If the node type of this node does not indicate a specific property type, then the property type of the supplied **Value** object is used and if the property already exists (has previously been set) it assumes both the new value and new property type.
	
	If the property is multi-valued, a **ValueFormatException** is thrown.
	
	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**.
	
	To save the addition or removal of a property, a **save** call must be performed that includes the parent of the property in its scope: that is, a **save** on either the session, this node, or an ancestor of this node. To save a change to an existing property, a **save** call that includes that property in its scope is required. This means that in addition to the above-mentioned **save** options, a **save** on the changed property itself will also work.
	
	A **ConstraintViolationException** will be thrown either immediately (by this method), or on **save**, if the change would violate a node type or implementation-specific constraint. Implementations may differ on when this validation is performed.
	
	A **VersionException** will be thrown either immediately (by this method), or on **save**, if **this** node is versionable and checked-in or is non-versionable but its nearest versionable ancestor is checked-in.

	Implementations may differ on when this validation is performed. A **LockException** will be thrown either immediately (by this method), or on **save**, if a lock prevents the setting of the property. Implementations may differ on when this validation is performed. A **RepositoryException** is thrown if another error occurs.
Property	**setProperty(String name, Value[] values)** Sets the specified (multi-value) property to the specified array of values. If the property does not yet exist, it is created. Same as **setProperty(String name, Value value)** except that an array of **Value** objects is assigned instead of a single **Value**. 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. 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. 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**. Note that this is different from passing an array that contains **null** elements. In such a case, the array is compacted by removing the **null**s. The resulting set of values of the property never contains **null**s. However, the set may be empty: **N.setProperty("P", new Value[]{null})** would set the property to the empty set of values.

	To save the addition or removal of a property, a **save** call must be performed that includes the parent of the property in its scope: that is, a **save** on either the session, this node, or an ancestor of this node. To save a change to an existing property, a **save** call that includes that property in its scope is required. This means that in addition to the above-mentioned **save** options, a **save** on the changed property itself will also work.
	A **ConstraintViolationException** will be thrown either immediately (by this method), or on **save**, if the change would violate a node type or implementation-specific constraint. Implementations may differ on when this validation is performed.
	A **VersionException** will be thrown either immediately (by this method), or on **save**, if **this** node is versionable and checked-in or is non-versionable but its nearest versionable ancestor is checked-in. Implementations may differ on when this validation is performed.
	A **LockException** will be thrown either immediately (by this method), or on **save**, if a lock prevents the setting of the property. Implementations may differ on when this validation is performed.
	A **RepositoryException** is thrown if another error occurs.

Property	`setProperty(String name,`
	`            Value[] values,`
	`            int type)`
	
	Sets the specified (multi-value) property to the specified array of values. If the property does not yet exist, it is created. The type of the property is determined by the **type** parameter specified.
	
	If the property type of the supplied **Value** objects is different from that specified, then a best-effort conversion is attempted. If the conversion fails, a **ValueFormatException** is thrown.
	
	If the property already exists it assumes both the new values and the new property type.
	
	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.

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**.

Note that this is different from passing an array that contains **null** elements. In such a case, the array is compacted by removing the **null**s. The resulting set of values of the property never contains **null**s. However, the set may be empty: **N.setProperty("P", new Value[]{null}, type)** would set the property to the empty set of values.

To save the addition or removal of a property, a **save** call must be performed that includes the parent of the property in its scope: that is, a **save** on either the session, this node, or an ancestor of this node. To save a change to an existing property, a **save** call that includes that property in its scope is required. This means that in addition to the above-mentioned **save** options, a **save** on the changed property itself will also work.

A **ConstraintViolationException** will be thrown either immediately (by this method), or on **save**, if the change would violate a node type or implementation-specific constraint. Implementations may differ on when this validation is performed.

A **VersionException** will be thrown either immediately (by this method), or on **save**, if **this** node is versionable and checked-in or is non-versionable but its nearest versionable ancestor is checked-in. Implementations may differ on when this validation is performed.

A **LockException** will be thrown either immediately (by this method), or on **save**, if a lock prevents the setting of the property. Implementations may differ on when this validation is performed.

A **RepositoryException** is thrown if another error occurs.

Property	**setProperty(String name, String[] values)** Sets the specified property to the specified array of values. Same as

	setProperty(String name, Value[] values)
	except that the values are specified as **String** objects instead of **Value** objects.

Property	**setProperty(String name,**
	            String[] values,
	            int type)
	
	Sets the specified property to the specified array of values and to the specified type. Same as
	
	setProperty(String name,
	            Value[] values,
	            int type)
	
	except that the values are specified as **String** objects instead of **Value** objects.

Property	**setProperty(String name,**
	            Value value,
	            int type)
	
	Sets the specified (single-value) property to the specified value. If the property does not yet exist, it is created. The type of the property is determined by the **type** parameter specified.
	
	If the property type of the supplied **Value** object is different from that required, then a best-effort conversion is attempted. If the conversion fails, a **ValueFormatException** is thrown.
	
	If the property is not single-valued then a **ValueFormatException** is also thrown.
	
	If the property already exists it assumes both the new value and the new property type.
	
	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**.
	
	To save the addition or removal of a property, a **save** call must be performed that includes the parent of the property in its scope: that is, a **save** on either the session, this node, or an ancestor of this node. To save a change to an existing property, a **save** call that includes that property in its scope is required. This means that in addition to the above-mentioned **save** options, a **save** on the changed property itself will also

	work.
	A **ConstraintViolationException** will be thrown either immediately (by this method), or on **save**, if the change would violate a node type or implementation-specific constraint. Implementations may differ on when this validation is performed.
	A **VersionException** will be thrown either immediately (by this method), or on **save**, if **this** node is versionable and checked-in or is non-versionable but its nearest versionable ancestor is checked-in. Implementations may differ on when this validation is performed.
	A **LockException** will be thrown either immediately (by this method), or on **save**, if a lock prevents the setting of the property. Implementations may differ on when this validation is performed.
	If another error occurs, a **RepositoryException** is thrown.
| **Property** | ```
setProperty(String name,
            String value,
            int type)
```<br><br>Sets the specified property to the specified value. Same as<br><br>```
setProperty(String name,
            Value value,
            int type)
```<br><br>except that the value is specified as a **String** object instead of a **Value** object. |
| **Property** | ```
setProperty(String name, String value)
```<br><br>```
setProperty(String name, InputStream value)
```<br><br>```
setProperty(String name, boolean value)
```<br><br>```
setProperty(String name, Calendar value)
```<br><br>```
setProperty(String name, double value)
```<br><br>```
setProperty(String name, long value)
```<br><br>```
setProperty(String name, Node value)
```<br><br>Sets the specified property to the specified value. In the context of these methods, each Java type *implies* a particular property type. The correspondence is: |

```
String: PropertyType.STRING

InputStream: PropertyType.BINARY

boolean: PropertyType.BOOLEAN

Calendar: PropertyType.DATE

double: PropertyType.DOUBLE

long: PropertyType.LONG

Node: PropertyType.REFERENCE
```

In the case of the signature that takes a `Node`, the `REFERENCE` property in question is set to *refer* to the supplied node (see 6.2.5.4 *Reference*).

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.

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.

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", (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`.

To save the addition or removal of a property, a `save` call must be performed that includes the parent of the property in its scope: that is, a `save` on either the session, this node, or an ancestor of this node. To save a change to an existing property, a `save` call that includes that property in its scope is required. This means that in addition to the above-mentioned `save` options, a `save` on the changed property itself will also work.

A property of type `PropertyType.NAME` or

	PropertyType.PATH may be created either by explicitly specifying the property type using a three-argument **setProperty** signature, or by using **ValueFactory** to create a property of the desired type and then calling **setProperty(String, Value)**.

A **ConstraintViolationException** will be thrown either immediately (by this method), or on **save**, if the change would violate a node type or implementation-specific constraint. Implementations may differ on when this validation is performed.

A **VersionException** will be thrown either immediately (by this method), or on **save**, if **this** node is versionable and checked-in or is non-versionable but its nearest versionable ancestor is checked-in. Implementations may differ on when this validation is performed.

A **LockException** will be thrown either immediately (by this method), or on **save**, if a lock prevents the setting of the property. Implementations may differ on when this validation is performed.

A **RepositoryException** is thrown if another error occurs. |

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) |

	Sets the value of this `Property` to the specified value. The behavior of these methods is identical with their corresponding `Node.setProperty` signature.

7.1.5.1 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);
```

7.1.5.2 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`.

7.1.5.3 Creating Value Objects

In order to use methods that accept a `Value` object the application needs a way to produce such objects. This is done by using the methods of the `ValueFactory` object, which is acquired through `Session.getValueFactory()` (see 7.1 *Writing Repository Content*). `ValueFactory` provides the following methods:

javax.jcr. **ValueFactory**	
Value	**createValue(String value)** Returns a `Value` object of `PropertyType.STRING` with the specified value.
Value	**createValue(String value, int type)** Returns a `Value` object of the `PropertyType` specified by `type` with the specified `value`. A `ValueFormatException` is thrown if the specified `value` cannot be converted to the specified `type`.

Value	**createValue(long value)** Returns a **Value** object of **PropertyType.LONG** with the specified **value**.
Value	**createValue(double value)** Returns a **Value** object of **PropertyType.DOUBLE** with the specified **value**.
Value	**createValue(boolean value)** Returns a **Value** object of **PropertyType.BOOLEAN** with the specified **value**.
Value	**createValue(Calendar value)** Returns a **Value** object of **PropertyType.DATE** with the specified **value**.
Value	**createValue(InputStream value)** Returns a **Value** object of **PropertyType.BINARY** with a value consisting of the content of the specified **InputStream**.
Value	**createValue(Node value)** Returns a **Value** object of **PropertyType.REFERENCE** that holds the UUID of the specified **Node**. This **Value** object can then be used to set a property that will be a reference to that **Node**. A **RepositoryException** is thrown if the specified **Node** is not referenceable, the current **Session** is no longer active, or another error occurs.

7.1.6 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 **ReferentialIntegrityException** will be thrown on **save** if this item or an item in its subtree is currently the target of a **REFERENCE** property located in this workspace but outside this item's subtree and the current **Session** has read access to that

	REFERENCE property. An **AccessDeniedException** will be thrown on **save** if this item or an item in its subtree is currently the target of a **REFERENCE** property located in this workspace but outside this item's subtree and the current **Session** *does not* have read access to that **REFERENCE** property. A **ConstraintViolationException** will be thrown either immediately (by this method), or on **save**, if removing this item would violate a node type or implementation-specific constraint. Implementations may differ on when this validation is performed. A **VersionException** will be thrown either immediately (by this method), or on **save**, if the parent node of this item is versionable and checked-in or is non-versionable but its nearest versionable ancestor is checked-in. Implementations may differ on when this validation is performed. A **LockException** will be thrown either immediately (by this method), or on **save**, if a lock prevents the removal of this item. Implementations may differ on when this validation is performed. A **RepositoryException** is thrown if another error occurs.

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*.

7.1.7 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`	**move(String srcAbsPath, String destAbsPath)** Moves the node at **srcAbsPath** (and its entire subtree) to the new location at **destAbsPath**. In order to persist the change, **save** must be called on either the session or a common ancestor to both the source and destination locations. A **ConstraintViolationException** is thrown either immediately (by this method) or on **save** if performing this operation would violate a node type or implementation-specific constraint. Implementations may differ on when this validation is performed. As well, a **ConstraintViolationException** will be thrown on **save** if an attempt is made to save *only* either the source or destination node separately. 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**. The **destAbsPath** provided must not have an index on its final element. If it does then a **RepositoryException** is thrown immediately. Strictly speaking, the **destAbsPath** parameter is actually an *absolute path* to the parent node of the new location, appended with the new *name* 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. 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). If no node exists at **srcAbsPath** or no node exists one level above **destAbsPath** (in other words, there is no node that will serve as the parent of the moved item) then a **PathNotFoundException** is thrown either

	immediately or on **save**. Implementations may differ on when this validation is performed.
	An **ItemExistsException** is thrown either immediately or on save if a property already exists at **destAbsPath** or a node already exists there and same-name siblings are not allowed. Implementations may differ on when this validation is performed.
	A **VersionException** is thrown either immediately or on save if the parent node of **destAbsPath** or the parent node of **srcAbsPath** is versionable and checked-in, or is non-versionable and its nearest versionable ancestor is checked-in. Implementations may differ on when this validation is performed.
	A **LockException** is thrown either immediately or on save if a lock prevents the move. Implementations may differ on when this validation is performed.
	A **RepositoryException** is thrown if another error occurs.

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

javax.jcr. **Workspace**	
void	**copy(String srcAbsPath,** **String destAbsPath)** 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**. Copies of referenceable nodes are automatically given new UUIDs. 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 *absolute path* to the parent node of the new location, appended with the new *name* 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.
	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).
	A **ConstraintViolationException** is thrown if the operation would violate a node-type or other implementation-specific constraint.
	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.
	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.
	A **PathNotFoundException** is thrown if the node at **srcAbsPath** or the parent of **destAbsPath** does not exist.
	An **ItemExistsException** is thrown if a property already exists at **destAbsPath** or a node already exists there and same-name siblings are not allowed.
	A **LockException** is thrown if a lock prevents the copy.
	A **RepositoryException** is thrown if another error occurs.
| **void** | ```
copy(String srcWorkspace,
     String srcAbsPath,
     String destAbsPath)
``` |
| | This method copies the subtree at **srcAbsPath** in **srcWorkspace** to **destAbsPath** in **this** workspace. Unlike **clone**, this method *does* 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**. |
| | 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 *absolute path* to the parent node of the new location, appended with the new *name* 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.

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).

A **NoSuchWorkspaceException** is thrown if **srcWorkspace** does not exist.

A **ConstraintViolationException** is thrown if the operation would violate a node-type or other implementation-specific constraint.

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.

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.

A **PathNotFoundException** is thrown if the node at **srcAbsPath** in **srcWorkspace** or the parent of **destAbsPath** in this workspace does not exist.

An **ItemExistsException** is thrown if a property already exists at **destAbsPath** or a node already exists there and same-name siblings are not allowed.

A **LockException** is thrown if a lock prevents the copy.

A **RepositoryException** is thrown if another error occurs. |
| **void** | ```
clone(String srcWorkspace,
      String srcAbsPath,
      String destAbsPath,
      boolean removeExisting)
```

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 *does not* assign new UUIDs to new referenceable nodes but preserves the UUIDs of their respective source |

nodes.

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.

If successful, the changes are persisted immediately, there is no need to call **save**.

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 *absolute path* to the parent node of the new location, appended with the new *name* 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.

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).

A **NoSuchWorkspaceException** is thrown if **srcWorkspace** does not exist.

A **ConstraintViolationException** is thrown if the operation would violate a node-type or other implementation-specific constraint

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.

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.

A **PathNotFoundException** is thrown if the node at **srcAbsPath** in **srcWorkspace** or the parent of **destAbsPath** in this workspace do not exist.

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. A **LockException** is thrown if a lock prevents the clone. A **RepositoryException** if another error occurs.
void	**move(String srcAbsPath, String destAbsPath)** 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 space and hence requires a **save**. 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 *absolute path* to the parent node of the new location, appended with the new *name* 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. 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). A **ConstraintViolationException** is thrown if the operation would violate a node-type or other implementation-specific constraint 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. An **AccessDeniedException** is thrown if the current session does not have sufficient access permissions to complete the operation. A **PathNotFoundException** is thrown if the item at **srcAbsPath** or the parent of **destAbsPath** does not exist.

	An **ItemExistsException** is thrown if a property already exists at **destAbsPath** or a node already exists there and same-name siblings are not allowed.
	A **LockException** is thrown if a lock prevents the move.
	A **RepositoryException** is thrown if another error occurs.

7.1.7.1 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.

7.1.8 Updating and Cloning Nodes across Workspaces

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

As mentioned in 4.10.2 **Multiple Workspaces and Corresponding Nodes**, a node's corresponding node is defined as follows:

- A node's corresponding nodes are those with the same *correspondence identifier*.

- The correspondence identifier of a referenceable node is its UUID.

- The correspondence identifier of a non-referenceable node with at least one referenceable ancestor is the pair consisting of the UUID of its nearest referenceable ancestor and its relative path from that ancestor.

- The correspondence identifier of a non-referenceable node with no referenceable ancestor is its absolute path.

Note also that (as stated in 4.9 **Referenceable Nodes**) if a repository has a workspace with a referenceable root node then *all* its workspaces must have referenceable root nodes *and* those root nodes must all have the same UUID.

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

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

7.1.8.1 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.

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).

7.1.8.2 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 **srcWorkspaceName**, then this replaces this node and its subtree with a clone of the corresponding node and its subtree. If this node does not have a corresponding node in the workspace **srcWorkspaceName**, then the update method has no effect. If the **update** succeeds, the changes made to **this** node and its subtree are persisted immediately, there is no need to call **save**. Note that **update** does not respect the checked-in status of nodes. An **update** may change a node even if it is currently checked-in (this fact is only relevant in an implementation that supports versioning, see 8.2 *Versioning*). If the specified **srcWorkspace** does not exist, a **NoSuchWorkspaceException** is thrown. If the current session does not have sufficient permissions to perform the operation, then an **AccessDeniedException** is thrown. An **InvalidItemStateException** is thrown if this **Session** (not necessarily this **Node**) has pending unsaved changes. A **LockException** is thrown if a lock prevents the update. A **RepositoryException** is thrown if another error occurs.

7.1.8.3 getCorrespondingNodePath

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

`javax.jcr.` **Node**	
`String`	**getCorrespondingNodePath(String workspaceName)** Returns the absolute path of the node in the specified workspace that corresponds to **this** node. If no corresponding node exists then an **ItemNotFoundException** is thrown. If the specified workspace does not exist then a **NoSuchWorkspaceException** is thrown. If the current **Session** does not have sufficient permissions to perform this operation, an **AccessDeniedException** is thrown. Throws a **RepositoryException** if another error occurs.

7.1.9 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**.

7.1.10 Treatment of UUIDs

A number of different methods in the 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	
Workspace.copy	**yes**	no	no	no	**copy** (both within

(see 7.1.7 *Moving and Copying*)					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 7.1.8 *Updating and Cloning Nodes across Workspaces*)	no	no	no	**yes**	`update` pulls the state of **this** node from another workspace using UUID matching, regardless of where (in terms of path) the source node is in the source workspace.
`Workspace.clone` (see 7.1.7 *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.importXML` `Session.importXML` `Session.` `getImportContentHandler`, `Workspace.` `getImportContentHandler` (see 7.3 *Importing Repository Content*)	**yes**	**yes**	**yes**	**yes**	All four options are supported.

7.1.11 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)`
	
	If this node supports child node ordering, this method inserts the child node at **srcChildRelPath** before its sibling, the child node at **destChildRelPath**, in the child node list. To place the node **srcChildRelPath** at the end of the list, a **destChildRelPath** of **null** is used.
	
	Note that (apart from the case where **destChildRelPath** is **null**) both of these arguments must be relative paths of depth one, in other words they are the *names* of the child nodes, possibly suffixed with an index (see 4.6.1 *Names vs. Paths*).
	
	If **srcChildRelPath** and **destChildRelPath** are the same, then no change is made.
	
	Changes to ordering of child nodes are persisted on **save** of the parent node.
	
	If this node does not support child node ordering, then a **UnsupportedRepositoryOperationException** thrown.
	
	If **srcChildRelPath** is not the relative path to a child node of this

	node then an **ItemNotFoundException** is thrown.
	If **destChildRelPath** is neither the relative path to a child node of this node nor **null**, then an **ItemNotFoundException** is also thrown.
	A **ConstraintViolationException** will be thrown either immediately (by this method), or on **save**, if this operation would violate a implementation-specific constraint. Implementations may differ on when this validation is performed.
	A **VersionException** will be thrown either immediately (by this method), or on **save**, if this node is versionable and checked-in or is non-versionable but its nearest versionable ancestor is checked-in. Implementations may differ on when this validation is performed.
	A **LockException** will be thrown either immediately (by this method), or on **save**, if a lock prevents the re-ordering. Implementations may differ on when this validation is performed.
	If another error occurs a **RepositoryException** is thrown.

If a node type returns **true** on a call to **NodeType.hasOrderableChildNodes()**, then all nodes of that node type *must* support the method **Node.orderBefore**. If a node type returns **false** on a call to this method, then nodes of that node type *may* support **Node.orderBefore**. 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.

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 content 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 `prefix` and `uri` in the global namespace registry of this repository.
	Assigning a new prefix to a URI that already exists in the namespace registry erases the old prefix. In general this can 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 `NamespaceRegistry.unregisterNamespace`:
	• Attempting to re-assign a built-in prefix (`jcr`, `nt`, `mix`, `xml` or the empty prefix) to a new URI will throw a `NamespaceException`.
	• Attempting to re-assign a prefix that is currently assigned to an "in-use" URI, i.e., one that is present in content, will throw a `NamespaceException`. This applies to URIs in use within item names and those within the values of `NAME` or `PATH` properties (including those in in-content node type definitions). However, one can change the prefix for an existing URI to any available new unique prefix, thus replacing the existing shorthand for that URI.
	• Attempting to register a namespace with a prefix that begins with the characters "`xml`" (in any combination of case) will throw a `NamespaceException`.
	• An implementation may prevent the re-assignment of any other namespace prefixes for implementation-specific reasons by throwing a `NamespaceException`.
	In a level 1 implementation, this method always throws an `UnsupportedRepositoryOperationException`.

	If the session associated with the **Workspace** object through which this registry was acquired does not have sufficient permissions to register the namespace an **AccessDeniedException** is thrown. A **RepositoryException** is thrown if another error occurs.
void	**unregisterNamespace(String prefix)** Removes a namespace mapping from the registry. The following restrictions apply: - Attempting to unregister a built-in namespace (**jcr**, **nt**, **mix**, **xml** or the empty namespace) 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**. In a level 1 implementation, this method always throws an **UnsupportedRepositoryOperationException**. If the session associated with the **Workspace** object through which this registry was acquired does not have sufficient permissions to unregister the namespace an **AccessDeniedException** is thrown. A **RepositoryException** is thrown if another error occurs.

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 universally 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.

- **`xml -> http://www.w3.org/XML/1998/namespace`**
  Reserved for reasons of compatibility with XML.

- "" (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*.

7.2.1 Visibility of Namespace Registry Changes

A content repository implementation must ensure that changes to the persistent namespace registry are immediately visible to all sessions.

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 7.3.8 *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 content repository (another application is to provide a context in which XPath queries are more readable than they would be in system view, see 6.6.1 *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. Namespace declarations in the incoming XML document that do not already exist in the repository namespace registry are added to the repository namespace registry.

2. Each XML element **E** becomes a content repository node of the same name, **E**.

3. The node type of the content repository node **E** is determined by the implementation in accordance with its policy on respecting property semantics (see 7.3.3 *Respecting Property Semantics* and 7.3.4 *Determining Node Types*).

4. Each child XML element **C** of XML element **E** becomes a content repository child node **C** of node **E**.

5. Each XML attribute **A** within an XML element **E** becomes a property **A** of content repository node **E**. The value of each XML attribute **A** becomes the value of the corresponding property **A**.

6. The type of each imported property is determined by the implementation in accordance with its policy on respecting property semantics (see 7.3.3 *Respecting Property Semantics* and 7.3.4 *Determining Node Types*).

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. An implementation that respects node type information may be able to determine whether a particular attribute is intended to be a single or multi value property, and treat any spaces embedded in the value according (either as delimiters or as literal spaces). Implementations are also free to rely on other out-of-band information (such as any schema associated by the incoming XML) to help determine the intended interpretation of whitespace with a particular incoming attribute value.

9. Text within an XML element `E` becomes a `STRING` property called `jcr:xmlcharacters` of a node called `jcr:xmltext`, which itself becomes a child node of the node `E`.

10. 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`.

11. An XML element can have a child element and an attribute with the same name while a content repository node cannot have a child node and property with the same name. For example, `<a b="x"><b/></a>` would imply a content repository 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 content 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
<?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: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..."
```

Note that the use of `nt:unstructured` as the default node type is an implementation-level issue, so the example is a valid outcome of the import, but not the only one.

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` (that is properties such as `jcr:primaryType`, `jcr:mixinTypes`, `jcr:uuid`, `jcr:lockIsDeep`, and so forth).

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 7.3.6 *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`.

If an implementation chooses to skip `jcr:primaryType`, the node type of the imported node is determined by the implementation (see 7.3.4 *Determining Node Types*, immediately below).

7.3.4 Determining Node Types

In cases of XML import where primary node type information is unavailable, either because it is skipped (see 7.3.3 *Respecting Property Semantics*, immediately above) or because it is not available in the first place (as is the case on document view import of arbitrary XML), the implementation must determine an appropriate node type to assign to each newly created node. This specification does not attempt to define or restrict how this is done. However, simply for the sake of illustration, some of the possibilities include:

- Making all imported nodes `nt:unstructured` (obviously this will only work if the implementation in question supports that node type).

- Dynamically creating node types appropriate to the incoming nodes. This approach might be suitable in cases where the incoming structures all fall into a few well defined and easily recognized patterns.

- Use node types created according to structure information provided to the repository from an external source such as a schema.

7.3.5 Determining Property Types

In document view import (unlike system view import) property type information is not explicitly recorded. As a result, the

implementation must determine a suitable property type for each incoming property. The determination of the property type for a particular incoming property must be done according to the following principles:

- If the property type is determinable from the node type assigned to its node (regardless of how this node type is itself determined; see 7.3.4 *Determining Node Types*) then that property type is used.

- If the property type is not determinable from the node type assigned to its node then the determination of the property is left up to the implementation. The most common possibility is to default to the type **STRING**, though this is not required. Some implementations may choose, for example, to attempt "guess" the type according to an analysis of the content. This specification does not attempt to define or restrict how an implementation handles this case.

7.3.6 Workspace Import Methods

The **Workspace** interface provides the following methods for importing content into the repository:

javax.jcr. **Workspace**	

ContentHandler	**getImportContentHandler(**
	                     String parentAbsPath,
	                     int uuidBehavior)
	
	 Returns an **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 *system view* XML document then it is interpreted as a *document view* XML document.
	
	 The incoming XML is deserialized into a subtree of items immediately below the node at **parentAbsPath**.
	
	 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**.
	
	 As SAX events are fed into the **ContentHandler**, 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 structures that violate node type constraints cannot be imported, fixed and then saved. Instead, a constraint violation will cause the **ContentHandler** to throw a **SAXException**. See **Session.getImportContentHandler** for a version of this method that *does* go through the **Session**.

The flag **uuidBehavior** governs how the UUIDs of incoming (deserialized) nodes are handled. There are four options (defined as constants in the interface **javax.jcr.ImportUUIDBehavior**):

- **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 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.

- **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.

A **SAXException** will be thrown by the returned **ContentHandler** during deserialization if the top-most 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.

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.

A **PathNotFoundException** is thrown if no node exists at **parentAbsPath**.

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.

Unlike **Session.getImportContentHandler**, this method will also enforce node type constraints by having the returned **ContentHandler** throw a **SAXException** during deserialization. However, which node type constraints are enforced depends upon whether node type information in the imported data is respected, and this is an implementation-specific issue (see 7.3.3 *Respecting Property Semantics*).

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.

A **LockException** is thrown if a lock prevents the addition of the subtree.

An **AccessDeniedException** is thrown if the session associated with this **Workspace** object does not have sufficient permissions to perform the import.

A **RepositoryException** is thrown if another error occurs.

void	`importXML(String parentAbsPath,`
	`         InputStream in,`
	`         int uuidBehavior)`
	
	Deserializes an XML document and adds the resulting item subtree as a child of the node at `parentAbsPath`.
	
	If the incoming XML stream does not appear to be a *system view* XML document then it is interpreted as a *document view* XML document.
	
	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 *does* go through the `Session`.
	
	The flag `uuidBehavior` governs how the UUIDs of incoming (deserialized) nodes are handled. There are four options (defined as constants in the interface `javax.jcr.ImportUUIDBehavior`):
	
	• `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 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.

An **ItemExistsException** will be thrown if the top-most 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.

An **IOException** is thrown if an I/O error occurs.

If no node exists at `parentAbsPath`, a **PathNotFoundException** is thrown.

If node-type or other implementation-specific constraints prevent the addition of the subtree, a **ConstraintViolationException** is thrown.

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.

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.

A **LockException** is thrown if a lock prevents the addition of the subtree.

An **AccessDeniedException** is thrown if the session associated with this **Workspace** object does not have sufficient permissions to perform the import.

If another error occurs, a **RepositoryException** is thrown.

7.3.7 Session Import Methods

The **Session** contains the following methods for importing and exporting content:

`javax.jcr.` **Session**	
`ContentHandler`	**getImportContentHandler(String parentAbsPath,** **int uuidBehavior)** Returns an **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 *system view* XML document then it is interpreted as a *document view* XML document. The incoming XML is deserialized into a subtree of items immediately below the node at **parentAbsPath**. This method simply returns the **ContentHandler** without altering the state of the session; the actual deserialization to the session transient space is done through the methods of the **ContentHandler**. Invalid XML data will cause the **ContentHandler** to throw a **SAXException**. 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 (depending on what constraint checks the implementation leaves until **save**) 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 *does not* go through the session. The flag **uuidBehavior** governs how the UUIDs of incoming (deserialized) nodes are handled. There are four options (defined as constants in the interface **javax.jcr.ImportUUIDBehavior**): • **IMPORT_UUID_CREATE_NEW**: Incoming referenceable nodes are added in the same way that new node is added with **Node.addNode**. That is, they are either assigned newly created UUIDs upon addition or upon **save** (depending on the implementation, see 4.9.1.1 *When UUIDs are*

Assigned). In either case, UUID collisions will not 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. Both the removal and the new addition will be persisted on **save**.

- **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 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. The change will be persisted on **save**.

- **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.

Unlike **Workspace.getImportContentHandler**, this method does not necessarily enforce all node type constraints during deserialization. Those that would be immediately enforced in a normal write method (**Node.addNode**, **Node.setProperty** etc.) of this implementation cause the returned **ContentHandler** to throw an immediate **SAXException** during deserialization. All other constraints are checked on **save**, just as they are in normal write operations. However, which node type constraints are enforced also depends upon whether node type information in the imported data is respected, and this is an implementation-specific issue (see 7.3.3 *Respecting*

	Property Semantics). 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. A **PathNotFoundException** is thrown either immediately or on **save** if no node exists at **parentAbsPath**. Implementations may differ on when this validation is performed. A **ConstraintViolationException** is thrown either immediately or on **save** if the new subtree cannot be added to the node at **parentAbsPath** due to node-type or other implementation-specific constraints. Implementations may differ on when this validation is performed. A **VersionException** is thrown either immediately or on **save** if the node at **parentAbsPath** is versionable and checked-in, or is non-versionable but its nearest versionable ancestor is checked-in. Implementations may differ on when this validation is performed. A **LockException** is thrown either immediately or on **save** if a lock prevents the addition of the subtree. Implementations may differ on when this validation is performed. A **RepositoryException** is thrown if another error occurs.

void	**importXML(String parentAbsPath,**
	            InputStream in,
	            int uuidBehavior)
	
	Deserializes an XML document and adds the resulting item subtree as a child of the node at **parentAbsPath**.
	
	If the incoming XML stream does not appear to be a *system view* XML document then it is interpreted as a *document view* XML document.
	
	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 (depending on what constraint checks the implementation leaves until **save**) 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.importXML** for a version of this method that *does not* go through the **Session**.

The flag **uuidBehavior** governs how the UUIDs of incoming (deserialized) nodes are handled. There are four options (defined as constants in the interface **javax.jcr.ImportUUIDBehavior**):

- **IMPORT_UUID_CREATE_NEW**: Incoming referenceable nodes are added in the same way that new node is added with **Node.addNode**. That is, they are either assigned newly created UUIDs upon addition or upon **save** (depending on the implementation, see 4.9.1.1 *When UUIDs are Assigned*). In either case, UUID collisions will not 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. Both the removal and the new addition will be persisted on **save**.

- **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 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. The change will be persisted on **save**.

- **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.

	Unlike **Workspace.importXML**, this method does not necessarily enforce all node type constraints during deserialization. Those that would be immediately enforced in a normal write method (**Node.addNode**, **Node.setProperty** etc.) of this implementation cause an immediate **ConstraintViolationException** during deserialization. All other constraints are checked on **save**, just as they are in normal write operations. However, which node type constraints are enforced depends upon whether node type information in the imported data is respected, and this is an implementation-specific issue (see 7.3.3 *Respecting Property Semantics*).
	A **ConstraintViolationException** will also be thrown immediately 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.
	A **PathNotFoundException** is thrown either immediately or on **save** if no node exists at **parentAbsPath**. Implementations may differ on when this validation is performed.
	A **VersionException** is thrown either immediately or on **save** if the node at **parentAbsPath** is versionable and checked-in, or is non-versionable but its nearest versionable ancestor is checked-in. Implementations may differ on when this validation is performed.
	A **LockException** is thrown either immediately or on **save** if a lock prevents the addition of the subtree. Implementations may differ on when this validation is performed.
	A **RepositoryException** is thrown if another error occurs.

7.3.8 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 (the constants below are defined in the interface **javax.jcr.ImportUUIDBehavior**).

IMPORT_UUID_CREATE_NEW: The XML element representing **jcr:root** is rendered as a normal node at the position specified (with the name **jcr:root**). It gets a new 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 operates 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 compliant 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 7.1.4 *Adding Nodes*).

Alternatively, in many cases the application using 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 *NodeDefinition*). 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 7.1.4 *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 **mixinName** to this node's **jcr:mixinTypes** property immediately. Semantically, the mixin node type assignment *may* take effect immediately and at the very least, it *must* take effect on **save**. A **ConstraintViolationException** is thrown either immediately or on **save** if a conflict with another assigned mixin or the primary node type or for an implementation-specific reason. Implementations may differ on when this validation is done. 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** either immediately or on **save**. If the specified mixin node type is not recognized a **NoSuchNodeTypeException** is thrown either immediately or on **save**. Implementations may differ on when this validation is done. A **VersionException** is thrown either immediately or on **save** if **this** node is versionable and checked-in, or is non-versionable but its nearest versionable ancestor is checked-in. Implementations may differ on when this validation is done. A **LockException** is thrown either immediately or on **save**

	if a lock prevents the addition of the mixin. Implementations may differ on when this validation is done.
	A **RepositoryException** will be thrown if another error occurs.
void	**removeMixin(String mixinName)**
	Removes the specified mixin node type from this node. Also removes **mixinName** from this node's **jcr:mixinTypes** property immediately. Semantically, the mixin node type removal *may* take effect immediately and at the very least, it *must* take effect on **save**.
	If this node does not have the specified mixin, a **NoSuchNodeTypeException** is thrown either immediately or on **save**. Implementations may differ on when this validation is done.
	A **ConstraintViolationException** will be thrown either immediately or on **save** if the removal of a mixin is not allowed. Implementations are free to enforce any policy they like with regard to mixin removal and may differ on when this validation is done.
	A **VersionException** is thrown either immediately or on **save** if **this** node is versionable and checked-in or is non-versionable but its nearest versionable ancestor is checked-in. Implementations may differ on when this validation is done.
	A **LockException** is thrown either immediately or on **save** if a lock prevents the removal of the mixin. Implementations may differ on when this validation is done.
	A **RepositoryException** will be thrown if another error occurs.
boolean	**canAddMixin(String mixinName)**
	Returns true if the specified mixin node type, **mixinName**, can be added to this node. Returns **false** otherwise. A result of **false** must be returned in each of the following cases:
	• The mixin's definition conflicts with an existing primary or mixin node type of this node.
	• This node is versionable and checked-in or is non-versionable and its nearest versionable ancestor is checked-in.

	- This node is *protected* (as defined in this node's **NodeDefinition**, found in the node type of this node's parent). - An access control restriction would prevent the addition of the mixin. - A lock would prevent the addition of the mixin. - An implementation-specific restriction would prevent the addition of the mixin. A **NoSuchNodeTypeException** is thrown if the specified mixin node type name is not recognized. A **RepositoryException** will be thrown if another error occurs.

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 (see 7.3.3 **Respecting Property Semantics**).

7.4.5 Serialization and Node Types

When deserializing content from another content repository, each imported node will come with its attached **jcr:primaryType** and **jcr:mixinTypes** properties. This information may 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 node types referenced by the imported content that are not skipped (see 7.3.3 **Respecting Property Semantics**) 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.

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*).

7.5 Thread-Safety Requirements

A content repository implementation is required to provide a thread-safe implementation of all methods of `javax.jcr.Repository`.

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

8 Optional Repository Features

This section provides an overview of optional features that may be supported by a content repository implementation. These are: Transactions, Versioning, Observation, Locking and SQL Search. 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 compliant content 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).

Whether a particular implementation supports transactions can be determined by querying the repository descriptor table with **`Repository.getDescriptor("OPTION_TRANSACTIONS_SUPPORTED")`** (a return value of **`true`** indicates support for transactions, see 6.1.1.1 **Repository Descriptors**).

The actual methods used to control transaction boundaries are not defined by this 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 application using the API. The JTA interfaces **`javax.transaction.TransactionManager`** and **`javax.transaction.Transaction`** are the relevant ones in this context (though the client, as mentioned, will never have a need to use these).

In the second case, user managed transactions, the application using the API may choose to control transaction boundaries from within the application. In this case the relevant interface is **`javax.transaction.UserTransaction`**. This is the interface that provides the methods **`begin`**, **`commit`**, **`rollback`** and so forth. Note that behind the scenes the **`javax.transaction.TransactionManager`** and **`javax.transaction.Transaction`** are still employed, but again, the client does not deal with these.

A content repository implementation must support both of these approaches.

8.1.1 Container Managed Transactions: Sample Request Flow
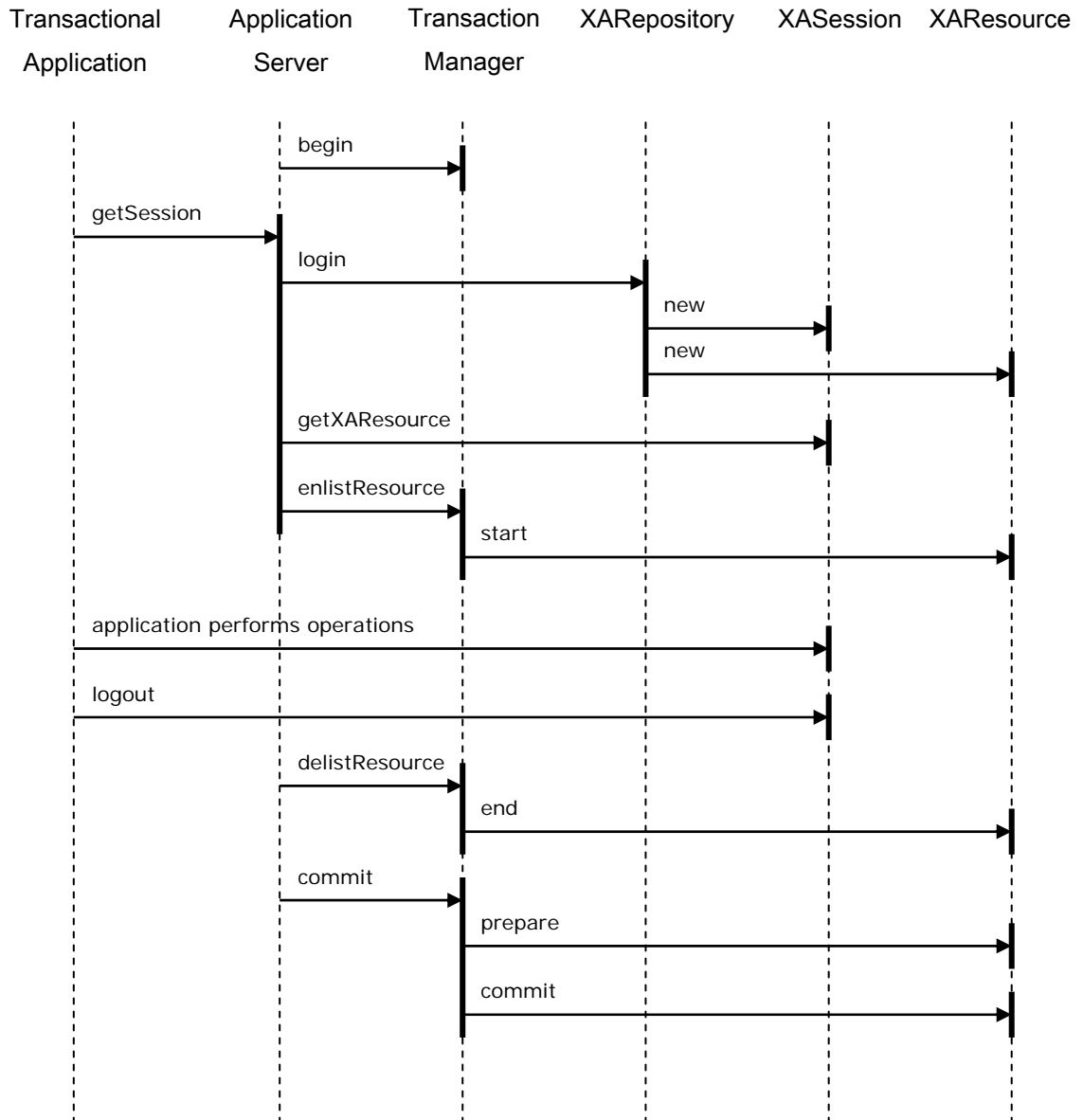
8.1.2 User Managed Transactions: Sample Code

```
// 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.

8.1.4 Single Session Across Multiple Transactions

Because modifications in the transient layer are not transactionalized, the possibility exists for some implementations to allow a `Session` to be shared across transactions. This possibility arises because in JTA, an `XAResource` may be successively associated with different global transactions and in many implementations the natural mapping will be to make the `Session` implement the `XAResource`. The following code snippet illustrates how an `XAResource` may be shared across two global transactions:

```
// Associate the resource (our Session) with a global
// transaction xid1
res.start(xid1, TMNOFLAGS);

// Do something with res, on behalf of xid1
// ...

// Suspend work on this transaction
res.end(xid1, TMSUSPEND);
```

```
// Associate (the same!) resource with another
// global transaction xid2
res.start(xid2, TMNOFLAGS);

// Do something with res, on behalf of xid2
// ...

// End work
res.end(xid2, TMSUCCESS);

// Resume work with former transaction
res.start(xid1, TMRESUME);

// Commit work recorded when associated with xid2
res.commit(xid2, true);
```

In cases where the **XAResource** corresponds to a **Session** (that is, probably most implementations), items that have been obtained in the context of **xid1** would still be valid when the **Session** is effectively associated with **xid2**. In other words, all transactions working on the same **Session** would share the transient items obtained through that **Session**.

In those implementations that adopt a copy-on-read approach to transient storage (see 7.1.3.4 *Seeing Changes Made by Other Sessions*) this will mean that the transient layer reflects an unchanged item's state *in the transaction context in which the item was acquired*. As soon as an item is refreshed or changed, the session will then reflect the state of that item in the transaction context within which that refresh or change took place.

Some implementers may choose to circumvent any problems with shared transient items by undoing changes inside the transient layer when a session is disassociated from a global transaction. This is however, outside the scope of this specification.

8.1.5 Mention of Transactions within this Specification

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 the absence of transactions throughout most of the specification and note the qualification here.

8.2 Versioning

A compliant content 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 versioning system is modelled after the Workspace Versioning and Configuration Management (WVCM) API defined by JSR 147.

Whether a particular implementation supports versioning can be determined by querying the repository descriptor table with **Repository.getDescriptor("OPTION_VERSIONING_SUPPORTED")** (a return value of **true** indicates support for versioning, see 6.1.1.1 **Repository Descriptors**).

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
PropertyDefinition
  Name **jcr:versionHistory**
  RequiredType **REFERENCE**
  ValueConstraints **["nt:versionHistory"]**
  DefaultValues **null**
  AutoCreated **false**
  Mandatory **true**
  OnParentVersion **COPY**
  Protected **true**
  Multiple **false**
PropertyDefinition
  Name **jcr:baseVersion**
  RequiredType **REFERENCE**
  ValueConstraints **["nt:version"]**
  DefaultValues **null**
  AutoCreated **false**
  Mandatory **true**
  OnParentVersion **IGNORE**
  Protected **true**
  Multiple **false**
PropertyDefinition
  Name **jcr:isCheckedOut**
  RequiredType **BOOLEAN**
  ValueConstraints **[]**
  DefaultValues **[true]**
  AutoCreated **true**
  Mandatory **true**
  OnParentVersion **IGNORE**
  Protected **true**
  Multiple **false**
PropertyDefinition
  Name **jcr:predecessors**
  RequiredType **REFERENCE**
  ValueConstraints **["nt:version"]**
  DefaultValues **null**
  AutoCreated **false**
  Mandatory **true**
  OnParentVersion **COPY**
  Protected **true**
  Multiple **true**
PropertyDefinition
  Name **jcr:mergeFailed**
  RequiredType **REFERENCE**
  ValueConstraints **["nt:version"]**
  DefaultValues **null**
  AutoCreated **false**
  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 version) or updates it to reflect the corresponding node (if the corresponding 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 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:
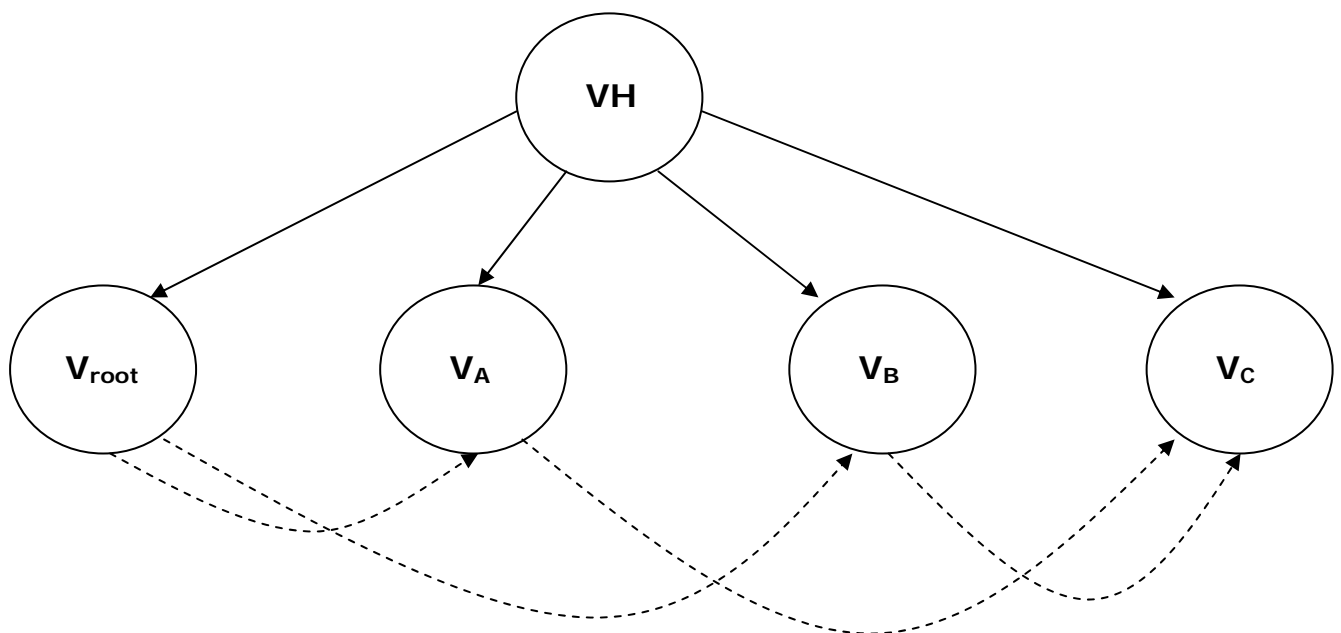
The solid arrows represent parent node to child node relations while the dotted arrows represent the successor relations between versions, implemented through **REFERENCE** properties.

Here we see an **nt:versionHistory** node, **VH** with child **nt:version** nodes V_{root} V_A V_B and V_C. The version graph begins at V_{root}, which has successors V_A and V_B, both of which, in turn, have V_C 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, applications cannot alter this subtree through standard write methods; though the implementation can, of course, alter it as a side-effect of the application 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), 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 left up to the implementation.

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, an application 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 application 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*):

NodeTypeName
  nt:versionHistory
Supertypes
  nt:base
  mix:referenceable
IsMixin
  false
HasOrderableChildNodes
  false
PrimaryItemName
  null
PropertyDefinition
  Name **jcr:versionableUuid**
  RequiredType **STRING**
  ValueConstraints **[]**
  DefaultValues **null**
  AutoCreated **true**
  Mandatory **true**
  OnParentVersion **ABORT**
  Protected **true**
  Multiple **false**
ChildNodeDefinition
  Name **jcr:rootVersion**
  RequiredPrimaryTypes **[nt:version]**
  DefaultPrimaryType **nt:version**
  AutoCreated **true**
  Mandatory **true**
  OnParentVersion **ABORT**
  Protected **true**
  SameNameSiblings **false**
ChildNodeDefinition
  Name **jcr:versionLabels**
  RequiredPrimaryTypes **[nt:versionLabels]**
  DefaultPrimaryType **nt:versionLabels**
  AutoCreated **true**
  Mandatory **true**
  OnParentVersion **ABORT**

```
  Protected true
  SameNameSiblings false
ChildNodeDefinition
  Name *
  RequiredPrimaryTypes [nt:version]
  DefaultPrimaryType nt:version
  AutoCreated false
  Mandatory false
  OnParentVersion ABORT
  Protected true
  SameNameSiblings 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.

This node 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 *ChildNodeDefinition*, 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

The **nt:versionLabels** node type has the following definition (repeated from 6.7.22.14 *nt:versionLabels*):

```
NodeTypeName
  nt:versionLabels
Supertypes
  nt:base
IsMixin
  false
HasOrderableChildNodes
```

```
     false
PrimaryItemName
   null
PropertyDefinition
   Name *
   RequiredType REFERENCE
   ValueConstraints ["nt:version"]
   DefaultValues null
   AutoCreated 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
PropertyDefinition
   Name jcr:created
   RequiredType DATE
   ValueConstraints []
   DefaultValues null
   AutoCreated true
   Mandatory true
   OnParentVersion ABORT
   Protected true
   Multiple false
PropertyDefinition
   Name jcr:predecessors
   RequiredType REFERENCE
   ValueConstraints ["nt:version"]
   DefaultValues null
   AutoCreated false
   Mandatory false
   OnParentVersion ABORT
   Protected true
   Multiple true
PropertyDefinition
   Name jcr:successors
   RequiredType REFERENCE
   ValueConstraints ["nt:version"]
   DefaultValues null
   AutoCreated false
   Mandatory false
   OnParentVersion ABORT
   Protected true
   Multiple true
ChildNodeDefinition
```

```
Name jcr:frozenNode
RequiredPrimaryTypes [nt:frozenNode]
DefaultPrimaryType null
AutoCreated false
Mandatory false
OnParentVersion ABORT
Protected true
SameNameSiblings 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 inherits 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

The **nt:frozenNode** node type has the following definition (repeated from 6.7.22.16 *nt:frozenNode*):

```
NodeTypeName
  nt:frozenNode
Supertypes
  nt:base
  mix:referenceable
IsMixin
  false
HasOrderableChildNodes
  true
PrimaryItemName
  null
PropertyDefinition
  Name jcr:frozenPrimaryType
```

```
        RequiredType NAME
        ValueConstraints []
        DefaultValues null
        AutoCreated true
        Mandatory true
        OnParentVersion ABORT
        Protected true
        Multiple false
PropertyDefinition
        Name jcr:frozenMixinTypes
        RequiredType NAME
        ValueConstraints []
        DefaultValues null
        AutoCreated false
        Mandatory false
        OnParentVersion ABORT
        Protected true
        Multiple true
PropertyDefinition
        Name jcr:frozenUuid
        RequiredType STRING
        ValueConstraints []
        DefaultValues null
        AutoCreated true
        Mandatory true
        OnParentVersion ABORT
        Protected true
        Multiple false
PropertyDefinition
        Name *
        RequiredType UNDEFINED
        ValueConstraints []
        DefaultValues null
        AutoCreated false
        Mandatory false
        OnParentVersion ABORT
        Protected true
        Multiple false
PropertyDefinition
        Name *
        RequiredType UNDEFINED
        ValueConstraints []
        DefaultValues null
        AutoCreated false
        Mandatory false
        OnParentVersion ABORT
        Protected true
        Multiple true
ChildNodeDefinition
        Name *
        RequiredPrimaryTypes [nt:base]
        DefaultPrimaryType null
        AutoCreated false
        Mandatory false
        OnParentVersion ABORT
        Protected true
        SameNameSiblings 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

The **nt:versionedChild** node type has the following definition (repeated from 6.7.22.17 *nt:frozenNode*):

```
NodeTypeName
  nt:versionedChild
Supertypes
  nt:base
IsMixin
  false
HasOrderableChildNodes
  false
PrimaryItemName
  null
PropertyDefinition
  Name jcr:childVersionHistory
  RequiredType REFERENCE
  ValueConstraints ["nt:versionHistory"]
  DefaultValues null
  AutoCreated 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 or predecessors. 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.

Assuming 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 `W`$_1$:

- `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** (call it V_0).

- V_0 is the root version of **VH**. It is a *dummy version* that serves as the starting point of the version graph. Like all version nodes, it has a subnode called **jcr:frozenNode**. But, in this case that frozen node does not contain any state information about **N** (other than the node type and UUID information found in the properties **jcr:frozenPrimaryType**, **jcr:frozenMixinTypes**, and **jcr:frozenUuid**).

- 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 V_0. This constitutes a reference from **N** to its current base version.

- The multi-value **REFERENCE** property **jcr:predecessors** of **N** is initialized to contain a single UUID, that of V_0 (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 application 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**[11], **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

---

[11] **Workspace.restore**, **Node.restore** (all signatures) and **Node.restoreByLabel**.

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**.

- **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 application calls **N.restore("x.y", removeExisting)** [12]. Assuming that the version node representing the version named "**x.y**" is node **V**, then 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

---

[12] 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.

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/C** and **Y** refers to **/A/B**. 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.

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 replaces **`this`** node and its subtree with a clone of the its corresponding node and its subtree 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, 7.1.8 *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 **`N`** in the subtree rooted at **`this`** node, a *merge test* is performed comparing **`N`** with its corresponding node in **`srcWorkspace`**, **`N'`**.

- The merge test is done by comparing *the base version of* **`N`** (call it **`v`**) and *the base version of* **`N'`** (call it **`v'`**).

- For any versionable node **`N`** there are three possible outcomes of the merge test: *update*, *leave* or *failed*.

- If **`N`** does not have a corresponding node then the merge result for **`N`** is *leave*.

- If **`N`** is currently checked-in then:

    - If **`v'`** is a successor (to any degree) of **`v`**, then the merge result for **`N`** is *update*.

    - If **`v'`** is a predecessor (to any degree) of **`v`** or if **`v`** and **`v'`** are identical (i.e., are actually the same version), then the merge result for **`N`** is *leave*.

    - If **`v`** is neither a successor of, predecessor of, nor identical with **`v'`**, then the merge result for **`N`** is *failed*. This is the case where **`N`** and **`N'`** represent divergent

branches of the version graph, thus determining the result of a merge is non-trivial.

- If **N** is currently checked-out then:

  - If **v'** is a predecessor (to any degree) of **v** or if **v** and **v'** are identical (i.e., are actually the same version), then the merge result for **N** is *leave*.

  - If any other relationship holds between **v** and **v'**, then the merge result for **N** is *fail*.

- If **bestEffort** is **false** then the first time a merge result of *fail* occurs, the entire merge operation on this 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 **N** with result *update* is updated to reflect the state of **N'**. The state of a node in this context refers to its set of properties and child node links.

  - Each versionable node **N** with result *leave* is left unchanged, *unless **N** is the child of a node with status* update *and **N** does not have a corresponding node in* **srcWorkspace**, *in which case it is removed.*

- If **bestEffort** is **true** then:

  - Each versionable node **N** with result *update* is updated to reflect the state of **N'**. The state of a node in this context refers to its set of properties and child node links.

  - Each versionable node **N** with result *leave* is left unchanged, *unless **N** is the child of a node with status* update *and **N** does not have a corresponding node in* **srcWorkspace**, *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**.

- o 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 8.2.10.2 *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**.

The merge method returns a **NodeIterator** over all versionable nodes in the subtree that received a merge result of *fail*.

Note that if **bestEffort** is **false**, then **merge** will either return an empty iterator (since no merge failure occurred) or throw a **MergeException** (on the first merge failure that was encountered).

If **bestEffort** is **true**, then the iterator will contain all nodes that received a *fail* during the course of this **merge** operation.

8.2.10.1 Merge Algorithm

The above declarative description can also be expressed in pseudo-code as follows:

let **ws'** be the workspace against which the **merge** is done.
let **bestEffort** be the flag passed to **merge**.
let **failedset** be a set of UUIDs, initially empty.
let **startnode** be the node on which **merge** was called.
domerge(startnode).
return the nodes with the UUIDs in **failedset**.

domerge(n)
    let **n'** be the corresponding node of **n** in **ws'**.
    if no such **n' doleave(n)**.

else if **n** is not versionable **doupdate(n, n')**.

else if **n'** is not versionable **doleave(n)**.

let **v** be base version of **n**.

let **v'** be base version of **n'**.

if **v'** is a successor of **v** and

   n is not checked-in **doupdate(n, n')**.

else if **v** is equal to or a predecessor of **v' doleave(n)**.

else **dofail(n, v')**.

dofail(n, v')

if **bestEffort** = **false** throw **MergeException**.

else add UUID of **v'** (if not already present) to the

   jcr:mergeFailed property of **n**,

   add UUID of **n** to **failedset**,

   doleave(n).

doleave(n)

 for each child node **c** of **n** **domerge(c)**.

doupdate(n, n')

replace set of properties of **n** with those of **n'**.

let **S** be the set of child nodes of **n**.

let **S'** be the set of child nodes of **n'**.

judging by the name of the child node:

   let **C** be the set of nodes in **S** and in **S'**

   let **D** be the set of nodes in **S** but not in **S'**.

   let **D'** be the set of nodes in **S'** but not in **S**.

remove from **n** all child nodes in **D**.

for each child node of **n'** in **D'** copy it (and its subtree) to **n**

   as a new child node (if an incoming node has the same

   UUID as a node already existing in this workspace,

   the already existing node is removed).

for each child node **m** of **n** in **C** **domerge(m)**.

8.2.10.2 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 application 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 application chooses not to join the two branches, then `cancelMerge(V')` is performed. This has the effect of removing the reference to `V'` from the `jcr:mergeFailed` property of `N` without adding it to `jcr:predecessors`.

Once the last reference in `jcr:mergeFailed` has been either moved to `jcr:predecessors` (with `doneMerge`) or just removed from `jcr:mergeFailed` (with `cancelMerge`) the `jcr:mergeFailed` property is automatically removed, thus enabling this node to be checked-in, creating a new version (note that before the `jcr:mergeFailed` is removed, its `OnParentVersion` setting of `ABORT` prevents check in). This new version will have a 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 `PropertyDefinition` or `NodeDefinition` 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 `PropertyDefinition` for `P` and the `NodeDefinition` 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 **COPY** applies on **checkin**, however the recursive copy terminates at each versionable node encountered further below in the subtree, at which points the standard **VERSION** behavior is again followed.

```
NodeTypeName
  nt:versionedChild
Supertypes
  nt:base
IsMixin
  false
HasOrderableChildNodes
  false
PrimaryItemName
  null
PropertyDefinition
  Name jcr:childVersionHistory
  RequiredType REFERENCE
  ValueConstraints ["nt:versionHistory"]
  DefaultValues null
  AutoCreated 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, this API 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 **jcr:isCheckedOut** property of this node is set to **false** thus putting the node into the *checked-in* state. This means that this node and its *connected non-versionable subtree* 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 (**addNode**, **setProperty**, etc.). The only exceptions to this rule are the **restore**[13], **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).

---

[13] **Workspace.restore**, **Node.restore** (all signatures) and **Node.restoreByLabel**.

	See 8.2.5 *Check In* for more details.
	If this node is already checked-in, this method has no effect but returns the current base version of this node.
	If this node is not versionable, an `UnsupportedRepositoryOperationException` is thrown.
	A `VersionException` is thrown or if a child item of this node has an `OnParentVersion` status of `ABORT`. This includes the case where an unresolved merge failure exists on this node, as indicated by the presence of the `jcr:mergeFailed` property.
	If `checkin` succeeds, the change to the `jcr:checkedOut` property is automatically persisted (there is no need to do an additional `save`).
	If there are unsaved changes pending on this node, an `InvalidItemStateException` is thrown.
	A `LockException` is thrown if a lock prevents the checkin.
	A `RepositoryException` is thrown if an error occurs.
`void`	`checkout()` Sets this versionable node to checked-out status by setting its `jcr:isCheckedOut` property to `true`, sets the `jcr:predecessors` property to be a reference to the current base version (the same value as held in `jcr:baseVersion`). This method puts the node into the *checked-out* state, making it and its connected non-versionable subtree no longer read-only (see `checkin`, above, for an explanation of the term "connected non-versionable subtree"). If successful, these changes are persisted immediately, there is no need to call `save`. See 8.2.6 *Check Out* for more details. If this node is not versionable, an `UnsupportedRepositoryOperationException` is thrown. A `LockException` is thrown if a lock prevents the checkout. A `RepositoryException` is thrown if an error occurs.
`NodeIterator`	`merge(String srcWorkspace, boolean bestEffort)`

	This method can be thought of as a version-sensitive update (see 7.1.8 *Updating and Cloning Nodes across Workspaces*). 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*. A **MergeException** 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. If successful, the changes are persisted immediately, there is no need to call **save**. This method returns a **NodeIterator** over all versionable nodes in the subtree that received a merge result of *fail*. If **bestEffort** is **false**, this iterator will be empty (since if it **merge** returns successfully, instead of throwing an exception, it will be because no failures were encountered). If **bestEffort** is **true**, this iterator will contain all nodes that received a *fail* during the course of this **merge** operation. If the specified **srcWorkspace** does not exist, a **NoSuchWorkspaceException** is thrown. If the current session does not have sufficient permissions to perform the operation, then an **AccessDeniedException** is thrown. An **InvalidItemStateException** is thrown if this **Session** (not necessarily this **Node**) has pending unsaved changes. A **LockException** is thrown if a lock prevents the merge. A **RepositoryException** is thrown if another error occurs.
void	**doneMerge(Version version)** Completes the merge process with respect to this node and the specified **version**. See 8.2.10 *Merge* for more details. If successful, the changes are persisted immediately,

	there is no need to call **save**.
	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.
	An **UnsupportedRepositoryOperationException** is thrown if this node is not versionable.
	If there are unsaved changes pending on this node, an **InvalidItemStateException** is thrown.
	A **LockException** is thrown if a lock prevents the operation.
	A **RepositoryException** is thrown if another error occurs.
void	**cancelMerge(Version version)**
	Cancels the merge process with respect to this node and the specified **version**.
	See 8.2.10 *Merge* for more details.
	If successful, the changes are persisted immediately, there is no need to call **save**.
	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.
	An **UnsupportedRepositoryOperationException** is thrown if this node is not versionable.
	If there are unsaved changes pending on this node, an **InvalidItemStateException** is thrown.
	A **LockException** is thrown if a lock prevents the operation.
	A **RepositoryException** is thrown if another error occurs.
boolean	**isCheckedOut()**
	Returns **true** if this node is either
	• versionable and currently checked-out,
	• non-versionable and its nearest versionable ancestor is checked-out or
	• non-versionable and it has no versionable

	ancestor.
	Returns **false** if this node is either
	- versionable and currently checked-in or - non-versionable and its nearest versionable ancestor is checked-in.
	A **RepositoryException** is thrown if an error occurs.
void	**restore(String versionName,** **boolean removeExisting)** Restores this node to the state defined by the version with the specified **versionName**. If this node is not versionable, an **UnsupportedRepositoryOperationException** is thrown. If successful, the change is persisted immediately and there is no need to call **save**. A **VersionException** is thrown if no version with the specified **versionName** exists in this node's version history or if an attempt is made to restore the root version (**jcr:rootVersion**). An **InvalidItemStateException** is thrown if this **Session** (not necessarily this **Node**) has pending unsaved changes. This method will work regardless of whether this node is checked-in or not. A UUID collision occurs when a node exists *outside the subtree rooted at this node* with the same UUID as a node that would be introduced by the **restore** operation *into the subtree at this node*. 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 *any* 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.
	A **LockException** is thrown if a lock prevents the restore.
	A **RepositoryException** is thrown if another error occurs.
`void`	**restore(Version version,** **boolean removeExisting)** Restores this node to the state defined by the specified **version**. If this node is not versionable, an **UnsupportedRepositoryOperationException** is thrown. If successful, the change is persisted immediately and there is no need to call **save**. A **VersionException** is thrown if the specified **version** is not part of this node's version history. An **InvalidItemStateException** is thrown if this **Session** (not necessarily this **Node**) has pending unsaved changes. This method will work regardless of whether this node is checked-in or not. A UUID collision occurs when a node exists *outside the subtree rooted at this node* with the same UUID as a node that would be introduced by the **restore** operation *into the subtree at this node*. 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 *any* 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. A **LockException** is thrown if a lock prevents the restore. A **RepositoryException** is thrown if another error

void	restore(Version version, String relPath, boolean removeExisting)
	occurs.

	restore(Version version,
	         String relPath,
	         boolean removeExisting)
	
	Restores the specified **version** to **relPath**, relative to this node.
	
	A node need not exist at **relPath**, though the parent of **relPath** must exist, otherwise a **PathNotFoundException** is thrown.
	
	If a node *does* exist at **relPath** then it must correspond to the version being restored (the version must be a version *of that node*), otherwise a **VersionException** is thrown.
	
	If no node exists at **relPath** then a **VersionException** is thrown if the parent node of **relPath** is versionable and checked-in or is non-versionable but its nearest versionable ancestor is checked-in.
	
	If there is a node at **relPath** 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.
	
	A UUID collision occurs when a node exists *outside the subtree rooted at **relPath*** with the same UUID as a node that would be introduced by the **restore** operation *into the subtree at **relPath*** (note that in cases where there is no node at **relPath**, this amounts to saying that a UUID collision occurs if there exists a node *anywhere* in this workspace with the same UUID as a node that would be introduced by the **restore**). 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 *any* 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.
	
	If the would-be parent of the location **relPath** is

275

	actually a property, or if a node type restriction would be violated, then a **ConstraintViolationException** is thrown.
	If the **restore** succeeds, the changes made to this node are persisted immediately, there is no need to call **save**.
	An **InvalidItemStateException** is thrown if this **Session** (not necessarily this **Node**) has pending unsaved changes.
	An **UnsupportedRepositoryOperationException** is thrown if versioning is not supported.
	A **LockException** is thrown if a lock prevents the restore.
	A **RepositoryException** is thrown if another error occurs.
void	**restoreByLabel(String versionLabel,** **boolean removeExisting)** Restores this node to the state defined by the version with the specified **versionLabel**. If this node is not versionable, an **UnsupportedRepositoryOperationException** is thrown. If successful, the change is persisted immediately and there is no need to call **save**. A **VersionException** is thrown if no version with the specified **versionLabel** exists in this node's version history. This method will work regardless of whether this node is checked-in or not. A UUID collision occurs when a node exists *outside the subtree rooted at this node* with the same UUID as a node that would be introduced by the **restore** operation *into the subtree at this node.* 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 *any* 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. An **InvalidItemStateException** is thrown if this **Session** (not necessarily this **Node**) has pending unsaved changes. A **LockException** is thrown if a lock prevents the restore. A **RepositoryException** is thrown if another error occurs.
VersionHistory	**getVersionHistory()** Returns the **VersionHistory** object of this node. This object provides access to the **nt:versionHistory** node holding this node's versions. If **this** node is not versionable, an **UnsupportedRepositoryOperationException** is thrown. A **RepositoryException** is thrown if another error occurs.
Version	**getBaseVersion()** Returns the current base version of this versionable node. If this node is not versionable, an **UnsupportedRepositoryOperationException** is thrown. A **RepositoryException** is thrown if another error occurs.

8.2.14.2 Workspace Versioning Methods

The **Workspace** object provides the "group restore" method.

`javax.jcr.` **Workspace**	
void	**restore(Version[] versions,** **boolean removeExisting)** Restores a set of versions at once. Used in cases where a "chicken and egg" problem of mutually referring **REFERENCE** properties would prevent the

restore in any serial order.

If the restore succeeds, the changes made are persisted immediately, there is no need to call **save**.

This method will work regardless of whether the nodes corresponding to the specified versions are checked-in or not.

The following restrictions apply to the set of versions specified:

If **s** is the set of versions being restored simultaneously,

- For every version **v** in **s** that corresponds to a *missing* node in the workspace, there must also be a parent of **v** in **s**.

- **s** must contain at least one version that corresponds to an *existing* node in the workspace.

- No **v** in **s** can be a root version (**jcr:rootVersion**).

If 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.

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 *outside these subtrees* 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 *any* 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.
	An **UnsupportedRepositoryOperationException** is thrown if versioning is not supported.
	A **LockException** is thrown if a lock prevents the restore.
	An **InvalidItemStateException** is thrown if this **Session** (not necessarily this **Node**) has pending unsaved changes.
	Throws a **RepositoryException** if another error occurs.

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	**getVersionByLabel(String label)**
	Retrieves a particular version from this version history by version label.
	Throws a **VersionException** if the specified **label** is not in this version history.
	A **RepositoryException** is thrown if an error occurs.
void	**addVersionLabel(String versionName,** **String label,** **boolean moveLabel)**
	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.
	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.
	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**.
	A **VersionException** is also thrown if the named version is not in this **VersionHistory** or if it is the root version (**jcr:rootVersion**).
	A **RepositoryException** is thrown if another error occurs.
boolean	**hasVersionLabel(String label)**
	Returns **true** if any version in the history has the given **label**.
	A **RepositoryException** is thrown if an error occurs.

boolean	**hasVersionLabel(Version version, String label)**
	Returns **true** if the given **version** has the given **label**.
	A **RepositoryException** is thrown if an error occurs.
String[]	**getVersionLabels()**
	Returns all version labels of the history or an empty array if there are none.
	A **RepositoryException** is thrown if an error occurs.
String[]	**getVersionLabels(Version version)**
	Returns all version labels of the given **version** - empty array if none.
	Throws a **VersionException** if the specified **version** is not in this version history.
	Throws a **RepositoryException** if another error occurs.
Void	**removeVersionLabel(String label)**
	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.
	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.
	If a **label** is specified that does not exist in this version history, a **VersionException** is thrown.
	A **RepositoryException** is thrown if another error occurs.
Void	**removeVersion(String versionName)**
	Removes the named version from this version history and automatically repairs the version graph. If the version to be removed is **v**, **v**'s predecessor set is **P** and **v**'s successor set is **S**, then the version graph is repaired s follows:
	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**.
	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.
	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** has read access to that **REFERENCE** property.
	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** *does not* have read access to that **REFERENCE** property.
	Throws an **UnsupportedRepositoryOperationException** if this operation is not supported by the implementation.
	Throws a **VersionException** if the named version is not in this **VersionHistory**.
	Throws a **RepositoryException** if another error occurs.

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()**
	Returns the predecessor versions of this version. This corresponds to returning all the **nt:version** nodes whose **jcr:successors** property includes a reference to the **nt:version** node that represents this version. A **RepositoryException** is thrown if an error occurs.

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:system/jcr:versionStorage**. The special status of these nodes with respect to versioning is transparent to the serialization mechanism.

The serialized content of the source version storage can be 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 compliant content 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.

Whether a particular implementation supports observation can be determined by querying the repository descriptor table with `Repository.getDescriptor("OPTION_OBSERVATION_SUPPORTED")` (a return value of `true` indicates support for observation, see 6.1.1.1 **Repository Descriptors**).

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: • 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.

Note that it is up to the implementation whether changes made to the subtree below `jcr:system` trigger events (6.8 **System Node**).

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

285

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	**addEventListener(EventListener listener,**
	                              int eventTypes,
	                              String absPath,
	                              boolean isDeep,
	                              String[] uuid,
	                              String[] nodeTypeName,
	                              boolean noLocal)
	
	 Adds an event listener that listens for the specified **eventTypes** (a combination of one or more event types encoded as a bit mask value).
	
	 The set of events can be filtered by specifying restrictions based on characteristics of the *associated parent node* of the event. The associated parent node of an event is the parent node of the item at (or formerly at) the path

	returned by **Event.getPath**. The following restrictions are available:
	• **absPath, isDeep**: Only events whose associated parent node is at **absPath** (or within its subtree, if **isDeep** is **true**) 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 **null** then no UUID-related restriction is placed on events received. Note that specifying an empty array instead of **null** 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 **null** then no node type-related restriction is placed on events received. Note that specifying an empty array instead of **null** would result in no node types being listened to.
	The restrictions are "ANDed" together. In other words, for a particular node to be listened to, it must meet *all* the restrictions.
	Additionally, if **noLocal** is **true**, then events generated by the session through which the listener was registered are ignored. Otherwise, they are not ignored.
	The filters of an already-registered **EventListener** can be changed at runtime by re-registering the same **EventListener** 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.
	A **RepositoryException** is thrown if an error occurs.
void	**removeEventListener(EventListener listener)**
	Deregisters an event listener.
	A **RepositoryException** is thrown if an error

	occurs.
`EventListenerIterator`	`getRegisteredEventListeners()`
	Returns all event listeners that have been registered through this session. If no listeners have been registered, an empty iterator is returned.
	A `RepositoryException` is thrown if an error occurs.

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:

  o 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.

  o 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`.

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 *must* be generated for the node on which the `remove` was called. Additionally, an implementation *should* also generate a `NODE_REMOVE` or `PROPERTY_REMOVE` (as appropriate) for each item in the removed subtree.

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, an implementation *must* generate a single **NODE_ADDED** event reflecting the addition of the root of the copied subtree at the destination location. Additionally, an implementation *should* generate appropriate events for each resulting node and property addition in the copied subtree.

8.3.7.7 Moving a Subtree

When a subtree is moved an implementation *must* generate a **NODE_REMOVED** for the removal of the root of the moved subtree from the source location and a **NODE_ADDED** for its addition at the destination location. Additionally, an implementation *should* generate a **NODE_REMOVE** or **PROPERTY_REMOVE** (as appropriate) for each node and property removed from its source path location and a **NODE_ADDED** or **PROPERTY_ADDED** (as appropriate) for each node and property added at its destination path location.

8.3.7.8 Re-ordering a set of Child Nodes

When an **orderBefore(A, B)** is performed, an implementation *must* generate a **NODE_REMOVED** for node **A** and a **NODE_ADDED** for node **A**. Note that the paths associated with these two events will either differ by the last index number (if the movement of A causes it to be re-ordered with respect to its same-name siblings) or be identical (if **A** does not have same-name siblings or if the movement of **A** does not change its order relative to its same-name siblings). Additionally, an implementation *should* generate appropriate events reflecting the "shifting over" of the node **B** and any nodes that come after it in the child node ordering. Each such shifted node would also produce a **NODE_REMOVED** and **NODE_ADDED** event pair with paths differing at most by a final index.

8.3.7.9 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**.

8.3.7.10 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**.

8.3.7.11 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).

8.3.7.12 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.

8.3.7.13 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 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 a compliant content repository will already prevent 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.

8.4.1 Discovery of Lock Capabilities

Whether a particular implementation supports locking can be determined by querying the repository descriptor table with **Repository.getDescriptor("OPTION_LOCKING_SUPPORTED")** (a return value of **true** indicates support for locking, see 6.1.1.1 **Repository Descriptors**).

8.4.2 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** (inherited as part of their primary node type or explicitly assigned) may hold locks. The definition of **mix:lockable** is:

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

```
            Protected  true
            Multiple   false
```

8.4.3 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.

Additionally, assuming a deep lock exists above a `mix:lockable` node any attempt to lock this lower level `mix:lockable` node will also fail, because it is already locked from above.

8.4.4 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 (that is, 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).

8.4.5 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 content repository 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 administrational 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**.

8.4.6 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**.

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.

8.4.7 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. Additionally, since a

session-scoped lock is always tied to the session that created it, it does not make sense to move the token of session-scoped lock from its originating session to some other session. Consequently, `Session.removeLockToken(token)` will always fail when `token` specifies a session-scoped lock (see 8.4.13 *Session Methods Related to the Lock Token*).

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 `Session.addLockToken`, `Session.removeLockToken` and `Session.getLockTokens` are provided. (see 8.4.13 *Session Methods Related to the Lock Token*).

To determine an existing lock's scoping, the method `Lock.isSessionScoped()` is provided.

8.4.8 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 (that is, serve as the source of a copy) by any user with sufficient access permissions.

When an action is prevented due to a lock, a `LockException` is thrown either immediately or on the subsequent `save`. Implementations may differ on which of these behaviors is used to enforce locking.

There is at most one lock on any node at one time.

8.4.9 Timing Out

An implementation may unlock any lock at any time due to implementation-specific criteria, such as time limits on locks.

8.4.10 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 `Session`s) upon commit of the transaction, and this includes changes in the locked status of a node. As a result, if a lock is enabled and then disabled within the same transaction, its effect never makes it to the persistent workspace and therefore it does nothing.

In order to use locks properly (that is, to prevent the "lost update problem"), locking and unlocking must be done in separate transactions. For example:

begin
   lock
commit

```
begin
    do A
    save
    do B
    save
    unlock
commit
```

This series of operations would ensure that the actions *A* and *B* are protected by the lock.

8.4.11 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	**lock(boolean isDeep, boolean isSessionScoped)** Places a lock on this node. If successful, this node is said to *hold* the lock. If **isDeep** is **true** then the lock *applies* to this node and all its descendant nodes; if **false**, the lock applies only to this, the holding node. 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. Returns a **Lock** object reflecting the state of the new lock and including a lock token. The lock token is also automatically added to the set of lock tokens held by the current **Session**. 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**. 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. If this node is not of mixin node type **mix:lockable** then a

	LockException is thrown. If this node is already locked (either because it holds a lock or a lock above it applies to it), a **LockException** is thrown. If **isDeep** is **true** and a descendant node of this node already holds a lock, then a **LockException** is thrown. If this node does not have a persistent state (has never been saved), a **LockException** is thrown. If the current session does not have sufficient permissions to place the lock, an **AccessDeniedException** is thrown. An **InvalidItemStateException** is thrown if this node has pending unsaved changes. An **UnsupportedRepositoryOperationException** is thrown if this implementation does not support locking. A **RepositoryException** is thrown if another error occurs.
Lock	**getLock()** 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. 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. If this node is not locked (no lock applies to this node) then a **LockException** is thrown. If the current session does not have sufficient permissions to get the lock, an **AccessDeniedException** is thrown. An **UnsupportedRepositoryOperationException** is thrown if this implementation does not support locking. A **RepositoryException** is thrown if another error occurs.
void	**unlock()** 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**. If this node 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 administrational clean-up of orphaned open-scoped locks. 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. If the current session does not have sufficient permissions to remove the lock, an **AccessDeniedException** is thrown. An **InvalidItemStateException** is thrown if this node has pending unsaved changes. An **UnsupportedRepositoryOperationException** is thrown if this implementation does not support locking. A **RepositoryException** is thrown if another error occurs.
boolean	**holdsLock()** Returns **true** if this node holds a lock; otherwise returns **false**. To *hold* a lock means that this node has actually had a lock placed on it specifically, as opposed to just having a lock *apply* to it due to a deep lock held by an node above. A **RepositoryException** is thrown if an error occurs.
boolean	**isLocked()** Returns **true** 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 **false**. A **RepositoryException** is thrown if an error occurs.

8.4.12 The Lock Object

The **Lock** object represents a lock on a particular node:

```
javax.jcr.lock.
Lock
```

`String`	`getLockOwner()` Returns the user ID of the user who owns this lock. This is the value of the `jcr:lockOwner` property of the lock-holding node. It is also the value returned by `Session.getUserID` 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 `unlock` or make changes to the locked nodes; that depends entirely upon who the token holder is.
`boolean`	`isDeep()` Returns `true` if this is a deep lock; `false` otherwise.
`Node`	`getNode()` Returns the lock holding node. Note that `N.getLock().getNode()` (where `N` is a locked node) will only return `N` if `N` is the lock holder. If `N` is in the subtree of the lock holder, `H`, then this call will return `H`.
`String`	`getLockToken()` 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`.
`boolean`	`isLive()` Returns `true` if this `Lock` object represents a lock that is currently in effect. If this lock has been unlocked either explicitly or due to an implementation-specific limitation (like a timeout) then it returns `false`. Note that this method is intended for those cases where one is holding a `Lock` Java object and wants to find out whether the lock (the repository-level entity that is attached to the lockable node) that this object originally represented still exists. For example, a timeout or explicit `unlock` will remove a lock from a node but the `Lock` Java object corresponding to that lock may still exist, and in that case its `isLive` method will return `false`. A `RepositoryException` is thrown if an error occurs.
`boolean`	`isSessionScoped()` Returns `true` if this is a session-scoped lock. Returns `false` if this is an open-scoped lock.
`void`	`refresh()` If this lock's time-to-live is governed by a timer, this

	method resets that timer so that the lock does not timeout and expire. If this lock's time-to-live is not governed by a timer, then this method has no effect.
	A **LockException** is thrown if this **Session** does not hold the correct lock token for this lock.
	A **RepositoryException** is thrown if another error occurs.

8.4.13 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 **Session** object of the lock owner to alter nodes that are locked by the lock specified by that particular lock token. A **LockException** is thrown if the specified lock token is already held by another **Session**. A **RepositoryException** is thrown if another error occurs.
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. A **LockException** is thrown if the lock associated with the specified lock token is session-scoped. A **RepositoryException** is thrown if another error occurs.

8.5 Searching Repository Content with SQL

A repository (either level 1 or level 2) may support search using the SQL query syntax.

Whether a particular implementation supports SQL can be determined by querying the repository descriptor table with **Repository.getDescriptor("OPTION_QUERY_SQL_SUPPORTED")** (a return value of **true** indicates support for SQL, see 6.1.1.1 **Repository Descriptors**).

8.5.1 The SQL Language

If supported, the SQL language syntax is invoked by specifying the constant **Query.SQL** in **QueryManager.createQuery** (see 6.6.8 Query API):

javax.jcr.query. **Query**	
String	**SQL** A string constant representing the SQL query language applied to the *database view* of the workspace.

8.5.2 Database View

SQL queries can be thought of as working against a *database view* of the workspace being searched (similar in principle to the two XML views described in 6.4 *XML Mappings*).

Note however, that it is entirely up to the implementation whether this database view directly reflects the underlying storage mechanisms of the repository (as it might in repositories that are actually database-backed). All that is required is that if SQL queries are supported, they behave as if they were running against the database view described below.

8.5.2.1 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.

8.5.2.2 Pseudo-property *jcr:path*

- A special column, **jcr:path**, that does not correspond to any actual property is present in node type tables. The **jcr:path** column holds the normalized absolute path for the node represented by each row.

- The **jcr:path** column always appears in the result table

- Note that the actual value of a particular **jcr:path** column within a particular **Row** of the result table can be calculated by the implementation at the time that a request is made for that value. For example, on the call **someRow.getValue("jcr:path")** (where **someRow** is an instance of **Row**). This type of "lazy loading" allows implementations to avoid calculating paths for all nodes in the return set at query time. Such an approach would be advantageous for those implementations in which path calculation is expensive.

- The **jcr:path** value 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]'
    ```

    Consequently, 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 as a match for a whole path segment (the part between two **/** characters) or within index brackets. This set of minimum requirements would, for example, allow the following path queries:

- o Exact path:
  `jcr:path='/books/mybooks/EffectiveJava'`

- o Child:
  `jcr:path LIKE '/books/%' AND NOT jcr:path LIKE '/books/%/%'`

- o Descendant:
  `jcr:path LIKE '/books/mybooks/%'`

- o Descendant or self:
  `jcr:path LIKE '/books/mybooks/%' OR jcr:path='/books/mybooks'`

- o Index test:
  `jcr:path LIKE '/books[%]/mybooks[%]'`

- See also 6.6.3.3 **Property Constraint** and 6.6.3.4 **Path Constraint**.

8.5.2.3 Path Literals

- Any path literals mentioned within a `WHERE` clause must be normalized (no ".." and "." or trailing "/").

8.5.2.4 Pseudo-property *jcr:score*

- Another special column, `jcr:score`, that does not correspond to any actual property, must also be present in the result table.

- The `jcr:score` can be used to return a relevance value for each row. The calculation of this value is not defined. It is not required to always be meaningful. If it is meaningful then it may, for example, be associated with the result of a `CONTAINS()` function (see 8.5.4.5 *CONTAINS*). Alternatively it may not be connected to full text search at all, and may reflect a relevance value derived according to some other criteria.

- Support for comparing `jcr:score` in the `WHERE` clause is not required.

- In some implementations the label for this column may not be literally "`jcr:score`" but instead be the function name "`jcr:score(...)`".

- See also 6.6.3.3 **Property Constraint** and 6.6.5.2 **jcr:contains Function**.

8.5.2.5 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 content repository 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.

8.5.2.6 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.

8.5.2.7 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 comparison operators function the same way they do in XPath when applied to multi-value properties: if the predicate is true of at least one value of a multi-value property then it is true for the property as a whole (see 6.6.4.10 **Searching Multi-value Properties**).

8.5.2.8 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:

  o The property is declared explicitly in the node type but happens not to be present on this particular node.

  o 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.

  o 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).

8.5.2.9 Undefined Property Types

- When a node type **T** defines a property **P** with an **UNDEFINED** property type this means that two node instances of type **T** can each have a property called **P** but with differing property types. In such cases the database view of **T** is a single column of type **VARCHAR** (see below). In other words, the different types are converted to strings.

8.5.2.10 Data Type Mapping

The following type mapping governs the usage of property variables and literals within a content repository SQL statement.

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)**

8.5.2.11 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 **FROM** 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 **FROM** 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.

8.5.3 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 propname [DESC|ASC]
            {',' propname [DESC|ASC]}

proplist ::= propname {',' propname}

ntlist ::= ntname {',' ntname}

whereexp ::= propname op value |
             propname IS NULL |
             propname IS NOT NULL |
             like |
             contains |
             whereexp AND whereexp |
             whereexp OR whereexp |
             NOT whereexp |
             '(' whereexp ')' |
             joinpropname '=' joinpropname


op ::= '='|'>'|'<'|'>='|'<='|'<>'

joinpropname ::= quotedjoinpropname |
                 unquotedjoinpropname

quotedjoinpropname ::= ''' unquotedjoinpropname '''

unquotedjoinpropname ::= ntname '.jcr:path'

propname ::= 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 [ escape ]

likepattern ::= ''' likechar { likepattern } '''

likechar ::= char | '%' | '_'

escape ::= ESCAPE ''' likechar '''

char ::= /* Any character valid within the
            string representation of a value
            except for the characters % and _
            themselves. These must be escaped  */

contains ::= CONTAINS(scope ',' searchexp ')'

scope ::= unquotedpropname | '.'
searchexp ::= ''' exp '''

exp ::= [-]term {whitespace [OR] whitespace [-]term}

term ::= word | '"' word {whitespace word} '"'

word ::= /* A string containing no whitespace */

whitespace ::= /* A string of only whitespace*/
```

8.5.4 SQL Syntax in Detail

A SQL statement in a content repository 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.

8.5.4.1 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 8.5.2 *Database View*. See also 6.6.3.1 **Column Specifier**.

8.5.4.2 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 table (in the database view) searched will contain all nodes of the named type plus all node of subtypes of the named type. However the columns available in that table will reflect only those declared or inherited by the named type. To search all node types (tables) one would

specify **FROM nt:base** (though an implementation can always prevent searches whose scope is unfeasibly large). See also 6.6.3.2 **Type Constraint**.

8.5.4.3 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, 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 **LONG**s can be converted to **DOUBLE**s; 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 6.6.3.3 **Property Constraint** and 6.6.4.10 *Searching Multi-value Properties.*

8.5.4.4 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%**'

See also 8.5.2.2 *Pseudo-property jcr:path*, 6.6.3.3 **Property Constraint**, 6.6.3.4 **Path Constraint** and 6.6.5.1 **jcr:like Function**.

8.5.4.5 CONTAINS

Within the **WHERE** clause, the **CONTAINS** function is used to embed a statement in a full-text search language. The function takes two parameters: **scope** and **searchexp** (see EBNF above)

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 double quotes ("").

- The entire text search expression (`searchexp` in the EBNF, above) *must be* delimited by single 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` specifies the particular property that the full-text search is to be performed on. However, support for searching on particular properties is not required. Specifying '.' indicates that the full-text search is to be done on all *indexed properties* of the nodes specified by the rest of the query. Only support for a scope of '.' is required.

The scope of the `CONTAINS` clause specifying '.' 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 nodes of type `mynt:product` that have am indexed 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 may be returned as in score column. The specification does not define the calculation of the score value, it is implementation specific.

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

See also **6.6.3.3 Property Constraint**, 6.6.5.2 **jcr:contains Function** and 8.5.2.4 *Pseudo-property*.

8.5.4.6 ORDER BY

The `ORDER BY` clause causes the returned `QueryResult` objects to be sorted according to the value of a particular property.

Implementations must support ordering on `jcr:score` (or `jcr:score(...)` depending on the implementation). Support for ordering on `PATH` and `NAME` properties is not required. If it is supported then the collation sequence for these types is implementation specific.

See also 8.5.2.4 *Pseudo-property*, 6.6.3.5 **Ordering Specifier** and 6.6.5.5 **order by Clause**.

8.5.5 Query Results

Results are returned in the same structure as with XPath queries. See 6.6.3 **Structure of a Query** and 6.6.12 *Query Results*.