RFC 9 - Preferences API

Members Only
cpeg-rfc_9_preferences-1_00

March 10, 2001

19 Pages

Abstract

The Preferences interface allows applications to store and retrieve system and user preference data. This data is stored persistently in an implementation-dependent backing store. Typical information stored in the user preference tree might include font choice, and color choice for a bundle which interacts with the user via a servlet. Typical information stored in the system preference tree might include installation configuration data, or things like high score information for a game program. The PreferencesService allows each bundle using this service to have its own preference trees: one for system preferences and one for each user.

0 Document Information

0.1 Table of Contents

All Page Within This Box

0.2 Status

This document specifies the Preferences and PreferencesService interfaces for the core platform of the Open Services Gateway Initiative, and requests discussion and suggestions for improvements. Distribution of this document is unlimited within OSGi.

0.3 Acknowledgement

The authors wish to thank Joshua Bloch for his assistance, and for allowing us to freely borrow from his design and implementation work on a Preferences API for the Java 2 Standard Edition [2].

0.4 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in [1].

`Source code is shown in this typeface.`

0.5 Revision History

Revision	Date	Comments
Draft A	January 8, 2001	First draft in new OSGi RFC format. As agreed at the Miami meeting, the following changes have been made: 1. Renamed the synch() method to sync(). 2. Changed flush() and sync() to require that the whole subtree be flushed or synchronized, not just the node itself. 3. Removed the event classes and listener interfaces. 4. Made node names and key names case-sensitive.

All Page Within This Box

Revision	Date	Comments
Draft B	February 20, 2001	Added a reference to the JCP JSR 10. Extended Motivation and Rationale section. Added a note in PreferencesService that newly installed bundles should have no preferences. Added MAX_NAME_LENGTH and inserted the numerical values of all three MAX…LENGTH constants. Fixed an editing error in Preferences.remove() Removed Preferences.toString()
Draft C	March 6, 2001	Removed all three MAX…LENGTH constants as decided by CPEG at the Dallas meeting. Also, some minor formatting changes.
Final	March 10, 2001	

1 Introduction

This interface allows applications to store and retrieve user and system preference data. This data is stored persistently in an implementation-dependent backing store. Typical implementations include flat files, OS-specific registries, directory servers and SQL databases.

For each bundle, there is a separate tree of preference nodes for each user, and one for system preferences. The precise description of "user" and "system" will vary from one bundle to another.

Each Preferences node has attached to it a set of actual preferences, which are key-value pairs. Each node can also have attached to it any number of child nodes and thus a tree of nodes can be built. Nodes can be modified i.e. values in a key-value pair can be changed, and key-value pairs can be added and removed. Nodes can also be added or removed.

Preferences trees exist in memory on the gateway and also in a persistent store, which may be on the gateway or on some backend server, or both. These trees may not always be in sync. This interface provides methods that will synchronize the trees, and flush changes from the gateway to the persistent store.

No mechanism is provided for one bundle to access the preferences of another. If a bundle wishes to allow another bundle to access its preferences, it can pass a Preferences or PreferencesService object to that bundle.

All Page Within This Box

2 Motivation and Rationale

Many bundles will need to maintain some preference (attribute/value) data persistently. They could do this using the java.util.Properties and the local file storage (accessible via BundleContext.getDataFile), but then there would be no way for a gateway operator to maintain or replicate the data on a remote server. This might be necessary or desirable for several reasons:

1. Some gateways may not support any file storage.
2. It may be desired to have a back-up in case of data loss on the gateway computer, or to avoid reconfiguration after a hardware upgrade.
3. It may be useful to support shared access to some of the preference data, either from another gateway, or from an application running on a remote server.

Using properties files can also be cumbersome when many of them are needed. The Preferences API provides a hierarchical structure which is much more convenient to use in this case.

An additional potential benefit of this API is that it allows user-specific preference data to be kept in a well-defined place, so that a user management system could locate it. This could be useful for such operations as cleaning up when a user is deleted, or to allow a user's preferences to be cloned for a new user.

It should be noted that this API is not a general-purpose persistence API. In particular, it cannot handle large data items. The intent is to allow for a variety of compact, efficient implementations. If the API were more general, implementations would have to be either larger or less efficient. Also, a different API might be appropriate for large data items.

Also note that this API does not do Configuration Management, although it could act as a persistence mechanism for a Configuration Management service.

3 API Specification

3.1 Interface PreferencesService

public interface **PreferencesService**

The Preferences Service.

All Page Within This Box

Each bundle using this service has its own set of preference trees: one for system preferences, and one for each user.

A PreferencesService object is specific to the bundle that obtained it from the service registry. If a bundle wishes to allow another bundle to access its preferences, it can pass its PreferencesService object to that bundle.

When a bundle is newly installed, calls to the methods in this service will return root nodes with no children and no preferences. The preference trees of a bundle persist for the lifetime of the bundle, including any bundle updates.

Method Summary

Preferences	**getSystemPreferences**() Returns the root system preference node for the calling bundle.
Preferences	**getUserPreferences**(java.lang.String name) Returns the root preference node for the specified user and the calling bundle.

Method Detail

3.1.1 getSystemPreferences

public Preferences **getSystemPreferences**()

Returns the root system preference node for the calling bundle.

3.1.2 getUserPreferences

public Preferences **getUserPreferences**(java.lang.String name)

Returns the root preference node for the specified user and the calling bundle.

3.2 Interface Preferences

public interface **Preferences**

A node in a hierarchical collection of preference data. This interface allows applications to store and retrieve user and system preference data. This data is stored persistently in an implementation-dependent backing store. Typical implementations include flat files, OS-specific registries, directory servers and SQL databases.

For each bundle, there is a separate tree of preference nodes for each user, and one for system preferences. The precise description of "user" and "system" will vary from one bundle to another. Typical information stored in the user preference tree might include font choice, and color choice for a bundle which interacts with the user via a servlet. Typical information stored in the system preference

All Page Within This Box

tree might include installation configuration data, or things like high score information for a game program.

Nodes in a preference tree are named in a similar fashion to directories in a hierarchical file system. Every node in a preference tree has a *node name* (which is not necessarily unique), a unique *absolute path name*, and a path name *relative* to each ancestor including itself.

The root node has a node name of the empty string (""). Every other node has an arbitrary node name, specified at the time it is created. The only restrictions on this name are that it cannot be the empty string, and it cannot contain the slash character ('/').

The root node has an absolute path name of `"/"`. Children of the root node have absolute path names of `"/" + <`*node name*`>`. All other nodes have absolute path names of `<`*parent's absolute path name*`> + "/" + <`*node name*`>`. Note that all absolute path names begin with the slash character.

A node *n*'s path name relative to its ancestor *a* is simply the string that must be appended to *a*'s absolute path name in order to form *n*'s absolute path name, with the initial slash character (if present) removed. Note that:

- No relative path names begin with the slash character.
- Every node's path name relative to itself is the empty string.
- Every node's path name relative to its parent is its node name (except for the root node, which does not have a parent).
- Every node's path name relative to the root is its absolute path name with the initial slash character removed.

Note finally that:

- No path name contains multiple consecutive slash characters.
- No path name with the exception of the root's absolute path name end in the slash character.
- Any string that conforms to these two rules is a valid path name.

Each preference node has zero or more preferences associated with it, where a preference consists of a name and a value. The bundle writer is free to choose any appropriate names for preferences. Their values can be of type String, long, int, boolean or byte array, but they can always be accessed as if they were Strings.

All node name and preference name comparisons are case-sensitive.

All of the methods that modify preference data are permitted to operate asynchronously; they may return immediately, and changes will eventually propagate to the persistent backing store, with an

implementation-dependent delay. The `flush` method may be used to synchronously force updates to the backing store.

Implementations must automatically attempt to flush to the backing store any pending updates for a bundle's preferences when the bundle is stopped or otherwise ungets the PreferencesService.

The methods in this class may be invoked concurrently by multiple threads in a single Java Virtual Machine (JVM) without the need for external synchronization, and the results will be equivalent to some serial execution. If this class is used concurrently *by multiple JVMs* that store their preference data in the same backing store, the data store will not be corrupted, but no other guarantees are made concerning the consistency of the preference data.

Implementation note: to handle cases where connections to the backing store are unreliable, implementations may provide a configuration option to control the time-out delay for the `flush` and sync methods.

Method Summary

`java.lang.String`	**`absolutePath`**`()` Returns this preference node's absolute path name.
`Preferences[]`	**`children`**`()` Returns the children of this preference node.
`void`	**`clear`**`()` Removes all of the preferences (key-value associations) in this preference node.
`void`	**`flush`**`()` Forces any changes in the contents of this preference node and its descendants to the persistent store.
`java.lang.String`	**`get`**`(java.lang.String key, java.lang.String def)` Returns the value associated with the specified key in this preference node.
`boolean`	**`getBoolean`**`(java.lang.String key, boolean def)` Returns the boolean value represented by the string associated with the specified key in this preference node.
`byte[]`	**`getByteArray`**`(java.lang.String key, byte[] def)` Returns the byte array value represented by the string associated with the specified key in this preference node.
`int`	**`getInt`**`(java.lang.String key, int def)` Returns the int value represented by the string associated with the specified key in this preference node.
`long`	**`getLong`**`(java.lang.String key, long def)` Returns the long value represented by the string associated with the specified key in this preference node.
`java.lang.String[]`	**`keys`**`()` Returns all of the keys that have an associated value in this preference node.
`java.lang.String`	**`name`**`()`

All Page Within This Box

	Returns this preference node's name, relative to its parent.
Preferences	**node**(java.lang.String pathName) Returns the named preference node, creating it and any of its ancestors if they do not already exist.
boolean	**nodeExists**(java.lang.String pathName) Returns true if the named preference node exists.
Preferences	**parent**() Returns the parent of this preference node, or null if this is the root.
void	**put**(java.lang.String key, java.lang.String value) Associates the specified value with the specified key in this preference node.
void	**putBoolean**(java.lang.String key, boolean value) Associates a string representing the specified boolean value with the specified key in this preference node.
void	**putByteArray**(java.lang.String key, byte[] value) Associates a string representing the specified byte array with the specified key in this preference node.
void	**putInt**(java.lang.String key, int value) Associates a string representing the specified int value with the specified key in this preference node.
void	**putLong**(java.lang.String key, long value) Associates a string representing the specified long value with the specified key in this preference node.
void	**remove**(java.lang.String key) Removes the value associated with the specified key in this preference node, if any.
void	**removeNode**() Removes this preference node and all of its descendants, invalidating any preferences contained in the removed nodes.
void	**sync**() Ensures that future reads from this preference node and its descendants reflect any changes that were committed to the persistent store (from any VM) prior to the sync invocation.

Method Detail

3.2.1 put

```
public void put(java.lang.String key,
                java.lang.String value)
```

Associates the specified value with the specified key in this preference node.

Parameters:

key - key with which the specified value is to be associated.

value - value to be associated with the specified key.

Throws:

NullPointerException - if key or value is null.

All Page Within This Box

IllegalStateException - if this node (or an ancestor) has been removed with the `removeNode` method.

3.2.2 get

```
public java.lang.String get(java.lang.String key,
                            java.lang.String def)
```

Returns the value associated with the specified key in this preference node. Returns the specified default if there is no value associated with the key, or the backing store is inaccessible.

Parameters:

`key` - key whose associated value is to be returned.

`def` - the value to be returned in the event that this preference node has no value associated with `key`.

Returns:

the value associated with `key`, or `def` if no value is associated with `key`.

Throws:

IllegalStateException - if this node (or an ancestor) has been removed with the `removeNode` method.

NullPointerException - if key is `null`. (A `null` default *is* permitted.)

3.2.3 remove

```
public void remove(java.lang.String key)
```

Removes the value associated with the specified key in this preference node, if any.

Parameters:

`key` - key whose mapping is to be removed from the preference node.

Throws:

IllegalStateException - if this node (or an ancestor) has been removed with the `removeNode` method.

See Also:

`get(String,String)`

3.2.4 clear

```
public void clear()
            throws BackingStoreException
```

Removes all of the preferences (key-value associations) in this preference node. This call has no effect on any descendants of this node.

Throws:

BackingStoreException - if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.

IllegalStateException - if this node (or an ancestor) has been removed with the `removeNode` method.

See Also:

`#remove()`

3.2.5 putInt

```
public void putInt(java.lang.String key,
                   int value)
```

Associates a string representing the specified int value with the specified key in this preference node. The associated string is the one that would be returned if the int value were passed to `Integer.toString(int)`. This method is intended for use in conjunction with `getInt(java.lang.String, int)`.

All Page Within This Box

Implementor's note: it is *not* necessary that the preference value be represented by a string in the backing store. If the backing store supports integer values, it is not unreasonable to use them. This implementation detail is not visible through the `Preferences` API, which allows the value to be read as an int (with `getInt` or a string (with `get`).

Parameters:
`key` - key with which the string form of value is to be associated.
`value` - value whose string form is to be associated with key.
Throws:
NullPointerException - if key is `null`.
IllegalStateException - if this node (or an ancestor) has been removed with the `removeNode` method.
See Also:
`getInt(String,int)`

3.2.6 getInt

```
public int getInt(java.lang.String key,
                  int def)
```

Returns the int value represented by the string associated with the specified key in this preference node. The string is converted to an integer as by `Integer#parseInt(String)`. Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if `Integer.parseInt(String)` would throw a `NumberFormatException` if the associated value were passed. This method is intended for use in conjunction with `putInt(java.lang.String, int)`.
Parameters:
`key` - key whose associated value is to be returned as an int.
`def` - the value to be returned in the event that this preference node has no value associated with `key` or the associated value cannot be interpreted as an int.
Returns:
the int value represented by the string associated with `key` in this preference node, or `def` if the associated value does not exist or cannot be interpreted as an int.
Throws:
IllegalStateException - if this node (or an ancestor) has been removed with the `removeNode` method.
See Also:
`putInt(String,int)`, `get(String,String)`

3.2.7 putLong

```
public void putLong(java.lang.String key,
                    long value)
```

Associates a string representing the specified long value with the specified key in this preference node. The associated string is the one that would be returned if the long value were passed to `Long.toString(long)`. This method is intended for use in conjunction with `getLong(java.lang.String, long)`.

Implementor's note: it is *not* necessary that the preference value be represented by a string in the backing store. If the backing store supports long values, it is not unreasonable to use them. This implementation detail is not visible through the `Preferences` API, which allows the value to be read as a long (with `getLong` or a string (with `get`).

Parameters:
`key` - key with which the string form of value is to be associated.

All Page Within This Box

`value` - value whose string form is to be associated with key.
Throws:
NullPointerException - if key is `null`.
IllegalStateException - if this node (or an ancestor) has been removed with the `removeNode` method.
See Also:
`getLong(String,long)`

3.2.8 getLong

```
public long getLong(java.lang.String key,
                    long def)
```

Returns the long value represented by the string associated with the specified key in this preference node. The string is converted to a long as by `Long#parseLong(String)`. Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if `Long.parseLong(String)` would throw a `NumberFormatException` if the associated value were passed. This method is intended for use in conjunction with `putLong(java.lang.String, long)`.

Parameters:
`key` - key whose associated value is to be returned as a long.
`def` - the value to be returned in the event that this preference node has no value associated with `key` or the associated value cannot be interpreted as a long.
Returns:
the long value represented by the string associated with `key` in this preference node, or `def` if the associated value does not exist or cannot be interpreted as a long.
Throws:
IllegalStateException - if this node (or an ancestor) has been removed with the `removeNode` method.
See Also:
`putLong(String,long)`, `get(String,String)`

3.2.9 putBoolean

```
public void putBoolean(java.lang.String key,
                       boolean value)
```

Associates a string representing the specified boolean value with the specified key in this preference node. The associated string is `"true"` if the value is true, and `"false"` if it is false. This method is intended for use in conjunction with `getBoolean(java.lang.String, boolean)`.

Implementor's note: it is *not* necessary that the preference value be represented by a string in the backing store. If the backing store supports boolean values, it is not unreasonable to use them. This implementation detail is not visible through the `Preferences` API, which allows the value to be read as an boolean (with `getBoolean`) or a string (with `get`).

Parameters:
`key` - key with which the string form of value is to be associated.
`value` - value whose string form is to be associated with key.
Throws:
NullPointerException - if key is `null`.
IllegalStateException - if this node (or an ancestor) has been removed with the `removeNode` method.
See Also:
`getBoolean(String,boolean)`, `get(String,String)`

All Page Within This Box

3.2.10 getBoolean

```
public boolean getBoolean(java.lang.String key,
                          boolean def)
```

Returns the boolean value represented by the string associated with the specified key in this preference node. Valid strings are `"true"`, which represents true, and `"false"`, which represents false. Case is ignored, so, for example, `"TRUE"` and `"False"` are also valid. This method is intended for use in conjunction with `putBoolean(java.lang.String, boolean)`.

Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if the associated value is something other than `"true"` or `"false"`, ignoring case.

Parameters:
`key` - key whose associated value is to be returned as a boolean.
`def` - the value to be returned in the event that this preference node has no value associated with `key` or the associated value cannot be interpreted as a boolean.
Returns:
the boolean value represented by the string associated with `key` in this preference node, or `null` if the associated value does not exist or cannot be interpreted as a boolean.
Throws:
IllegalStateException - if this node (or an ancestor) has been removed with the `removeNode` method.
See Also:
`get(String,String)`, `putBoolean(String,boolean)`

3.2.11 putByteArray

```
public void putByteArray(java.lang.String key,
                         byte[] value)
```

Associates a string representing the specified byte array with the specified key in this preference node. The associated string is the *Base64* encoding of the byte array, as defined in RFC 2045, Section 6.8, with one minor change: the string will consist solely of characters from the *Base64 Alphabet*; it will not contain any newline characters. This method is intended for use in conjunction with `getByteArray(java.lang.String, byte[])`.

Implementor's note: it is *not* necessary that the preference value be represented by a string in the backing store. If the backing store supports byte array values, it is not unreasonable to use them. This implementation detail is not visible through the `Preferences` API, which allows the value to be read as an a byte array (with `getByteArray`) or a string (with `get`).

Parameters:
`key` - key with which the string form of value is to be associated.
`value` - value whose string form is to be associated with key.
Throws:
NullPointerException - if key or value is `null`.
IllegalStateException - if this node (or an ancestor) has been removed with the `removeNode` method.
See Also:
`getByteArray(String,byte[])`, `get(String,String)`

3.2.12 getByteArray

```
public byte[] getByteArray(java.lang.String key,
```

All Page Within This Box

byte[] def)

Returns the byte array value represented by the string associated with the specified key in this preference node. Valid strings are *Base64* encoded binary data, as defined in RFC 2045, Section 6.8, with one minor change: the string must consist solely of characters from the *Base64 Alphabet*; no newline characters or extraneous characters are permitted. This method is intended for use in conjunction with `putByteArray(java.lang.String, byte[])`.

Returns the specified default if there is no value associated with the key, the backing store is inaccessible, or if the associated value is not a valid Base64 encoded byte array (as defined above).

Parameters:
`key` - key whose associated value is to be returned as a byte array.
`def` - the value to be returned in the event that this preference node has no value associated with `key` or the associated value cannot be interpreted as a byte array.
Returns:
the byte array value represented by the string associated with `key` in this preference node, or `def` if the associated value does not exist or cannot be interpreted as a byte array.
Throws:
IllegalStateException - if this node (or an ancestor) has been removed with the `removeNode` method.
See Also:
`get(String,String)`, `Array(String,byte array)`

3.2.13 keys

```
public java.lang.String[] keys()
                          throws BackingStoreException
```

Returns all of the keys that have an associated value in this preference node. (The returned array will be of size zero if this node has no preferences.)
Returns:
an array of the keys that have an associated value in this preference node.
Throws:
BackingStoreException - if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.
IllegalStateException - if this node (or an ancestor) has been removed with the `removeNode` method.

3.2.14 children

```
public Preferences[] children()
                       throws BackingStoreException
```

Returns the children of this preference node. (The returned array will be of size zero if this node has no children.)
Returns:
the children of this preference node.
Throws:
BackingStoreException - if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.
IllegalStateException - if this node (or an ancestor) has been removed with the `removeNode` method.

3.2.15 parent

```
public Preferences parent()
```

All Page Within This Box

Returns the parent of this preference node, or `null` if this is the root.
Returns:
the parent of this preference node.
Throws:
IllegalStateException - if this node (or an ancestor) has been removed with the `removeNode` method.

3.2.16 node

`public Preferences` **`node`**`(java.lang.String pathName)`

Returns the named preference node, creating it and any of its ancestors if they do not already exist. Accepts a relative or absolute pathname. Absolute pathnames (which begin with `'/'`) are interpreted relative to the root of this preference node. Relative pathnames (which begin with any character other than `'/'`) are interpreted relative to this preference node itself. The empty string (`""`) is a valid relative pathname, referring to this preference node itself.

If the returned node did not exist prior to this call, this node and any ancestors that were created by this call are not guaranteed to become persistent until the `flush` method is called on the returned node (or one of its descendants).

Parameters:
`pathName` - the path name of the preference node to return.
Returns:
the specified preference node.
Throws:
java.lang.IllegalArgumentException - if the path name is invalid.
IllegalStateException - if this node (or an ancestor) has been removed with the `removeNode` method.
See Also:
`flush()`

3.2.17 nodeExists

`public boolean` **`nodeExists`**`(java.lang.String pathName)`
`throws BackingStoreException`

Returns true if the named preference node exists. Accepts a relative or absolute pathname. Absolute pathnames (which begin with `'/'`) are interpreted relative to the root of this preference node. Relative pathnames (which begin with any character other than `'/'`) are interpreted relative to this preference node itself. The pathname `""` is valid, and refers to this preference node itself.

If this node (or an ancestor) has already been removed with the `removeNode` method, it *is* legal to invoke this method, but only with the pathname `""`; the invocation will return `false`. Thus, the idiom `p.nodeExists("")` may be used to test whether p has been removed.

Parameters:
`pathName` - the path name of the node whose existence is to be checked.
Returns:
true if the specified node exists.
Throws:
BackingStoreException - if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.
IllegalStateException - if this node (or an ancestor) has been removed with the `removeNode` method and `pathname` is not the empty string (`""`).

All Page Within This Box

3.2.18 removeNode

```
public void removeNode()
            throws BackingStoreException
```

Removes this preference node and all of its descendants, invalidating any preferences contained in the removed nodes. Once a node has been removed, attempting any method other than `name()`, `absolutePath()` or `nodeExists("")` on the corresponding `Preferences` instance will fail with an `IllegalStateException`. (The methods defined on `Object` can still be invoked on a node after it has been removed; they will not throw `IllegalStateException`.)

The removal is not guaranteed to be persistent until the `flush` method is called on the parent of this node. (It is illegal to remove the root node.)

Throws:
IllegalStateException - if this node (or an ancestor) has already been removed with the `removeNode` method.
See Also:
`flush()`

3.2.19 name

```
public java.lang.String name()
```

Returns this preference node's name, relative to its parent.

Returns:
this preference node's name, relative to its parent.

3.2.20 absolutePath

```
public java.lang.String absolutePath()
```

Returns this preference node's absolute path name. Note that:

- The path name of the root node is `"/"`.
- Path names other than that of the root node may not end in slash (`'/'`).
- `"."` and `".."` have no special significance in preference path names.
- The only illegal path names are those that contain multiple consecutive slashes, or that end in slash and are not the root.

Returns:
this preference node's absolute path name.

3.2.21 flush

```
public void flush()
          throws BackingStoreException
```

All Page Within This Box

Forces any changes in the contents of this preference node and its descendants to the persistent store. Once this method returns successfully, it is safe to assume that all changes made in the subtree rooted at this node prior to the method invocation have become permanent.

Implementations are free to flush changes into the persistent store at any time. They do not need to wait for this method to be called.

When a flush occurs on a newly created node, it is made persistent, as are any ancestors (and descendants) that have yet to be made persistent. Note however that any preference value changes in ancestors are *not* guaranteed to be made persistent.

Throws:
BackingStoreException - if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.
IllegalStateException - if this node (or an ancestor) has been removed with the `removeNode()` method.
See Also:
`sync()`

3.2.22 sync

```
public void sync()
          throws BackingStoreException
```

Ensures that future reads from this preference node and its descendants reflect any changes that were committed to the persistent store (from any VM) prior to the `sync` invocation. As a side-effect, forces any changes in the contents of this preference node and its descendants to the persistent store, as if the `flush` method had been invoked on this node.

Throws:
BackingStoreException - if this operation cannot be completed due to a failure in the backing store, or inability to communicate with it.
IllegalStateException - if this node (or an ancestor) has been removed with the `removeNode()` method.
See Also:
`flush()`

3.3 Class BackingStoreException

```
java.lang.Object
  |
  +-java.lang.Throwable
        |
        +-java.lang.Exception
              |
              +-org.osgi.service.prefs.BackingStoreException
```

---

public class **BackingStoreException**
extends java.lang.Exception

Thrown to indicate that a preferences operation could not complete because of a failure in the backing store, or a failure to contact the backing store.

All Page Within This Box

Constructor Summary
BackingStoreException(java.lang.String s) Constructs a BackingStoreException with the specified detail message.

Methods inherited from class java.lang.Throwable
fillInStackTrace, getLocalizedMessage, getMessage, printStackTrace, printStackTrace, printStackTrace, toString

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

3.3.1 BackingStoreException

public **BackingStoreException**(java.lang.String s)
Constructs a BackingStoreException with the specified detail message.

4 Security Considerations

Preferences are private to a bundle, unless it chooses to expose them to other bundles. Therefore, no permission checks are required.

All Page Within This Box

5 Document Support

5.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Bloch, J. et al., JSR #000010 Preferences API Specification, http://java.sun.com/aboutJava/communityprocess/jsr/jsr_010_prefs.html

5.2 Author's Addresses

Name	David Bowen
Company	Sun Microsystems, Inc.
Address	901 San Antonio Road, M/S UCUP01-203 Palo Alto, CA 94303
Voice	+1 408 343 1407
e-mail	david.bowen@sun.com

Name	Venkatesh Narayanan
Company	Sun Microsystems, Inc.
Address	901 San Antonio Road, M/S UCUP01-207 Palo Alto, CA 94303
Voice	+1 408 517 5311
e-mail	venky.narayanan@sun.com

5.3 Acronyms and Abbreviations

OSGi	Open Services Gateway Initiative

5.4 End of Document

All Page Within This Box