FileNet Content Services Java Connector v3.0 Developer's Guide

For updates to any Content Services Java Connector documentation, choose the Documentation link on the FileNet Worldwide Support web site **http://www.css.filenet.com/**, and navigate to the current CS Java Connector release for your platform. If you do not have a customer support services (CSS) Web Account, click the New User button and follow the online instructions.

Copyright

Software Numbers

Software Release: 3.0

Trademarks

Contents

Code Listings

Overview

The CS Java Connector v3.0 provides networked, platform-independent access to objects stored in Content Services (CS). The toolkit consists of the CS Java Connector and accompanying documentation.

- To install and configure the CS Java Connector, see "Installation" on page 9.
- For an introduction to key concepts and components of the CS Java Connector, see "Getting Started" on page 21.
- For detailed information about specific objects in the CS Java Connector (ObjectStore, Folder, Document, Permission, etc.), refer to the applicable section in this guide.

Additional Documentation

In addition to this Developer's Guide, you'll find related information in:

- The *Release Notes* listing known issues and other last-minute information. The *Release Notes* are available on www.css.filenet.com.
- *CS Java Connector v3.0 Javadoc* providing reference information on each interface and class in the CS Java Connector. The Javadoc is located in the Documentation directory of the distribution CD. To launch Javadoc, choose Documentation\index.html.
- *Content Services API Manual* included with the Content Services Toolkit.
- *Content Services* documentation, Solaris Oracle edition.
- *Content Services* documentation, HP-UX Oracle edition.

For Content Services Java Connector documentation updates, log into www.css.filenet.com and choose the Product Info tab. Navigate to the Product Documentation for your current Content Services Java Connector release and platform. If you do not have a customer support services (CSS) Web Account, click the New User link and follow the online instructions.

Installation

This section tells you how to install and configure the CS Java Connector v3.0 on a development machine for the Windows, Solaris and HP-UX platforms.

Requirements

- Java 2 Development Kit v1.4.2 or higher.
- One of the following OS platforms:
  - Windows 2000, any version
  - Windows 2003, any version
  - Sun Solaris 8
  - Sun Solaris 9
  - HP-UX 11i
- One of the following Content Services
  - Content Services 5.3 with Hot Fix Pack #3 + CS-5.3.0-1011
  - Content Services 5.4 with CS-5.4.0-1005
- One of the following Application Server (Optional):
  - Windows 2000/ Windows 2003: BEA WebLogic v7.0 SP5, BEA WebLogic v8.1 SP3 or IBM WebSphere Business Integration Server Foundation v5.1
  - Sun Solaris 8/ 9: BEA WebLogic v7.0 SP5, BEA WebLogic v8.1 SP3, IBM WebSphere Business Integration Server Foundation v5.1 or Sun Java System Application Server Standard Edition v7.0 SP 2004Q2
  - HP-UX 11i: BEA WebLogic v7.0 SP5, BEA WebLogic v8.1 SP3 or IBM Business Integration Server Foundation v5.1

Installing CS Java Connector

A Java-based program with a wizard interface is provided to install the CS Java Connector and documentation. Note that you must manually set system environment variables.

1. From the CS Java Connector CD, launch the installer for your platform. The installers are located in the Setup directory.
   - Windows 2000/2003: filenet_cs_java_connector_v30_win32.exe
   - Sun Solaris version 8/9: filenet_cs_java_connector_v30_solaris.bin
   - HP-UX 11i: filenet_cs_java_connector_v30_hpux.bin
2. Respond to the wizard prompts.
3. When you complete the wizard, refer to “Post-Installer Steps” on page 11 for additional installation information.

4. To setup environment for developing applications with CS Java Connector, see “Setting Up Your Development Environment” on page 19.
5. If you intend to use the CS Java Connector in a Web application, see “Using CS Java Connector in a Web Application” on page 20.

Note: To uninstall, see “Uninstalling the CS Java Connector” on page 20.

Post-Installer Steps

After running the installer program, you must perform additional steps to complete the installation. Use the table below to identify the steps for your platform.

Table 1: Post-Installer Steps

Platform	Step
Windows only	Install the CS client libraries included with your Content Services package. **Note:** For the UNIX platform, the CS client libraries were installed by the installer.
Windows only	If you will be accessing Content Services using an Oracle database, install the Oracle client libraries.
Windows only	Create an ODBC Data Source (DSN) for each CS object store (library) you will access.
Windows only	Set environment variables. See “Windows: Setting Environment Variables” on page 12.
Solaris only	Set environment variables. See “Solaris: Setting Environment Variables” on page 13.
HP-UX only	Set environment variables. See “HP-UX: Setting Environment Variables” on page 14.
Windows, Solaris and HP-UX	Modify the FileNet\CS Java Connector\config\EntireNetwork.properties file to contain valid Content Services object stores (libraries). See “EntireNetwork.properties” on page 16.
Windows, Solaris and HP-UX	Review the section on configuration files. You may have to modify files in addition to EntireNetwork.properties. See “Configuration Files” on page 16.

Windows: Setting Environment Variables

On a Windows platform, you must set the following environment variables:

Table 2: Windows Environment Variables

Environment Variable	Setting
CSJC_HOME	The CSJC_HOME variable must include Installation location of CS Java Connector. For example: CSJC_HOME=C:\Programs Files\FileNet\CS Java Connector
CLASSPATH	The CLASSPATH variable must include: %CSJC_HOME%\Java\panagon.jar %CSJC_HOME%\lib\connector.jar %CSJC_HOME%\lib\javaapi.jar %CSJC_HOME%\lib\jta.jar %CSJC_HOME%\lib\log4j.jar %CSJC_HOME%\lib\xalan.jar %CSJC_HOME%\lib\xerces.jar %CSJC_HOME%\lib\soap.jar %CSJC_HOME%\config
PATH	The PATH variable must include: %CSJC_HOME%\jni %CSJC_HOME%\lib PATH variable should also include paths pointing to Content Services client libraries.

Solaris: Setting Environment Variables

On a Solaris platform, you must set the following environment variables for the CS Java Connector:

Table 3: Solaris Environment Variables

Environment Variable	Setting
CSJC_HOME	The CSJC_HOME variable must include Installation location of CS Java Connector. For example: "/opt/FileNet/CS Java Connector"
CLASSPATH	The CLASSPATH variable must include: $CSJC_HOME/Java/panagon.jar $CSJC_HOME/lib/connector.jar $CSJC_HOME/lib/javaapi.jar $CSJC_HOME/lib/jta.jar $CSJC_HOME/lib/log4j.jar $CSJC_HOME/lib/soap.jar $CSJC_HOME/lib/xalan.jar $CSJC_HOME/lib/xerces.jar $CSJC_HOME/config
LD_LIBRARY_PATH	For connecting to Content Services v5.3, LD_LIBRARY_PATH must include: $CSJC_HOME/lib $CSJC_HOME/jni/lib53 $CSJC_HOME/Shared $CSJC_HOME/Shared/ORACLE For connecting to Content Services v5.4, LD_LIBRARY_PATH must include: $CSJC_HOME/lib $CSJC_HOME/jni/lib54 $CSJC_HOME/Shared $CSJC_HOME/Shared/ORACLE In LD_LIBRARY_PATH, paths referring to Shared folder of CS Java Connector are for Content Services client libraries. These can be pointed to the other location also, where CS client libraries are already installed with mandatory patch applied.
NLS_LANG	NLS_LANG=AMERICAN_AMERICA.WE8ISO8859P1 **Note:** The value of this variable must match the value set for the Oracle database. In addition, this variable and the PANAGON_ENCODING variable must be properly mapped to each other. For details, see "Unix : Mapping NLS_LANG and PANAGON_ENCODING" on page 15.

HP-UX: Setting Environment Variables

On a HP-UX platform, you must set the following environment variables for the CS Java Connector:

Table 4: HP-UX Environment Variables

Environment Variable	Setting
CSJC_HOME	The CSJC_HOME variable must include Installation location of CS Java Connector. For example: "/opt/FileNet/CS Java Connector"
CLASSPATH	The CLASSPATH variable must include: $CSJC_HOME/Java/panagon.jar $CSJC_HOME/lib/connector.jar $CSJC_HOME/lib/javaapi.jar $CSJC_HOME/lib/jta.jar $CSJC_HOME/lib/log4j.jar $CSJC_HOME/lib/soap.jar $CSJC_HOME/lib/xalan.jar $CSJC_HOME/lib/xerces.jar $CSJC_HOME/config
SHLIB_PATH	For connecting to Content Services v5.3, SHLIB_PATH must include: $CSJC_HOME/lib $CSJC_HOME/jni/lib53 $CSJC_HOME/Shared $CSJC_HOME/Shared/ORACLE For connecting to Content Services v5.4, SHLIB_PATH must include: $CSJC_HOME/lib $CSJC_HOME/jni/lib54 $CSJC_HOME/Shared $CSJC_HOME/Shared/ORACLE In SHLIB_PATH, paths referring to Shared folder of CS Java Connector are for Content Services client libraries. These can be pointed to the other location also, where CS client libraries are already installed with mandatory patch applied.
NLS_LANG	NLS_LANG=AMERICAN_AMERICA.WE8ISO8859P1 **Note:** The value of this variable must match the value set for the Oracle database. In addition, this variable and the PANAGON_ENCODING variable must be properly mapped to each other. For details, see "Unix : Mapping NLS_LANG and PANAGON_ENCODING" on page 15.

Note: In addition, make sure that the environment variables for the Oracle database and the Content Services (CS) server are correctly set, and verify the CS/Oracle connection (using SQL Plus or other utility). For Oracle and Content Services configuration details, refer to the *Content Services Installation Guide, Solaris/ HP-UX Oracle Edition*.

Unix : Mapping NLS_LANG and PANAGON_ENCODING

On a Unix platform there are two environment variables used to define how characters are translated from the Oracle database into Java. One is NLS_LANG, an Oracle environment variable used by the CS/SPI API and Oracle to retrieve data in the specified code set. This variable usually begins with a language/territory specification, for example, AMERICAN_AMERICA.WE8MSWIN1252. The documentation for NLS_LANG can be found in the Oracle documentation.

The other environment variable is PANAGON_ENCODING. The CS Java Connector uses Java encoding to translate the retrieved data into Java strings. Valid Java encodings can be found at the following URL: http://www.iana.org/assignments/character-sets

Failure to have proper mappings between NLS_LANG and PANAGON_ENCODING **could result in severe performance degradation and/or mistranslated characters.**

Use the following rules when setting NLS_LANG and PANAGON_ENCODING.

- In the Unix environment, the Oracle NLS_LANG environment variable must be set, and the value of the variable must match the value set for the Oracle database. The absence of this setting will result in a ContentEngineException having a fault code of 350 and an error message about failure converting to/from multi-byte.
- If PANAGON_ENCODING is not set, the CS Java Connector will default to the following Java encoding based on the value of NLS_LANG:

NLS_LANG	Default Java Encoding
WE8MSWIN1252	windows-1252
WE8ISO8859P1	ISO-8859-1
WE8ISO8859P2	ISO-8859-2
WE8ISO8859P3,	ISO-8859-3
WE8ISO8859P4,	ISO-8859-4
WE8ISO8859P5,	ISO-8859-5
WE8ISO8859P6,	ISO-8859-6
WE8ISO8859P7,	ISO-8859-7
WE8ISO8859P8,	ISO-8859-8
WE8ISO8859P9,	ISO-8859-9
WE8ISO8859P10	ISO-8859-10
WE8ISO8859P11	ISO-8859-11
WE8ISO8859P12	ISO-8859-12
WE8ISO8859P13	ISO-8859-13
WE8ISO8859P14	ISO-8859-14
WE8ISO8859P15	ISO-8859-15

- If the NLS_LANG variable is set to a value other than what's listed in the above table, you must set the PANAGON_ENCODING variable with a properly mapped value; otherwise, an error will be raised.

  Note: The PANAGON_ENCODING variable must match the code set of the clients used to populate the data in the Oracle database.

- If your installation is using the Windows 1252 code set, note that this code set uses a Microsoft specific value for the Euro character. Specifying the ISO 8859-1 code set in NLS_LANG when the data in the database was populated with a Windows 1252 code set will not give an error, but the Euro will not be translated into Java correctly. When this is the case, use the WE8MSWIN1252 value for NLS_LANG to get a proper translation of the Euro character.

Configuration Files

This section describes the configuration files listed below that are installed with the CS Java Connector toolkit.

- EntireNetwork.properties
- mimeTypes.properties
- WcmApiConfig.properties

Your development work will require that you modify the default settings for at least EntireNetwork.properties. These files reside in the ../CS Java Connector/config directory.

EntireNetwork.properties

This configuration file has two purposes. First, it is used by the com.filenet.wcm.api.EntireNetwork interface to get a list of valid object stores (libraries). Second, it is used by all of the XML methods to retrieve id and/or name of an object store.

Each object store on a Content Services server is specified on a line in this format:

objectStore^*host*=*description*

where:

objectStore is the name of the library.

host is the name of the Content Services server.

description identifies the object store for your purposes.

For example:

```
pin2ms^Pinta=Pinta test machine
```

mimeTypes.properties

This file is used by the com.filenet.wcm.api.TransportInputStream class to return the mime type of a downloaded document. The file consists of a set of common/default translations.

Each line in the file maps file extensions to mime types in the following format:

file-extension=*mime-type*

For example:

```
.zip=application/x-zip-compressed
```

If TransportInputStream cannot locate a mime type by file extension, it returns the "application/octet-stream" mime type.

WcmApiConfig.properties

This configuration file contains session configuration settings expressed in the following keyword/value pairs:

Table 5: WcmApiConfig Keywords

Keyword	Description
CredentialsProtection	Identifies the default credentials protection scheme to be used. Value is either "Clear" or "Symmetric". Example: `CredentialsProtection = Clear` To encrypt credentials, see "Setting Up Symmetric Encryption" on page 18.
CryptoKeyFile	Points to the location of a file that contains the cryptographic keys to be used when `CredentialsProtection=Symmetric` is specified. Example: `CryptoKeyFile = C:\\filenet\\CryptoKeyFile.properties`
InactiveSessionTimeout	The number of consecutive milliseconds that a session can be inactive before the CS Java Connector terminates the session and the session context (SCTX) on Content Services is reclaimed. Note that each session context consumes a license on Content Services.

Keep in mind the following points about WcmApiConfig:

- Keywords are case-sensitive; string values are not.
- The CS Java Connector reads WcmApiConfig.properties as a Java PropertyResourceBundle, which defines the general syntax and part of the file naming conventions. Note that any literal backslash (\) characters used in the value part of the keyword/value entry must be doubled (\\).
- The configuration settings apply to all applications calling into the CS Java Connector. However, you can specify application ID-specific values. These are indicated by following the keyword with a forward slash and an appId value (that is, the same appId value that is used when the Session object is instantiated). When a value is needed from the configuration file, the CS Java Connector first looks for an appId-specific entry. If not found, the generic entry is used. The following example is an appId-specific entry (for appId "goosebump") followed by the generic entry:

  ```
  CredentialsProtection/goosebump = Clear
  CredentialsProtection = Symmetric
  ```

- Your application can use the Session object's methods to explicitly override the `CredentialsProtection` settings in WcmApiConfig.properties. For details, see the com.filenet.wcm.api.Session topic in the *CS Java Toolit Javadoc*.
- Your application can call `Session.setConfiguration` method, passing an InputStream object that contains configuration information. This method should be called immediately after the Session object is created. (Note that if the configuration file has already been read for the current session, a call to the `Session.setConfiguration` is ignored.)
- If you use the WcmApiConfig.properties settings in a production environment, keep in mind that changes that a site administrator makes to WcmApiConfig.properties will affect your application.

Locating the Configuration File

The CS Java Connector locates the configuration file in the following sequential manner:

- First, the CS Java Connector looks for a Java system property called "filenet.wcmapiconfig". If found (and it is not an empty string), the CS Java Connector uses its value as the full and complete location of the configuration file and will look no further even if the file cannot be found or read.
- If the filenet.wcmapiconfig Java system property does not exist or is an empty string, the CS Java Connector looks for a file named WcmApiConfig.properties in each of the following directories in turn:
  - the current directory (Java system property "user.dir")
  - the user's home directory (Java system property "user.home")
  - the Java JVM home directory (Java system property "java.home")
- If the WcmApiConfig. properties file is not found through any of the above mechanisms, the CS Java Connector tries to locate it according to the search scheme described in the Java class ResourceBundle. In practice, this means that the CS Java Connector looks for the file "WcmApiConfig.properties" along the ClassPath. The file can be either in a directory or in a JAR file (as long as the directory or JAR file is on the ClassPath).

Setting Up Symmetric Encryption

The following procedure describes how to create a crypto key file that uses the Blowfish encryption algorithm.

Note: In the CS Java Connector, encrypted tokens are not passed to Content Services, but can be used in other security mechanisms used in your environment. For more information, see “Session Authentication” on page 31.

To set up symmetric encryption:

1. Make sure that you have the following files installed and in your Java CLASSPATH:
   - jce1_2_1.jar
   - sunjce_provider.jar
   - local_policy.jar
   - US_export_policy.jar
   - javaapi.jar
   - soap.jar
2. Enter the following command:

   ```
   java com.filenet.wcm.util.MakeCryptoKeys -n numberOfKeys -s bitLength > fileName
   ```

   where:

   -n *numberOfKeys* is the number of distinct keys to create and store in the key file. At runtime, when a key is needed, one is randomly selected from among those present in the crypto key file. Multiple keys allow you to take a key out of service without having to be completely in sync on both ends. For example, you can remove key #2 the server where the CS Java Connector is running, wait a while (so anybody using key #2 can still authenticate on the Content Services server), and then copy the new crypto key file to the Content Services server. This parameter has no default, but we recommend three.

-s *bitLength* is the length of the keys. This argument is optional and 448 is the default size. Key size must be a multiple of 8, and range from 32 to 448 inclusive.

fileName is the name of the ASCII text file to which the key output is redirected. We recommend that you include ".properties" for the file extension since the crypto key file is read as a Java PropertyResourceBundle.

Note: There may be a delay of several seconds when running this program.

3. Specify the symmetric encryption setting in WcmApiConfig.properties.

   a. Set the `CredentialProtection` entry to "Symmetric". (Alternatively, your application can set symmetric encryption when it creates the session.

   b. Set the `CryptoKeyFile` entry to the name and path of the crypto key file that you created. You can specify the absolute path.

      Note: Because WcmApiConfig.properties is read as a Java PropertyResourceBundle, you must use double backslashes in the file pathname, for example, `c:\\contentAccess\\config\\CryptoKeyFile.properties`.

The CryptoKeyFile.properties file follows the syntax rules for a Java PropertyResourceBundle. The CS Java Connector locates the CryptoKeyFile.properties file in the following sequential manner:

- If the value contains either a forward slash (/) or backslash (\) character, the value is taken as the full and complete path name to the file. (As a reminder, backslashes must be doubled in the configuration file.)
- Else, try to open "CryptoKeyFile" according to the search scheme described in the Java class ResourceBundle. In practice, this means the CS Java Connector looks for the file "CryptoKeyFile.properties" along the ClassPath.

Setting Up Your Development Environment

To develop CS Java Connector applications, specify in your project's class search path the JARs and directories listed in "Windows: Setting Environment Variables" on page 12, or in "Solaris: Setting Environment Variables" on page 13 or in "HP-UX: Setting Environment Variables" on page 14.

In addition, import the following classes and packages:

- com.filenet.wcm.api.*;
- com.filenet.Panagon.ObjectFactory
- com.filenet.Panagon.Session
- org.w3c.dom.NodeList
- org.w3c.dom.Node
- org.apache.xerces.parsers.DOMParser
- org.xml.sax.InputSource

Setting Cache Directory Structure

The CS Java Connector will create a cache directory structure which defaults to the current working directory. The cache directory structure is as follows:

.\CS-BASE-CACHE – The base cache directory. This location is configurable within the application, as explained below.

.\CS-BASE-CACHE\CS-UPLOAD – An upload cache where content to be uploaded to Content Services is temporarily stored, that is, during a `Document.checkin(...)` operation.

.\CS-BASE-CACHE\CS-CONTENT – The location where content that has been retrieved from Content Services is stored, that is, during a `Document.getContent()` operation.

.\CS-BASE-CACHE\CS-TEMP – The Content Services temp location.

Note that you can retrieve the default location (current working directory) using the following call:

```
System.getSystemProperty("user.dir");
```

You may configure the location of the CS-BASE-CACHE directory by setting a custom Java System property called "com.filenet.Panagon.tempDir" within your application. For example, you would include the following statement in the initialization phase of your application:

```
System.setProperty("com.filenet.Panagon.tempDir", "C:\\myCache");
```

In this example, your application would use "C:\myCache\CS-BASE-CACHE" as it's base cache location. The cache subdirectories would be nested below the base cache location.

Using CS Java Connector in a Web Application

The CS Java Connector v.3.0 has been certified to run on the BEA WebLogic v7.0 SP5, BEA WebLogic v8.1 SP3, IBM Business Integration Server Foundation v5.1 and Sun Java System Application Server Standard Edition v7.0 SP2004Q2 application server. If you use the CS Java Connector in a Web application, declare the CS Java Connector resources in application server startup script. Depending on your platform, see “Windows: Setting Environment Variables” on page 12, or “Solaris: Setting Environment Variables” on page 13 or “HP-UX: Setting Environment Variables” on page 14.

Uninstalling the CS Java Connector

Follow the procedure applicable to your platform.

- Windows 2000 or Windows 2003: In the Control Panel, Add/Remove Programs, select FileNet CS Java Connector v3.0, and choose Remove.

  Alternatively, go to the .\FileNet\CS Java Connector_uninst directory, and launch filenet_cs_java_connector_uninstaller.exe.

- Sun Solaris version 8/9 or HP-UX 11i: Go to the ./FileNet/CS Java Connector/_uninst directory, and launch filenet_cs_java_connector_uninstaller.bin.

  Alternatively, go to the ./FileNet/CS Java Connector/_uninst directory, and run command “java -jar cs_java_connector_uninstall.jar”.

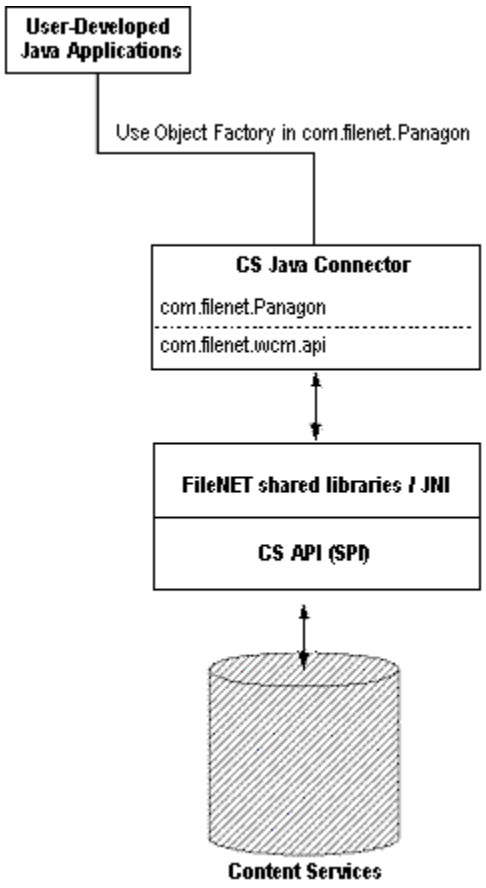
Getting Started

The CS Java Connector is a platform-independent Connector for building Web applications or thick clients that access and manipulates objects stored on Content Services. This section orients you to the CS Java Connector and introduces key concepts and components. It consists of the following topics:

Architecture

As shown in Figure 1, the CS Java Connector consists of Java-based and C++ Business Objects, and the Java Native Interface (JNI) between them.

Figure 1: Content Services Java Connector Architecture



The Java-based components are platform-neutral. For specific libraries -- referred to as object stores in the CS Java Connector, they can also provide limited, transparent, caching of object properties and property and class descriptions. The Connector provides a Java-based Session object, which transparently handles authentication, and leverages the Apache LOG4J framework for logging.

Java objects call into the CS API, a C-language API also referred to as SPI. (The CS API is documented in the *Content Services API Manual*.)

CS Java Connector Packages

The CS Java Connector consists of two Java packages:

- **com.filenet.wcm.api** contains the base classes and interfaces from which many of the Panagon modules derive. This package is contained in **javaapi.jar**.
- **com.filenet.Panagon** contains the interfaces and classes for accessing Content Services objects such as ObjectStore (library), Folder, and Document. It includes the ObjectFactory class for creating new objects and empty collection objects through a set of static methods. This package is contained in **panagon.jar**.

Most of the Panagon interfaces define Content Services properties and states.They extend the PanagonObject interface, which defines the common properties of object name (`idmName`) and unique identifier (`idmId`). In addition to defining properties and states, the following Panagon interfaces extend the functionality of their superinterfaces:

- com.filenet.Panagon.Session allows you to set the group for the logged-on user and set the directory location of downloaded files.
- com.filenet.Panagon.ObjectStore includes methods for getting the property description for one or more specified properties.
- com.filenet.Panagon.Document includes a method for retrieving the ClassDescription object for the Document.

The com.filenet.Panagon package also includes a set of exception classes, all of which derive from the base com.filenet.wcm.api.BaseRuntimeException class.

For details on the CS Java Connector packages, see the *CS Java Connector v3.0 Javadoc*.

Creating Objects

To access the CS Java Connector, you start with the ObjectFactory, by creating a Session object and other high-level objects (Objectstore or EntireNetwork), as discussed in "High-Level Objects" on page 24. You can also create the following objects with the ObjectFactory: Property, PropertyDescription, Permission, Search, and Value.

Objects created with the ObjectFactory are not populated with actual data from the object store, but rather get default values (if any exist) from the metadata described in the class description or with data you provide in the parameters to the method. Objects created with the ObjectFactory class have no affiliation with any object on the Content Services server until you take some action that forces a round-trip to the server. It is possible to populate these objects with invalid data for your object store. In many cases, doing so will not cause an exception to be thrown until the object is passed to a method.

You can also use the ObjectFactory to create empty collections of the following objects: ClassDescriptions, ObjectStores, Permissions, Properties, PropertyDescriptions, and Values. (The ObjectFactory also includes a method to create a PropertyDescriptions collection with elements.) Once created, you can populate an empty collection object before passing it as an argument to a method.

Use the ObjectFactory to create an object or to create an empty collection object for the purpose of passing it to a method that takes such a collection as a parameter.

You can create new Document and Folder objects and persist them to the Content Services server. To create a new Document, call the `createObject()` method on an ObjectStore; to create a new Folder, call the `addSubFolder()` method on a Folder.

You can also return objects from methods. Two key methods for returning objects are `getObject()`, called on an ObjectStore, and `getProperties()`, called on objects that implement the ReadableMetaData interface. You can use the `getObject()` method to retrieve the following objects: Document, Folder, ClassDescription, and VersionSeries. Note that instances of these objects include metadata about the class on which the objects are based.

The `getProperties()` method returns all of the properties of an object in a collection. Note that results are cached, eliminating network roundtrips to the server on subsequent calls.

High-Level Objects

The highest level objects in the CS Java Connector are the Session, EntireNetwork, and ObjectStore objects. The Session object carries user credentials, as well as security and configuration information. The EntireNetwork object represents the object stores on Content Services. The ObjectStore object provides access to the contents of a particular object store. An object store -- called a library in Content Services legacy documentation -- is a repository for a group of objects and functionality that a client application can access.

Session Object

Before you can communicate with a Content Services server, you must establish a session by creating Session object. A Session object holds user credentials and configuration information (which includes the encryption scheme). The Content Services server authenticates this information before providing access to its resources.

You can create a Session object by calling one of the three signature forms of `getSession()` on the `ObjectFactory` class. You supply all parameters to the base form of the method. The parameters specify the configuration information and the values to be used when building the credentials token. The following example creates a new Session object for the "MyAppName" application:

Listing 1: Creating a Session object

```
Session sess = ObjectFactory.getSession("MyAppName",
    com.filenet.wcm.api.Session.CLEAR, "myacct", "passwd", null);
```

For the two convenience forms of `getSession()`, you supply one or two parameters; the method signature supplies Java `null` for the remaining parameters. For more information about sessions, see “WcmApiConfig.properties” on page 16 and “Session Authentication” on page 31.

To create a EntireNetwork or ObjectStore object, you must explicitly supply a reference to a Session object. For other objects, a reference to the Session object is inherited during the object instantiation process.

EntireNetwork Object

Representing the overall structure of the object stores defined in the Content Services server, the EntireNetwork object provides access to object stores existing on one or more Content Services servers. As described in “EntireNetwork.properties” on page 16, an object store must be listed in the configuration file in order for an EntireNetwork object to recognize it.

To retrieve an EntireNetwork object, call the `getEntireNetwork()` method on the ObjectFactory and pass in a Session object, as shown in the following code fragment. You can then return all of the object

stores in a collection. Note that the logon does not occur until a method is called on the ObjectStore object that requires a roundtrip to the Content Services server.

Listing 2: Creating an EntireNetwork object

```
...
Session sess = com.filenet.Panagon.ObjectFactory.getSession("myApp",
  com.filenet.wcm.api.Session.CLEAR, "joe", "anchor", null);
EntireNetwork netw = com.filenet.Panagon.ObjectFactory.getEntireNetwork(sess);
ObjectStores   objStores    = en.getObjectStores();
...
```

Object Store

Objects are affiliated with, and persisted to, an object store (library). To work with objects in a Content Services object store, you must instantiate an ObjectStore object. For example, to begin working with documents, you must instantiate an ObjectStore object for the object store in which the documents reside.

As shown in Listing 2, you can instantiate an ObjectStore object indirectly by calling `getObjectStores()` on the EntireNetwork object. To directly instantiate an object store, call the `getObjectStore()` method on the ObjectFactory and pass in a Session object, as shown in the following code fragment.

Listing 3: Creating an ObjectStore object

```
...
Session sess = com.filenet.Panagon.ObjectFactory.getSession("myApp",
  com.filenet.wcm.api.Session.CLEAR, "joe", "anchor", null);
String ObjectStoreId = "pin2ms^Pinta";
ObjectStore objStore =
com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId, sess);
...
```

Note that the logon does not occur until a method is called on the ObjectStore object. Each object store requires a separate logon.

When you have an ObjectStore object, you can call its `getObject()` method to create other objects. For more information, refer to “Object Stores” on page 43.

BaseObjects Collection

With the exception of the Properties collection, the collection classes in the CS Java Connector derive from the BaseObjects collection, which can contain elements of various object types. For example, the following call on a Folder object returns a BaseObjects collection containing Folder objects and Document objects.

```
BaseObjects collObjs = oFolder.getContainees();
```

Unlike the BaseObjects collection, derived collection objects are strongly-typed.

Strongly-typed Collection Objects

Strongly-typed collection objects contain only elements whose object type is directly related to the object type of the collection object. That is, an ObjectStores collection contains only ObjectStore objects, a Documents collection object contains only Document objects, and so on.

Although the BaseObjects collection is not inherently strongly-typed, you can retrieve a BaseObjects collection of a specific object type, then cast that returned collection into a strongly-typed collection. For example, you can specify a particular object type to return in the `getContainees()` method of a Folder object. The following statement returns a BaseObjects collection of only the Document objects that exist in the folder:

```
BaseObjects collObjs = oFolder.getContainees(BaseObject.TYPE_DOCUMENT);
```

Another notable point about the `Folder.getContainees()` method is that it returns a populated collection object. Other examples of methods that return populated collections are `EntireNetwork.getObjectStores()` and `ObjectStore.getPropertyDescriptions()`.

As mentioned previously, you can call methods on the ObjectFactory that create empty collections, which you can then pass to a method that can takes an empty collection, or to which you can add elements of the appropriate type. With the exception of the Properties collection object, you can also use methods (such as add, remove, and clear) inherited from java.util.Collection and java.util.List interfaces to manipulate elements in all CS Java Connector collection objects.

The following code fragment creates a new empty Permissions collection and adds a Permission object to the collection. It also explicitly persists a Permissions collection object with a `setPermissions()` method call. Note that the Permissions collection object is the only persistable collection object; you cannot persist other collection objects to the Content Services server.

Listing 4: Adding an object to an empty collection

```
...
//Create a new Permissions collection
Permissions perms = ObjectFactory.getPermissions();
//Create a new Permission object and add it to the collection
Permission perm = ObjectFactory.getPermission(idmAccessViewer, Permission.TYPE_ALLOW,
    "Everyone", BaseObject.TYPE_GROUP);
//Adds permission object to the collection
perms.add(perm);
doc.setPermissions(perms);
...
```

None of the CS Java Connector collection objects are set as read-only (un-modifiable) collections. Therefore, although there are some collections where it does not make sense to add or remove objects from the collection (such as the collection of contained objects returned in a call to `Folder.getContainees)`, the CS Java Connector will not throw an exception if you attempt to do this.

ReadableMetadataObjects Collection

For all strongly-typed CS Java Connector collection interfaces (with the exception of the Properties interface), you can use methods inherited from the ReadableMetadataObjects interface to return one or more objects for a given property, relational operator, and value. These methods are useful when you want

to limit the number of objects returned or when you want to get a specific object based on the object's property. For example, you could use this technique to retrieve items that you want to delete. The following code fragment filters an existing collection to return a new collection that contains only objects for which the value of the `idmDateAdded` property is less than the current date:

Listing 5: Filtering out selected items from a collection

```
...
// Get a new collection containing expired objects
Documents collExpiredObjs = (Documents)collDocObjs.filterByProperty(
                             com.filenet.Panagon.Document.idmDateAdded,
                             ReadableMetadataObjects.IS_LESS,
                             new Date());
...
```

You can also use this filtering capability to return a single object from an existing collection. The following code fragment returns the single element from a Documents collection for which the name is "TechnicalUpdate":

Listing 6: Finding specific items in a collection

```
...
Document d = collDocObjs.findByProperty(com.filenet.Panagon.Document.idmName,
                                        ReadableMetadataObjects.IS_EQUAL,
                                        "TechnicalUpdate");
...
```

BaseObject Object

Most interfaces in the CS Java Connector derive from BaseObject, which defines constants that specify object types, and methods that return basic information about objects. A number of methods in the CS Java Connector take object parameters specified by ID, type, or name. BaseObject methods give you this information.

The BaseObject interface also defines convenience methods for returning property values. These methods only require the symbolic name of a property, for example `idmAddedByGroup` or `idmComment`.

Constants for Specifying Object Types

The BaseObject interface defines constants (fields) for all object types used within the Content Services server, including some not used in the CS Java Connector. Use these constants in methods that require an object type parameter. One such method is the ObjectStore's `getObject( ) method,` for example `os.getObject(BaseObject.TYPE_DOCUMENT).`

Retrieving Object Information

BaseObject object includes methods to retrieve the following information about an object:

- **ID.** The `BaseObject.getID()` retrieves an object's unique identifier. For more information about IDs, see "Object IDs" on page 28.

- **Type.** The `BaseObject.getObjectType()` method returns an integer indicating an object's type, for example, 1 for Document, 2 for Folder, and 4 for ObjectStore. These integers are defined as constants in the interface (see "Constants for Specifying Object Types" on page 27).

  When working with methods that require you to pass an object type as an argument, use `getObjectType()`. For example, one form of the `ObjectStore.getClassDescriptions()` `method passes` an integer array of BaseObject objects.To retrieve class descriptions of unknown objects , call `getObjectType() on each object,` add the returned integers to an integer array, , and pass the array in the `ObjectStore.getClassDescriptions()` call.

- **Name**. The `BaseObject.getName()` method returns the value of the `idmName` property for the object, defined in the PanagonObject interface.

- **Class Identifier.** The `BaseObject.getClassId()` method returns a string identifying an object's class. For a Document object, the class identifier is the name of the Document object's class, for example "General". The class identifier for a Folder, ObjectStore, and StoredSearch object is the class type, that is, _BASE_FOLDER, _BASE_CATALOG, and _BASE_STORED_SEARCH, respectively.

  For other objects, `BaseObject.getClassId()` returns a GUID identifying an object's class. GUIDs are defined as fields in the com.filenet.wcm.api.ClassDescription interface. See the *CS Java Connector v3.0 Javadoc* for more information.

- **ObjectStore ID.**The `BaseObject.getObjectStoreId()` method returns the object store location in which the BaseObject object is stored. The returned string is a concatenation of the library name, the carat character, and host machine name, for example: `Aslan^Narnia`.

Note that these BaseObject methods are especially useful for identifying objects in a BaseObjects collection. For example, Folder.`getContainees()` returns a BaseObjects collection, with each element an object (such as a subfolder or document) contained within the parent folder. You can iterate through the collection and use the applicable BaseObject method to identify each object.

Object IDs

In the CS Java Connector, you need to specify IDs to retrieve objects that implement the GettableObject interface: ClassDescription, Document, Folder, and VersionSeries. These objects are retreived with `ObjectStore.getObject()`, which takes the object ID as one of its parameters. To retrieve an object store, you need to specify the ID in `ObjectFactory.getObjectStore()`.

An object ID is the unique identifier for the object, and is stored in the `idmId` property of an object. This property and `idmName` (descriptive name of the object) are defined in the PanagonObject interface, from which most of the other interfaces in the Panagon package derive.

The following objects extend PanagonObject and can therefore be referenced by ID:

- ObjectStore, with an ID that's a concatenation of the host machine name and library name as follows: <library>^<host>.
- Document, with an ID that matches the ID on Content Services, such as "003677005". (To get a VersionSeries object, you specify the ID of a document for which you want version information.)
- Folder, with an ID that matches the ID on Content Services, such as "1013800640".
- PropertyDescription, with an ID that is the symbolic name of a property defined in a com.filenet.Panagon interface, such as `idmAddedByGroup` or `idmComment`.
- ClassDescription, with an ID that's the unique document type name (the value of the `idmDocType` property for a Document object).

- ChoiceList, with an ID that matches the ID on Content Services.

The BaseObject object, from which PanagonObject derives, consists of methods to retrieve an object's ID as well as other basic information about an object. See "BaseObject Object" on page 27.

Properties and Metadata

All persisted objects contain properties. In the com.filenet.Panagon interfaces, the properties are defined as fields using symbolic names, for example, `IdmId`, `IdmName`, and `IdmAccessLevel`. When instantiated, an object's properties are stored in the object's fields. Properties are cached locally in Java to enhance Connector performance.

A property can be represented by a PropertyDescription object, allowing you to obtain metadata for each property of a class or for an object like a document or folder. Individual properties of a PropertyDescription object are individual items of metadata with symbolic names, such as `idmPropertyType`, `idmLabel`, `idmDefault`, and `idmTypeId` -- all of which are defined in the com.filenet.Panagon.PropertyDescription interface.

Note: The CS Java Connector returns PropertyDescription objects only for properties that have been configured on Content Services with the CS Admin Tools.

In addition, you can set properties on many objects. Methods that set properties take a Properties collection as a parameter and attempt to persist the given values to the object store.

For more information, see "Properties" on page 52.

Using XML Forms of Methods

Many interfaces in the CS Java Connector include both object and XML forms of their methods. The XML forms return data as XML-formatted strings. You can call the XML versions of methods to retrieve XML suitable for mapping data to fields in a user interface.

The following listing shows the XML returned by the `ObjectStore.getPropertiesXML(`*`properties`*`)`, where *properties* is a String array of the following specified properties: `idmName`, `idmAddedByUser`, `idmComment`, and `idmDateAdded`.

Listing 7: XML returned by getPropertiesXML( ) method

```
<response xmlns="http://filenet.com/namespaces/Panagon/apps/1.0">
  <objectset>
    <objectstore>
      <objectstore>
        <id>pin2ms^Pinta</id>
        <name>Pinta development test machine</name>
      </objectstore>
      <properties>
        <property>
          <symname>idmName</symname>
          <name>System Name</name>
          <datatype>8</datatype>
          <value>pin2ms</value>
        </property>
        <property>
          <symname>idmAddedByUser</symname>
          <name>Installer</name>
          <datatype>8</datatype>
          <value>bp</value>
        </property>
        <property>
          <symname>idmComment</symname>
          <name>Comment</name>
          <datatype>8</datatype>
          <value>FileNet IDM Content Services</value>
        </property>
        <property>
          <symname>idmDateAdded</symname>
          <name>Installation Date</name>
          <datatype>3</datatype>
          <value>Tue Dec 05 14:46:41 PST 2000</value>
        </property>
 </properties>
      <permissions/>
    </objectstore>
    <count>1</count>
  </objectset>
</response>
```

Security

Authentication is the process by which the logon credentials of a user are validated against the security established on the Content Services server. Authentication hard-coded must occur before resources can be accessed on the Content Services.

The point at which logon occurs on an object store depends on how you instantiate the ObjectStore object:

- If you use objectFactory.`getObjectStore()`, `logon occurs immediately.`
- If you use `EntireNetwork.getObjectStores()` to return an ObjectStores collection, logon does not occur until a method is called on an ObjectStore object that requires a roundtrip to the Content Services server.

After a successful logon, a user's access to objects is controlled by the access rights assigned to the individual user, and by the access rights assigned to any group of which that user is a member. Access rights to individual objects are checked each time the user attempts to access an object from an object store.

This section covers the following security topics:

- "Session Authentication" on page 31.
- "Access Rights to Documents and Folders" on page 33.

Session Authentication

When a user attempts to log on to Content Services, a Session object is instantiated to hold user credentials in a token. The credentials token is built from the information you supply to the `getSession()` method on the ObjectFactory class. This method has three signature forms: a base form and two convenience forms. All forms require the application ID parameter. The parameter values in the base form of the method map to the keyword/value pairs of the credentials token. When you use the convenience forms of `getSession()`, optional parameters for which you don't supply a value default to Java null.

The following statement creates a Session object with an application identifier for the MyTestApp application and a protection level of null. (You can also specify the `Session.DEFAULT` constant for this parameter with the same result; that is, the CS Java Connector reads the default protection level from the WcmApiConfig.properties file.) This method call is supplying only the userid ("tester") of the remaining three parameters; password and domain are null.

```
Session sess = ObjectFactory.getSession("com.example.apps.MyTestApp", null, "tester",
null, null);
```

In the CS Java Connector, encrypted tokens are not passed to Content Services, but can be used in other security mechanisms used in your environment. You can retrieve an encrypted or unencrypted token and extract information from it with the Session object's `getToken()` and `fromToken(`*`tokenString`*`)` methods. The `getToken()` method returns a String representing the token. The `fromToken(`*`tokenString`*`)` method extracts credentials information from the token. For example, you could extract the userID so that it could be validated as part of another security mechanism employed for your environment. The `fromToken()` method returns a Java HashMap object containing the constituent keyword/value pairs from the specified credentials token. If the credentials token uses `Session.SYMMETRIC` encryption, the method automatically attempts decryption of the keyword/value pairs. If decryption fails, the method throws an exception with a message indicating the decryption failure.

Use the HashMap.get() method to retrieve the data you want, using `Session.USERID` and related constants as keys.

Credentials Token Components

The credentials token comprises several components, described below. Except for locale, you can set the value for the components described below by calling the `getSession` method on the ObjectFactory interface. The application ID, user ID, and password are required. (To set the locale, call the `setLocale` method on the Session interface.) Some components indicate configuration information; other components represent information about the user who is attempting to gain access to resources on the Content Services server.

Application ID

The application ID, passed in the `appId` parameter of an `ObjectFactory.getSession()` method call, represents the unique name of the application to which a token applies.

Your application program is responsible for ensuring that the String is unique. You can ensure uniqueness by including the fully qualified name of the Java class that contains the application's main() method, for example, "`com.example.apps.MyTestApp`". The String cannot be null and cannot contain a forward slash character (/).

The application ID is used to conditionally select values from the site configuration file, WcmApiConfig.properties, and is passed in every request to the Content Services server.

Credentials Protection Level

The credentials protection level is passed in the `credTag` parameter of an `ObjectFactory.getSession` method call. If you specify either Java null or the `Session.DEFAULT` constant for this parameter, the method uses the site-wide protection level, which is specified in the WcmApiConfig.properties file. To explicitly specify the protection level, supply a String containing the appropriate Session constant (`Session.CLEAR` or `Session.SYMMETRIC`) or case-insensitive string (for example, "clear" or "Clear"). A valid value in this parameter overrides any credentials protection keyword/value pair specified in the WcmApiConfig.properties file.

User ID and Password

The user ID and password are required String objects that represents the identifier of a user (which can be a person or a program) attempting to gain access to resources on the Content Services server. The Content Services server validates the user ID and password against the account on the server. Logon will fail if one or more parameters you provide are found to be invalid, or you do not supply one or more of the parameters that are required by the account on the Content Services server. For example, if you supply the user ID but no password, logon will fail.

If you do not initially set the user ID or password when you create the Session object, you can specify that information by calling `Session.setUserid()` or `Session.setPassword()` immediately after creating the Session object and before making any CS Java Connector calls that cause a round trip to the Content Services server.

Domain

The CS Java Connector does not support domain, and will ignore the value if you specify one.

Locale

A locale represents a specific geographical, political, or cultural region and determines the default language in which error messages and other system information will be displayed. The default locale is specified during the installation of the Java virtual machine. If you do not set a specific locale for the session, the CS Java Connector uses the default locale established by the operating system. However, you can change the locale for a particular session by calling `Session.setLocale()`. Create a Java Locale object, specifying a language and country, and pass the Locale object as a parameter in your method call. Operations on the Content Services server are performed under the specified Locale, which presents the opportunity for localizing diagnostics and other information returned.

Note: Setting the Locale for the Session object does not affect the Locale under which the Java program is running. You would still need to use the standard `Locale.setDefault` method to make that change if appropriate to your application.

Access Rights to Documents and Folders

In a CS object store (library), every object (such as Document and Folder) has an Access Control List (ACL) that contains access for Groups and Users. Each Group and User in the ACL has one of five possible access rights to an object, from no access to total, administrative access. Using the CS Java Connector, you control security access that a User or Group has to a CS object with a Permission object. A Permission object represents a security access control element. It consists of a reference to a User or a Group and a specified Access level. Permission objects are contained in Permissions collections, allowing you to iterate through a set of Permission objects and perform actions.

This release of the CS Java Connector allows you to retrieve and set access rights to Document and Folder objects. The access rights for these two objects are explained in this section. For code examples of retrieving and setting access rights for these objects, see “Retrieving and Setting Permissions” on page 35.

Content Services object stores have five levels of access rights: None, Viewer, Author, Owner, and Admin. Each level of access incorporates the rights assigned to all lower access levels; for example, Admin access to an object includes Owner, Author, and Viewer rights as well as special administrative privileges. Also, the relevant access to an object for the user is the higher of their user and group access to that object. The meanings of these access rights depend on the object. General information on these access rights is shown below. For information on access rights for a specific object type, see the following sections:

- “Document Access Rights” on page 34
- “Folder Access Rights” on page 35

The available levels of access rights, from lowest to highest, are:

- idmAccessNone - Allows users no privileges to an object (that is, prohibits read or write access). This level of access can be particularly useful at sites where some users might require even lower access privileges than the typical general users.
- idmAccessViewer - Generally, the ability to view the object and its properties or to make copies of the associated versions.
- idmAccessAuthor - Viewer access rights plus the ability to check out, check in, and copy associated versions and modify property values for the version. In addition, the user might be allowed to modify designated custom property values for the document.
- idmAccessOwner - Author access rights plus the ability to delete documents, modify security, and modify most properties.

- idmAccessAdmin - Owner access rights plus the ability to modify all property values. Active members of the Administrators group are automatically assigned Admin access rights to all properties, even though their names do not appear in any access lists. Users who are not members of the Administrators group can be explicitly assigned Admin access rights to the properties associated with particular objects.

Access to individual properties (for example, the right to view or modify values) is derived from these access rights. For example, a user who has Owner access to a Document object can modify only a specific subset of property values. Thus, the level of access rights is assigned at the object level, but it is applied on a property-by-property basis.

Not every User and Group defined in an object store is represented in every object's Access Control List. If a User or Group is not represented, then CS assumes None access unless the User or Group is an administrator. In this case, the administrator has Admin access to every object.

Document Access Rights

The following table describes the access rights to documents.

Table 6: Access Rights to Documents

This access right	Allows the user or group to
idmAccessNone	Do nothing with this document.
idmAccessViewer	View the document. • Copy associated versions. • Create annotations for this document version.
idmAccessAuthor	Same as idmAccessViewer access plus: • Move the document. • Check out the document. • Check in the document. • Make a document permanent. • Make a document impermanent.
idmAccessOwner	Same as idmAccessAuthor access plus: • Modify the document security. • Modify document properties. • Archive the document. • Delete the document in pre-5.0 CS releases; for CS 5.0 or later releases, the administrator can configure this to Owner or Admin. • Reclaim the document for the archive.
idmAccessAdmin	Same as idmAccessOwner access plus: • Modify all property values. • Modify security.

Folder Access Rights

The following table describes the access rights to folders. The access rights of a folder are independent of any other folder, and of any documents filed in the folder.

Table 7: Access Rights to Folders

This access right	Allows the user or group to
idmAccessNone	Do nothing with this folder.
idmAccessViewer	View the folder. • Copy the folder.
idmAccessAuthor	Same as idmAccessViewer access plus: • Add the folder. • Create sub-folders. • File documents into the folder. • Unfile documents from the folder.
idmAccessOwner	Same as idmAccessAuthor access plus:• Move the folder. • Delete the folder. • Modify folder security. • Modify folder properties.
idmAccessAdmin	Same as idmAccessOwner access plus: • Modify security. • Modify all property values.

Retrieving and Setting Permissions

In this release of the CS Java Connector, you can retrieve and set permissions for Document and Folder objects. This section explains how. For full code samples that illustrate this subject, see “Permission Code Samples” on page 38.

Represented as an object, a permission indicates security access to document or folder for a particular user or group included in the document or folder’s Access Control List (ACL). To retrieve permissions, use the `getPermissions()` method on a Folder or Document object, which retrieves a collection of Permission objects. By iterating through the collection, you can execute the following methods to retrieve information on the Permission object: `getGranteeName`( ) to retrieve the name of the user or group being granted access to the document or folder; `getGranteeType`( ) to check if a grantee is a user or a group; and `getAccess`( ) to retrieve the access rights to the document or folder. To change information on the Permission object, you would use `setGranteeName()`, `setGranteeType()`, and `setAccess()`.

The following code fragment shows how to create Permissions collections for a Document object and a Folder object, and to iterate through each collection and print the security level of each user or group with access to the object.

Listing 8: Retrieving permissions for a folder and a document

```
...
// Retrieve permissions for the specified folder
        Folder fld = (Folder) os.getObject(Folder.TYPE_FOLDER, "988143960" );
// Get Permissions collection for the folder
        Permissions permsFolder = fld.getPermissions();
/* Create iterator object and for each permission on the folder,
    print grantee name, grantee  type (group or user), and access level. */
        Iterator iterator = permsFolder.iterator();
        while(iterator.hasNext())
        {
            Permission perm = (Permission) iterator.next();
            System.out.println("Grantee Name:" + perm.getGranteeName());
            System.out.println("Access type:" + perm.getAccessType());
            if (perm.getGranteeType() == BaseObject.TYPE_USER)
            {
                System.out.println("Grantee type:User");
            }
            else if (perm.getGranteeType() == BaseObject.TYPE_GROUP)
            {
                System.out.println("Grantee type:Group");
            }
            else
            {
                System.out.println("Grantee type:UNKNOWN!!!!!");
            }
       }

// Retrieve permissions for the specified document
        Document doc = (Document) os.getObject(Document.TYPE_DOCUMENT, "003676242" );

// Get Permissions collection for the document
        Permissions permsDoc = doc.getPermissions();

 /* Create iterator object and for each permission on the document,
    print grantee name, grantee  type (group or user), and access level. */

        Iterator docIter = permsDoc.iterator();
        while(docIter.hasNext())
        {
            Permission perm = (Permission) docIter.next();
            System.out.println("Grantee Name:" + perm.getGranteeName());
            System.out.println("Access:" + perm.getAccess());
            if (perm.getGranteeType() == BaseObject.TYPE_USER)
            {
                System.out.println("Grantee type:User");
            }
            else if (perm.getGranteeType() == BaseObject.TYPE_GROUP)
            {
                System.out.println("Grantee type:Group");
            }
            else
            {
                System.out.println("Grantee type:UNKNOWN!!!!!");
            }
        }
```

In the previous code fragment (Listing 8), Permission collections are created by calling `getPermissions()` on a Folder object and a Document object. These collections contain Permission objects, representing existing permissions set on the folder and document. In the next code fragment, the `getPermissions()` method is called on the ObjectFactory, creating an empty collection. A Permission object is added to the collection, and the collection is then set to a Document object. This has the effect of overwriting any permissions that were previously set on the document. With the new permission, all users in the group "Everyone" can view the document's properties and content.

Listing 9: Setting new permissions on a document

```
...
// Create an empty Permissions collection with ObjectFactory
   Permissions perms = ObjectFactory.getPermissions();

// Get a Permission object from ObjectFactory and fill in the access level,
// access type (allow or deny), name of the group, and user type (user or group)
   Permission perm =
     ObjectFactory.getPermission(com.filenet.Panagon.Permission.idmAccessViewer,
                                 Permission.TYPE_ALLOW,
                                 "Everyone",
                                 BaseObject.TYPE_GROUP);

// Add the newly created Permission object to the Permissions collection
   perms.add(perm);

// Set the updated permission on the Document object
   doc.setPermissions(perms);
...
```

Retrieving Groups and Users

The ObjectStore object includes the `getUsers()` and `getGroups()` methods for retrieving Users and Groups collections, which represent all of the users and groups defined on the Content Services server by the system administrator. The ObjectStore includes XML versions of these methods as well.

From Users and Groups collections you can retrieve User and Group objects. Derived from SecurityGrantee, User and Group objects are identified as grantees in a Permissions collection.

With either a User or Group object, you can get the group(s) in which the user or group is a member. And with a Group object, you can get the users in the group.

In the following code fragment, the `getUsers()` method is used to retrieve all of the users in the object store, and a Permission object is created for each user. Then each Permission object is added to a Permissions collection, which is used to set the permissions on a document object.

Listing 10: Creating one permission per user in the object store

```
...
// Create an empty Permissions collection with ObjectFactory
   Permissions perms = ObjectFactory.getPermissions();

// Get all users in the object store and iterate through the collection
   Users users = os.getUsers();
   Iterator iterator = users.iterator();
   while(iterator.hasNext())
   {
     User usr = (User) iterator.next();
     System.out.println("\tId:" + usr.getId() + "\t\tName:" + usr.getName());

// For each user, create a Permission object, then add it to the Permissions collection
   Permission perm = com.filenet.Panagon.ObjectFactory.getPermission(
          com.filenet.Panagon.Permission.idmAccessOwner,
          Permission.TYPE_ALLOW,
          usr);
   perms.add(perm);
   }

// Set the permissions on the Document object
   doc.setPermissions(perms);
...
```

Permission Code Samples

This section provides runnable code samples that demonstrate ways of working with the Permissions and Permission objects. For these samples to work in your development environment, you must modify hard-coded references to servers, object stores, documents, and other resources stored on a Content Services server. You must also create a Session object to log into an object store.

The following code samples are included:

- Listing 11 adds a new permission to an existing set of permissions on a folder.
- Listing 12 changes a permission on a folder.
- Listing 13 gets user information.
- Listing 14 gets group information.

Listing 11: permissionAdd code sample

```
// Adds a new permission to an existing set of permissions on a folder
public void permissionAdd()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
     sess);

   //Get the folder
   Folder fld     = (Folder) os.getObject(Folder.TYPE_FOLDER, "1013800642" );
   System.out.println("Folder name: "+fld.getName());

   //Get the existing permissions on the folder
   Permissions perms = fld.getPermissions();
   // Create a new permission for acctg group
   Permission newPerm = ObjectFactory.getPermission(
      com.filenet.Panagon.Permission.idmAccessViewer,
      Permission.TYPE_ALLOW,
      "acctg",
      BaseObject.TYPE_GROUP);

    //Add the new permission to the collection
    perms.add(newPerm);

    // Set the expanded permission collection on the folder
    fld.setPermissions(perms);
}
```

Listing 12: permissionChange code sample

```
// Changes the access level of a one permission a folder. The grantee name of
// the permission is "General Users".
public void permissionChange()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
     sess);

   //Get the folder
   Folder fld     = (Folder) os.getObject(Folder.TYPE_FOLDER, "1013800642" );
   System.out.println("Folder name: "+fld.getName());

   //Get the existing permissions on the folder
   Permissions perms = fld.getPermissions();

   Iterator iterator = perms.iterator();
   while(iterator.hasNext())
   {
       Permission perm = (Permission) iterator.next();

       // If grantee name is "General Users" group, change access level
       // from Viewer to Author.
       if (perm.getGranteeName().compareTo("General Users")==0
       {
           System.out.println("Updating access level for General Users");
           perm.setAccess(com.filenet.Panagon.Permission.idmAccessAuthor);
       }
   }
}
```

Listing 13: getUsers code sample

```
// Gets user information using different ObjectStore methods.
// The getUsers() method returns a collection of User objects
// and prints the name of each user and the groups that each user
// belongs to. The getUsersXML() method returns Users in XML format
// and writes the string to a file.
public void getUsers()
{
     String ObjectStoreId = "pin2ms^Pinta";
     ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(
                       ObjectStoreId, sess);
     // Get all users in the object store
     Users users = os.getUsers();
     System.out.println("Found " + users.size() + " users");

     // Iterate through users and identify.
     Iterator iterator = users.iterator();
     while(iterator.hasNext())
     {
        User usr = (User) iterator.next();
        System.out.println("\tId:" + usr.getId() + "\t\tName:" + usr.getName());

        // Now get the groups that this user belongs to.
        Groups grps = usr.getParentGroups();

        // Iterate through groups that user belongs to and identify.
        Iterator grpIter = grps.iterator();

        System.out.println("\t*** Groups that this user belongs to ***");
        while(grpIter.hasNext())
        {
           Group grp = (Group) grpIter.next();
           System.out.println("\t\tId:" + grp.getId() + "\t\tName:" + grp.getName());
        }

           System.out.println("\t*** End of Groups that this user belongs to ***");
     }

     System.out.println("***** getUsersXML *****");
     // Get XML string of all Users defined in the object store.
     String xml = os.getUsersXML();

     // Send to file.
     String fileName = "getUsersXML.xml";
     try
     {
        java.io.OutputStream fos = new java.io.FileOutputStream(fileName);
        java.io.Writer w = new java.io.BufferedWriter(new java.io.OutputStreamWriter
                                                      (fos, "Unicode"));
        w.write(xml);
        w.close();
     }
     catch(Exception e)
     {
        System.out.println(e.toString());
     }
     System.out.println("ObjectStore getUsersXML sent to file:" + fileName);
}
```

Listing 14: getGroups code sample

```
// Gets group information using different ObjectStore methods.
// The getGroups() method returns a collection of Group objects
// and prints the name of each group and the users in each group.
// The getGroupsXML() method returns Groups in XML format
// and writes the string to a file.
public void getGroups()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
                     sess);

   // Get all groups in the object store
   Groups grps = os.getGroups();
   System.out.println("Found " + grps.size() + " groups");

   // Iterate through users and identify.
   Iterator iterator = grps.iterator();
   while(iterator.hasNext())
   {
       Group grp = (Group) iterator.next();
       System.out.println("\tId:" + grp.getId() + "\t\tName:" + grp.getName());
       // Now get the users that belong to this group
       Users usrs = grp.getUsers();
       // Iterate through the users in each group and identify.
       Iterator usrIter = usrs.iterator();

       System.out.println("\t*** Users that belong to this group ***");
       while(usrIter.hasNext())
       {
          User usr = (User) usrIter.next();
           System.out.println("\t\tId:" + usr.getId() + "\t\tName:" + usr.getName());
       }

           System.out.println("\t*** End of Users that belong to this group ***");
   }

   System.out.println("***** getGroupsXML *****");
   // Get XML string of all Groups defined in the object store.
   String xml = os.getGroupsXML();
   // Send to file.
   String fileName = "getGroupsXML.xml";
   try
   {
        java.io.OutputStream fos = new java.io.FileOutputStream(fileName);
        java.io.Writer w = new java.io.BufferedWriter(new java.io.OutputStreamWriter
                                                      (fos, "Unicode"));
        w.write(xml);
        w.close();
   }
   catch(Exception e)
   {
        System.out.println(e.toString());
   }

   System.out.println("ObjectStore getGroupsXML sent to file:" + fileName);
}
```

Object Stores

An object store is a library -- a location where objects are stored and retrieved on a Content Services server. Each object store is represented as an instance of an ObjectStore object. A set of ObjectStore objects is represented by an ObjectStores collection object. The main use of an ObjectStores collection is for displaying a list of available object stores to the user, or for getting a reference to an object store within the collection.

An ObjectStore object possesses the following characteristics:

- You can instantiate an ObjectStore directly via the ObjectFactory class or indirectly via the EntireNetwork class. See “Instantiating an ObjectStore Object” on page 43.

  Note: You cannot create an object store from the CS Java Connector.

- An ObjectStore provides the `getObject()` method for retrieving any persisted objects that implement the GettableObject interface, such as Document, Folder, and ClassDescription objects. See “Getting an Object from the ObjectStore” on page 44.

- An ObjectStore provides the `createObject()` method for creating a Document object. See “Creating a Document Object” on page 86. It also provides a method for retrieving a collection of Document objects checked out by the logged-on user. See “Retrieving Checked-Out Documents” on page 101.

- An ObjectStore object provides methods for retrieving property information. See “Retrieving Properties” on page 45. It also provides methods for retrieving choice list information. See “Retrieving Choice Lists” on page 45.

- An ObjectStore provides methods for accessing library metadata. See “Accessing Metadata” on page 46.

- An ObjectStore object includes the `support(...)` method for determining whether a particular library capability is supported or not. See “Determining ObjectStore Support” on page 47.

- You can retrieve an object store's root folder and a string representation of its subfolders (which are the top level folders in the object store). For more information, refer to “Containment” on page 71.

- An ObjectStore provides methods for returning collections of User and Group objects, representing user accounts defined on the Content Services server. User and Group objects comprise grantees in a Permissions collection, which in turn represents the access rights to a Document or Folder object. For more information, refer to “Retrieving Groups and Users” on page 37.

For code samples illustrating the use of the ObjectStore object, see page 47.

Instantiating an ObjectStore Object

You can directly instantiate an ObjectStore object with the ObjectFactory, or you can indirectly instantiate an ObjectStore object through EntireNetwork.

To create a new instance of an ObjectStore object, call `ObjectFactory.getObjectStore()` with the object store name and the Session object parameters. The object store name is a concatenation of the library name and host machine name, as follows:

```
ObjectStore objStore = com.filenet.Panagon.ObjectFactory.getObjectStore
                       ("pin2ms^Pinta", sess);
```

When you use `objectFactory.getObjectStore()`, logon occurs immediately.

You can also use `getObjectStores()` on the EntireNetworkobject, which returns an ObjectStores collection. You can then return an ObjectStore object from the collection.

The following code fragment creates an EntireNetwork object and retrieves an ObjectStores collection. The code iterates through the collection and, by retrieving the value of the `idmName` property for each element in the collection, returns the name of each ObjectStore object.

Listing 15: Retrieving ObjectStores collection from EntireNetwork

```
...
EntireNetwork oNetwork =
     com.filenet.Panagon.ObjectFactory.getEntireNetwork(session);
 ObjectStores objStores = oNetwork.getObjectStores();
 ObjectStore objStore = null;
 String objStoreName = null;

 // Get each object store
 Iterator iterator = objStores.iterator();
 while(iterator.hasNext())
 {
     objStore = (ObjectStore) iterator.next();
     // Get the name of the object store
     objStoreName =
        objStore.getPropertyStringValue(com.filenet.Panagon.PanagonObject.idmName);
 }
```

If you use `EntireNetwork.getObjectStores()` to return an ObjectStores collection, logon and validation do not occur until an ObjectStore method is called that requires a roundtrip to the Content Services server. Each object store requires a separate logon.

Until logon occurs, the ObjectStore is simply an object with the name and Id of an object store. Note that you can call the following two methods on the non-validated ObjectStore object: `getId()` and `getName()`.

Getting an Object from the ObjectStore

You can create a reference to an object that resides in the ObjectStore with a call to the `getObject()` method. `ObjectStore.getObject()` returns a GettableObject object that you cast to the appropriate BaseObject type.

The following `getObject` calls use IDs to create instances of a Document object and a Folder object.

```
Document doc = (Document) os.getObject(Document.TYPE_DOCUMENT, "003676270");
Folder fld  = (Folder) os.getObject(Folder.TYPE_FOLDER,"988143961");
```

Retrieving Properties

You can retrieve property information about an object store in several ways.

The following methods return a Properties collection:

- `getProperties()`
- getProperties(propNames)

In the following code fragment, the `getProperties()` method retrieves all of the properties of an ObjectStore object. Once you have a Properties collection, you can iterate through it and gather information about the object store (not shown).

Listing 16: Retrieving ObjectStore properties

```
...
String ObjectStoreId = "pin2ms^Pinta";
ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
sess);

// Return all of the object store's properties in a collection
try
{
   props = os.getProperties();
}
catch (Exception e)
{
   return;
}
...
```

With the `getPropertiesXML(propNames)` method, you can return a string in XML that represents the requested properties for an ObjectStore object.

In addition, you can call a number of getProperty[*type*]Value methods, to return the value of a named property. (The property's data type is reflected in the *type* part of the method name; for example, getPropertyBooleanValue.) For example:

```
// Get the identifier of the object store
objStoreName= objStore.getPropertyStringValue(com.filenet.Panagon.PanagonObject.idmId);
```

Retrieving Choice Lists

You can retrieve choice list information from an ObjectStore object, either as collection of ChoiceLists objects (getChoiceLists(...)) or as a string in XML format (getChoiceListsXML(...)). Each element in a returned ChoiceLists collection is a ChoiceList collection object. A ChoiceList object represents a set of choices for a settable property. The elements in the ChoiceList collection are Choice objects, each of which represents a possible value for a property.

In the following code fragment, the `getChoiceLists(...)` method retrieves all of the ChoiceLists collection objects in ObjectStore object. Once you have a ChoiceLists collection, you can iterate through it and gather information about individual ChoiceList objects and the Choice objects that they contain (not shown).

Listing 17: Retrieving ObjectStore choice lists

```
...
String ObjectStoreId = "pin2ms^Pinta";
ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
sess);

// Return all of the object store's choice lists in a collection
Vector v = new Vector();
ChoiceLists clsCollection = os.getChoiceLists(v, 0);
...
```

Accessing Metadata

Within a given Content Services library, an ObjectStore object includes methods that return ClassDescription or PropertyDescription objects, which provide access to metadata. To retrieve a list of all ClassDescription or PropertyDescription objects contained within an object store, call one of the `getClassDescriptions()` or `getPropertyDescriptions()` methods on the ObjectStore object.

The following code fragment retrieves all of the class descriptions in an object store.

Listing 18: Retrieving all of the class descriptions in an object store

```
...
String ObjectStoreId = "pin2ms^Pinta";
ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
sess);

// Get all class descriptions
ClassDescriptions allClasses = os.getClassDescriptions();
...
```

The following code fragment retrieves the Document class descriptions in an object store:

Listing 19: Retrieving Document class descriptions in an object store

```
...
String ObjectStoreId = "pin2ms^Pinta";
ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
sess);

// Create integer array and specify Document type
int[] types =  new int[1];
types[0]=Document.TYPE_DOCUMENT;

// Get the class descriptions for Document type
ClassDescriptions docClasses = os.getClassDescriptions(types);
...
```

The following code fragment retrieves all property descriptions contained in the ObjectStore.

Listing 20: Retrieving property descriptions for the ObjectStore

```
...
String ObjectStoreId = "pin2ms^Pinta";
ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
sess);


// Create integer array and specify ObjectStore type
int[] types =  new int[1];
types[0]=ObjectStore.TYPE_OBJECT_STORE);

// Get the property descriptions for the object store
PropertyDescriptions propDescriptions = os.getPropertyDescriptions(types);
...
```

For more information about properties, refer to “Properties” on page 52. For information about class descriptions, refer to “Class Descriptions” on page 64 .

Determining ObjectStore Support

To determine an object store's support for a particular capability, call the `supports()` method. This method returns true if the object store supports the capability and false if it doesn't. For example, the following call will return true if the object store supports document annotations. To specify the object store capability, use the SUPPORTS_XXX constants defined in the com.filenet.wcm.api.ObjectStore interface and in the com.filenet.Panagon.ObjectStore interface.

```
os.supports(os.SUPPORTS_ANNOTATIONS);
```

ObjectStore Code Samples

This section provides runnable code samples that demonstrate ways of working with the ObjectStore object. For these samples to work in your development environment, you must modify hard-coded references to servers, object stores, documents, and other resources stored on a Content Services server. You must also create a Session object to log into an object store.

The following code samples are included:

- Listing 21 prints object store properties.
- Listing 22 prints specified object store properties in XML format.
- Listing 23 prints features supported in a specified object store.
- Listing 24 prints all Document property descriptions in the object store.
- Listing 25 retrieves choice lists information

For samples that include the `ObjectStore.getObject()` method, see “ClassDescription Code Samples” on page 66, “Containment Code Samples” on page 75, and “Indexing a Document” on page 88.

For a sample that uses the `ObjectStore.createObject()` method, see "createDocument code sample" on page 94.

Listing 21: objectStoreProperties code sample

```
// Prints properties of specified object store
public void objectStoreProperties()
{
  String ObjectStoreId = "pin2ms^Pinta";
  ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
    sess);

  com.filenet.wcm.api.Properties props = null;

  try
  {
    props = os.getProperties();
  }
  catch (Exception e)
  {
    return;
  }

  for (int ip=0; ip < props.size(); ++ip)
    {
      Property prop = (Property)props.get(ip);
      System.out.println("Property Name:" + prop.getName() + "\tValue:" +
prop.toString());
    }
}
```

Listing 22: objectStorePropertiesXML code sample

```
// Prints specified object store properties in XML format
public void objectStorePropertiesXML()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
     sess);

   //Specify object store properties to retrieve
   String [] descNames = new String[10];
   descNames[0] = com.filenet.Panagon.ObjectStore.idmName;
   descNames[1] = com.filenet.Panagon.ObjectStore.idmAddedByUser;
   descNames[2] = com.filenet.Panagon.ObjectStore.idmComment;
   descNames[3] = com.filenet.Panagon.ObjectStore.idmDateAdded;
   descNames[4] = com.filenet.Panagon.ObjectStore.idmAccessLevel;
   descNames[5] = com.filenet.Panagon.ObjectStore.idmLibAccessDomain;
   descNames[6] = com.filenet.Panagon.ObjectStore.idmLibAddItem;
   descNames[7] = com.filenet.Panagon.ObjectStore.idmLibAdminGroup;
   descNames[8] = com.filenet.Panagon.ObjectStore.idmLibAdminUser;
   descNames[9] = com.filenet.Panagon.ObjectStore.idmLibBackupDevice;

   String osProps = os.getPropertiesXML(descNames);
   System.out.println("ObjectStore getPropertiesXML:\n" + osProps);
}
```

Listing 23: objectStoreSupport code sample

```
// Prints support features of specified object store; true if supported, false if not.
public void objectStoreSupport()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
   sess);

  System.out.println("Support doc deletions= "+ os.supports(os.SUPPORTS_DOC_DELETE)+
    "\nSupport Access Control Lists changes= "+
      os.supports(os.SUPPORTS_ACL_MODIFICATIONS)+
    "\nSupport document annotations= "+ os.supports(os.SUPPORTS_ANNOTATIONS)+
    "\nSupport custom objects= "+ os.supports(os.SUPPORTS_CUSTOM_OBJECT)+
    "\nSupport content searching= "+ os.supports(os.SUPPORTS_CONTENT_SEARCH)+
    "\nSupport document versioning= "+ os.supports(os.SUPPORTS_DOC_VERSIONS)+
    "\nSupport compound documents= "+ os.supports(os.SUPPORTS_COMPOUND_DOCUMENTS)+
    "\nSupport publishing= "+ os.supports(os.SUPPORTS_PUBLISHING));
}
```

Listing 24: objectStoreGetPropertyDescriptionsByObjectType code sample

```
// Prints all Document property descriptions in the object store
public void objectStoreGetPropertyDescriptionsByObjectType()
{
    String ObjectStoreId = "pin2ms^Pinta";
    ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
      sess);

    int[] types = new int[1];
    types[0] = Document.TYPE_DOCUMENT;
    PropertyDescriptions propDescriptions = os.getPropertyDescriptions(types);
    com.filenet.wcm.api.Properties props = null;

    Iterator iterator = propDescriptions.iterator();
    try
    {
       while(iterator.hasNext())
       {
           System.out.println("\r\n*** Property Description ***");
           BaseObject bo = (BaseObject) iterator.next();
           ReadableMetadataObject pd = (ReadableMetadataObject) bo;
           try
           {
              props = pd.getProperties();
           }
           catch (Exception e){System.out.println(e);};

           for (int ip=0; ip < props.size(); ++ip)
           {
              Property prop = (Property)props.get(ip);
              System.out.println("Property Name:" + prop.getName() + "\tValue:" +
                prop.toString());
           }
       }
     }
        catch (Exception e){};
}
```

Listing 25: ObjectStoreGetChoiceLists code sample

```
// Gets choice lists information using different ObjectStore methods.
// The getChoiceLists(...) method returns a collection of ChoiceLists
// and prints the name of each ChoiceList object. The getChoiceListsXML(...)
// method returns ChoiceLists in XML format and writes the string to a file.
public void ObjectStoreGetChoiceLists()
{
     String ObjectStoreId = "pin2ms^Pinta";
    ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
                      sess);

     System.out.println("***** Begin Test: getChoiceLists *****");

     Vector v = new Vector();
     // Get a collection of ChoiceLists collection objects.
     // Note that the 2nd parameter is ignored by the CS Java Connector
     ChoiceLists clsCollection = os.getChoiceLists(v, 0);

     // Iterate through the ChoiceLists collection and print the name
     // of each ChoiceList object.
     Iterator iterator = clsCollection.iterator();
     while(iterator.hasNext())
     {
        System.out.println(((com.filenet.Panagon.ChoiceList)
                            iterator.next()).getName());
     }
     System.out.println("***** End Test: getChoiceLists *****");

     System.out.println("***** Begin Test: getChoiceListsXML *****");
     // Get XML string of all ChoiceLists defined in the object store.
     // Note that the 2nd parameter is ignored by the CS Java Connector
     String xml = os.getChoiceListsXML(v, 0);

     String fileName = "ObjectStore.getChoiceListsXML().xml";
     // Send to file.
     try
     {
        java.io.OutputStream fos = new java.io.FileOutputStream(fileName);
        java.io.Writer w = new java.io.BufferedWriter(new java.io.OutputStreamWriter
                                                      (fos, "Unicode"));
        w.write(xml);
        w.close();
      }
      catch(Exception e)
      {
            System.out.println(e.toString());
      }
     System.out.println("ObjectStore.getChoiceListsXML() sent to file:" + fileName);
     System.out.println("***** End Test: getChoiceListsXML *****");
}
```

Properties

All persisted Content Services objects contain properties, for example, `IdmId`, `IdmName`, and `IdmAccessLevel`. Most persisted Content Services objects contain state properties as well, which usually return boolean values. The values of state properties are derived by how other properties are set, and by the access level that a user has for a particular object. For example, `ObjectStore.STATE_CAN_ADD_DOCUMENTS` returns True if the user has sufficient access to add documents to the library. When instantiated, an object's property and state values are stored in the object's fields. You'll find a description of fields for the Panagon interfaces in the *CS Java Connector v3.0 Javadoc*.

A Panagon property has a specified data type. The property's values can be either a singleton of the specified data type (a single-valued property) or can contain multiple instances of the specified data type (a multi-valued property). If the value is an object, that object can either be a scalar object, or a collection of values held in a Values collection. A Values collection is a collection of Value objects for a multi-valued property. Each Value object in the collection is of the same type (you cannot have a multi-valued property consisting of heterogeneous data types).

Properties are cached locally in Java to enhance API performance. Although the cache operates transparently, you should familiarize yourself with caching details to handle property data in an efficient manner. For example, in environments where properties are continually updated on Content Services, you'll want to refresh the Java cache to ensure that you're working with the latest data. For more information, see "Property Cache" on page 53.

Property values are accessed through the overloaded `getProperties` methods defined in the ReadableMetaData interface. Property values can also be accessed through the interface's getProperty*XXX*Value methods, where *XXX* specifies the data type (String, Int, Double, etc.) Note that state property values can only be accessed using the getProperty*XXX*Value methods. For more information on accessing property values, see "Retrieving Property Values for an Object" on page 53.

You can set properties as well, using `setProperties(...)` and other methods. See "Setting Properties" on page 56.

Programmatically, a property can be represented by a PropertyDescription object, allowing you to obtain metadata for each property of a class or for an object type like a document or folder. The metadata defines the characteristics of a property, such as its default value and data type. When you instantiate a PropertyDescription object, the metadata is stored in the object's fields, for example:

- The identifier of the property (IdmName)
- The data type of the property (IdmTypeID)
- Whether the property is required (IdmPropRequired)
- Whether the property has a default value (STATE_PROP_HAS_DEFAULT)
- A default value for the property (IdmDefault)
- The maximum size of the property (idmSize)
- The participation of the property in query operations (STATE_SELECTABLE, STATE_PROP_SEARCHABLE)

For a description of all of the fields, see the PropertyDescription interface in the *CS Java Connector v3.0 Javadoc*. To use PropertyDescription objects to retrieve metadata, see "Retrieving Metadata with PropertyDescriptions" on page 58.

See "Properties Code Samples" on page 59 for additional examples that illustrate the use of properties in the *CS Java Connector v3.0 Javadoc*.

Property Cache

The CS Java Connector uses a caching mechanism to reduce round-trips to the Content Services server for property retrievals. The property cache used in this caching mechanism is a property bag attached to a specific Java object. For example, if you use the CS Java Connector to return separate references to the same Content Services object by different means (such as calling `ObjectStore.getObject` and `Folder.getContainees`), the two Java objects have separate property caches.

For the typical caching scenario (where properties are requested from Content Services), the property cache has two main characteristics of interest: complete/incomplete and stale/not stale.

- The cache is complete if it knows about every possible property for the object. The cache is incomplete when you first retrieve an object from Content Services (either directly with the ObjectFactory or as the result of an object returned by a method of some other object). The object will have a minimal set of generally-useful properties brought with it and placed into the cache. The actual set of initial properties varies by object type.
- The cache is stale if one or more properties in it is stale. A property is stale if its value doesn't match the same property value for the object in the Content Services object store (library). In general, the CS Java Connector cannot tell if the cache is stale simply because a different application could have changed the property value since the CS Java Connector last retrieved it.

A call to getProperties( ) with no parameters returns all the properties for a ReadableMetadataObject object. When the cache is incomplete , the call will result in a round-trip to Content Services that retrieves all possible properties for the object. For a Document object, the properties retrieved are those specified in the Document's class description. Those properties are then placed into the cache and the cache is complete.

A call to getProperties( ) is handled locally if the cache is complete; that is, the call does not result in a round-trip to Content Services because the CS Java Connector knows through internal tracking that it already has all the properties for the object.

A call to getProperties(*propNames*) requests a list of specific properties. If the cache is complete and a requested property is not in the cache, the CS Java Connector throws an immediate exception. If the cache is incomplete, then the specified properties may or may not be in the cache. For properties that are not already cached, the CS Java Connector requests the data from Content Services and stores it in the cache.

Note that if your application is operating in an environment where other applications are modifying properties on the Content Services, your application should call one of the forms of the `refresh` method, to either replace all or selected property data in the cache. This method is defined in the ReadableMetadataObject interface.

Retrieving Property Values for an Object

To return the values of object properties, you can call the overloaded `getProperties` methods defined in the ReadableMetaDataObject interface. In Listing 26, the form of the `getProperties` method that is used retrieves all of the properties for the specified Folder object. In Listing 27 , the form of the method that is used takes a String array that identifies specific Folder properties to return. In both examples, the code iterates through the retrieved properties collection, printing out the name and value of each property.

Listing 26: Retrieving all values with getProperties( )

```
...
Folder fld = (Folder) os.getObject(BaseObject.TYPE_FOLDER, "988143960" );
try
{
   com.filenet.wcm.api.Properties fldProps = fld.getProperties();
   Iterator pi = fldProps.iterator();
   while (pi.hasNext())
   {
     com.filenet.wcm.api.Property fldProp = (com.filenet.wcm.api.Property) pi.next();
      String strPropVal = "null";
      Object objPropVal = fldProp.getValue();
      if (objPropVal != null)
          strPropVal = objPropVal.toString();
         System.out.println("Property: \""  + fldProp.getName() + "\" has value: \"" +
            strPropVal + "\"");
   }
...
```

Listing 27: Retrieving selected values with getProperties(propNames)

```
...
Folder fld = (Folder) os.getObject(BaseObject.TYPE_FOLDER, "988143960");
String [] propNames = new String[7];
propNames[0] = com.filenet.Panagon.Folder.idmName;
propNames[1] = com.filenet.Panagon.Folder.idmDateAdded;
propNames[2] = com.filenet.Panagon.Folder.idmComment;
propNames[3] = com.filenet.Panagon.Folder.idmAccessLevel;
propNames[4] = com.filenet.Panagon.Folder.idmAddedByUser;

try
{
   com.filenet.wcm.api.Properties fldProps = fld.getProperties(propNames);
   Iterator pi = fldProps.iterator();
   while (pi.hasNext())
   {
     com.filenet.wcm.api.Property fldProp = (com.filenet.wcm.api.Property) pi.next();
      String strPropVal = "null";
      Object objPropVal = fldProp.getValue();
      if (objPropVal != null)
          strPropVal = objPropVal.toString();
         System.out.println("Property: \""  + fldProp.getName() + "\" has value: \"" +
            strPropVal + "\"");
   }
...
```

Alternatively, you can call the getProperty[type]Value convenience methods defined in the ReadableMetadataObject interface. These methods cast or convert a value to a specific data type or, in the case of `getPropertyValuesValue(...)`, to a Values collection. In Listing 28, the `getPropertyStringValue()` method is called on the Folder object, with each call specifying a different Folder property. (These properties are defined in the com.filenet.Panagon.Folder interface.)

Listing 28: Retrieving values with getPropertyStringValue(namedProperty)

```
...
Folder fld = (Folder) os.getObject(BaseObject.TYPE_FOLDER, "988143960" );
String userFolder=null;
String createFolder=null;
String deleteFolder =null;
String moveFolder =null;


// Assume the folderObj is an already-instantiated Folder object
userFolder= folderObj.getPropertyStringValue(
   com.filenet.Panagon.Folder.idmAddedByUser);
createFolder= folderObj.getPropertyStringValue(
   com.filenet.Panagon.Folder.STATE_CAN_CREATE_SUB_FOLDER);
deleteFolder = folderObj.getPropertyStringValue(
   com.filenet.Panagon.Folder.STATE_CAN_DELETE);
folderObj.getPropertyStringValue(com.filenet.Panagon.Folder.STATE_CAN_MOVE);
...
```

The following example shows how to retrieve a multi-valued, String-valued property. Note that properties whose values are Content Services collection objects are returned as Values collections in the CS Java API.

Listing 29: Retrieving values with getPropertyValuesValue(namedProperty)

```
Document aDoc = (Document)objectStore.getObject(BaseObject.TYPE_DOCUMENT,
                                                "/Folder/SomeDoc");
  Values vmProps = aDoc.getPropertyValuesValue("SomeMVProp");
  Iterator it = vmProps.iterator();
  while (it.hasNext())
  {
      Value v = (Value)it.next();
      String s = v.getStringValue();
      System.out.println(s);
  }
```

Note: In general, it is better performance-wise to ask for any properties your application requires in one call to `getProperties` instead of retrieving each property as you need it. Doing so places the requested properties into the properties cache so later accesses (for example, via `getPropertyStringValue(...)`) will retrieve from the cache. If you request a property that is not in the cache, the CS Java API makes a roundtrip to the Content Services server and also clears the cache to avoid an inconsistent state between properties already in the cache and properties newly-retrieved. Therefore, if you were to initially request each property individually, the Content Java API might make a roundtrip to the server for each property requested.

Setting Properties

You can set properties on objects that implement the WriteableMetadataObject or CreatableObject interface. Methods that set properties include:

- The WriteableMetadataObject interface's `setProperties(...)` method. See "Setting Properties via setProperties" on page 57.
- The CreatableObject object's `createObject(...)` method, and the Folder object's `addSubFolder(...)`. See "Setting Properties for a New Object" on page 57.
- A Document object's `changeClass(...)` method. For more information on using this method, see "Changing a Document's Class" on page 88.

All of these methods take a Properties collection as a parameter and attempt to persist the given values of those properties to the object store:

- If a property is read-only and you explicitly attempt to set its value, the method throws an exception. (Note that the exception is thrown during the call that sets the properties, not during the Property.setValue call.) To avoid this, either filter the properties based on the STATE_PROP_CAN_WRITE property's setting or know if your application can set the properties.
- If an update to any specific property value fails, the entire set of updates is discarded. In other words, if you get an exception from the calling method, no property value changes will have occurred.
- If the Properties collection contains more than one entry for a given Property object, the order in which the updates are applied is undefined.

The Content Java API tracks which properties are explicitly set and attempts to persist only those properties:

- Copying properties - If your application is copying properties from one object to another, you are explicitly attempting to set each property's value.
- Modifying properties in place - If your application is retrieving a collection of properties, modifying some properties, and then resubmitting the original collection, the method that is attempting to set properties will set only the modified properties; any Property objects that did not change will be silently dropped before sending the collection to the Content Services server.
- Creating properties via ObjectFactory - Any Property objects that your application creates via ObjectFactory are sent to the Content Services server. For example, your application could retrieve a collection of properties, display them in a Graphical User Interface (GUI), then copy all the properties back (changed or not). If the application creates new Property objects via ObjectFactory for sending the values back, then all properties are considered explicitly set (regardless of whether the property's value changed or not).

Note that one of the ways the CS Java Connector tracks which properties are explicitly set is by calls made to `setValue(...)`. Therefore, if your application has a reference to an object contained in a Value object and then modifies that reference directly, the CS Java Connector would not detect the value as having been changed and will drop it. For example:

```
byte[] b = someProperty.getBinaryValue();
b[1] = 42; // we don't know you set the value
someProperty.setValue(b); // you must call setValue
```

Note that when calling `setValue(...)` to set floating-point values, the value you specify is limited to the range imposed by the database you are using. An attempt to persist a floating-point value to the object store will fail with a RemoteServerException if the number is outside the database's range for the data

type. Refer also to the Property interface's setValue(float64Value) methods in the *CS Java Connector v3.0 Javadoc*.

Setting Properties for a New Object

In this release of the CS Java Connector, you can create a new Document object and a new Folder object with the `createObject(...)` method and the `addSubFolder(...)` method, respectively. Both methods take a Properties collection as a parameter. If the object has properties that are required but have no default values, you must supply values for them. If you pass in an empty Properties collection or if some properties are not specified, properties that have default values in the object store take the defaults.

The following example sets a property for a new document by passing a new Properties collection to the createObject method.

```
  ...
// Create an empty Properties collection
  Properties docProps = ObjectFactory.getProperties();
  // Create a new Property object
  Property docName = ObjectFactory.getProperty(com.filenet.Panagon.Document.idmComment);
  // Set a value for the property
  docName.setValue("Source Document");
  // Add the Property to the Properties collection
  docProps.add(docName);
  // Set the properties for the document
  newDoc = (Document)objStore.createObject(ClassDescription.DOCUMENT, docProps, null);
  ...
```

Setting Properties via setProperties

You can set properties on objects that implement the WriteableMetadataObject interface using the `setProperties(...)` method. This method takes a Properties collection and attempts to persist the given values of those properties to the object store. If you pass in null or an empty Properties collection, the `setProperties(...)` method throws an IllegalArgumentException.

For example, the following code fragment retrieves the DocumentTitle property from a document, changes the value of the property, and persists the change to the Content Engine server.

```
...
Document aDoc = (Document)objectStore.getObject(BaseObject.TYPE_DOCUMENT, "/Folder/
                                                SomeDoc");
String oldName = aDoc.getPropertyStringValue(Property.DOCUMENT_TITLE);
String newName = "New document title";
Properties props = ObjectFactory.getProperties();
Property nameProp = ObjectFactory.getProperty(Property.DOCUMENT_TITLE);
nameProp.setValue(newName);
props.add(nameProp);
aDoc.setProperties(props);
System.out.println("Renamed from '" + oldName + "' to '" + newName    + "'");
...
```

The following code fragment shows how to set a multi-valued, String-valued property.

```
...
Document aDoc = (Document)objectStore.getObject(BaseObject.TYPE_DOCUMENT, "/Folder/
                                                SomeDoc");
Properties props = ObjectFactory.getProperties();
Property prop = ObjectFactory.getProperty("SomeMVProp");
Values vs = ObjectFactory.getValues();
Value v1, v2;
v1 = ObjectFactory.getValue();
```

```
v1.setValue("application/excel");
vs.add(v1);
v2 = ObjectFactory.getValue();
v2.setValue("application/powerpoint");
vs.add(v2);
prop.setValue(vs);
props.add(prop);
aDoc.setProperties(props);
...
```

Retrieving Metadata with PropertyDescriptions

A PropertyDescription object allows you to obtain descriptive information (metadata) about the properties of a class or of an object type. For example, `idmVerCsiStatus` is a Document property, which can be represented as a PropertyDescription object. Individual properties of this PropertyDescription object are individual items of metadata, such as `idmPropertyType`, `idmLabel`, `idmDefault`, and `idmTypeId` -- all of which are defined in the com.filenet.Panagon.PropertyDescription interface. Calling the `getProperties()` method on a PropertyDescription object returns the individual items of metadata.

To retrieve metadata, you return a collection of PropertyDescription objects. You can do this in the following ways:

- By calling the `getPropertyDescriptions`( ) method on a ClassDescription object, which returns all property descriptions for the ClassDescription object.
- By calling an overloaded `getPropertyDescriptions` method on an ObjectStore object, which returns a collection of PropertyDescription objects for a ClassDescription object or for an object type.

Note: A particular PropertyDescription object may exhibit property value differences depending on the ClassDescription or ObjectStore object from which the Property Description was obtained. For details, see "PropertyDescription Discrepancies" on page 58.

With a PropertyDescriptions collection, you can do the following:

- Call `filterByProperty` on the collection to retrieve a new PropertyDescriptions collection, filtered by a specific property.
- Call `findByProperty` on the collection to retrieve a single PropertyDescription, filtered by a specific property.
- Use java.util.Collection and java.util.List methods to retrieve a PropertyDescription from the collection.

PropertyDescription Discrepancies

Because of how property descriptions are created and used on a Content Services server, programmatic discrepancies can occur between multiple PropertyDescription objects that represent the same property description on the server. Property descriptions are initially created on an object store with a default set of property values. These same Property descriptions can later be placed into classes, and their default property values can be overridden as members of a class. In fact, a property description assigned to two or more classes can have different values in each class of which it is a member.

Consider the following scenario where the CS administrator creates a custom property in Content Services, and a CS Java Connector program accesses it as a PropertyDescription object .

Using Admin Tools in Content Services, the administrator:

1. Creates a custom document property on an object store called "Interest Rate". At the object store level this custom property has no default value defined. (Programmatically, a custom property is represented as a PropertyDescription object, with its own set of properties that define the object.)
2. Creates two document classes that both contain "Interest Rate". One is called "Loan Class Bad Credit" and the other is "Loan Class Great Credit". (Programmatically, a class is represented as a ClassDescription object.)

   For "Loan Class Bad Credit", the Administrator configures "Interest Rate" with a Default Value of "30%" and makes "Interest Rate" a Required property for the class.

   For "Loan Class Great Credit", the Administrator configures "Interest Rate" with a Default Value of "5%" but "Interest Rate" is not Required.

A CS Java Connector program:

1. Obtains the "Interest Rate" PropertyDescription object directly from an ObjectStore object. There is no default value attached to the object.
2. Obtains the "Interest Rate" PropertyDescription object from a ClassDescription object representing the "Loan Class Bad Credit" class. The Property Description object has a Default Value property of 30% and a Required property of true, meaning that a user must provide an interest rate value when creating a document with this class.
3. Obtains the "Interest Rate" PropertyDescription object from a ClassDescription object representing the "Loan Class Great Credit" class. The Property Description object has a Default Value property of 5% and a Required property of false.

Properties Code Samples

This section provides runnable code samples that demonstrate ways of working with property-related objects. For these samples to work in your development environment, you must modify hard-coded references to servers, object stores, documents, and other resources stored on a Content Services server. You must also create a Session object to log into an object store.

The following code samples are included in this section:

- Listing 30 retrieves the property descriptions for Folder type, and prints the metadata for all Folder properties.
- Listing 31 illustrates the use of the `filterByProperty` method. Two PropertyDescriptions collections are created, one to contain all of the properties and metadata for the Document type in the object store, the other to hold a filtered set of properties and metadata. When the `filterByProperty` method is called, the method filters out all Document properties with a data type of Date, such as `idmDateAccessed`, `idmDateAdded`, etc, and places the non-Date property types into the filtered collection. The code then prints the metadata from the filtered collection.
- Listing 32 modifies document properties.

You'll find property-related code samples in other sections as well, including "ObjectStore Code Samples" on page 47, "Containment Code Samples" on page 75, "Document Code Samples" on page 89, and "Versioning Code Samples" on page 104.

Listing 30: propertyDescriptionsAllForFolderType code sample

```
/* Retrieves property descriptions for Folder type, and prints the metadata for all
   Folder properties. */
public void propertyDescriptionsAllForFolderType()
{
    String ObjectStoreId = "pin2ms^Pinta";
    ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
      sess);

   int[] types = new int[1];
   types[0] = Folder.TYPE_FOLDER;
   PropertyDescriptions propDescriptions = os.getPropertyDescriptions(types);

    Iterator iterator = propDescriptions.iterator();
    try
    {
        while(iterator.hasNext())
        {
           System.out.println("*** Property Description ***");
           BaseObject bo = (BaseObject) iterator.next();
           printProperties( (ReadableMetadataObject) bo);
        }
    }
    catch (Exception e){};
}

public void printProperties(ReadableMetadataObject ro)
{
   com.filenet.wcm.api.Properties props = null;
   try
   {
      props = ro.getProperties();
   }
   catch (Exception e)
   {
      return;
   }

   for (int ip=0; ip < props.size(); ++ip)
   {
      Property prop = (Property)props.get(ip);
      System.out.println("\tProperty Name:" + prop.getName() + "\tValue:" +
prop.toString());
   }
}
```

Listing 31: propertyDescriptionsFilteredForDocumentType code sample

```
/* Retrieves property descriptions for Document type, filters out Document
   properties with Date type, and prints metadata for the non-Date property types. */
public void propertyDescriptionsFilteredForDocumentType()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
    sess);
   int[] types = new int[1];
   types[0] = Document.DOCUMENT;

   /* Create PropertyDescriptions collection to contain all of the properties and
      metadata for the Document type in the object store. */
   PropertyDescriptions propDescriptions = os.getPropertyDescriptions(types);
   // Declare variable for PropertyDescriptions collection to contain filtered
   // properties.
   PropertyDescriptions filteredList;

   // Filter out all Document properties with a data type of Date
   try
   {
      // find all the "Date" property descriptions
      filteredList = (PropertyDescriptions)
      propDescriptions.filterByProperty(
         com.filenet.Panagon.PropertyDescription.idmTypeId,
         ReadableMetadataObjects.IS_NOT_EQUAL,
         Property.TYPE_DATE);
    }
    catch (com.filenet.wcm.api.PropertyNotFoundException e)
    {
       e.printStackTrace();
       return;
    };
```

continued on next page

Listing 31: propertyDescriptionsFilteredForDocumentType code sample *(continued)*

```
    // Print metadata of Document properties in filteredList collection.
    Iterator iterator = filteredList.iterator();
    try
    {
        while(iterator.hasNext())
        {
           System.out.println("*** Property Description ***");
           BaseObject bo = (BaseObject) iterator.next();
           printProperties( (ReadableMetadataObject) bo);
        }
    }
    catch (Exception e){};
}

public void printProperties(ReadableMetadataObject ro)
{
   com.filenet.wcm.api.Properties props = null;
   try
   {
// Print metadata of Document properties in filteredList collection.
    Iterator iterator = filteredList.iterator();
    try
    {
        while(iterator.hasNext())
        {
           System.out.println("*** Property Description ***");
           BaseObject bo = (BaseObject) iterator.next();
           printProperties( (ReadableMetadataObject) bo);
        }
    }
    catch (Exception e){};
}
```

Listing 32: documentSetProperties code sample

```
// Modifies properties on an existing document. Sets single-value Property objects
// and a multi-value Property object.
public void documentSetProperties()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
                     sess);

   Document doc = (Document) os.getObject(Document.TYPE_DOCUMENT, "003676270" );

   // Create Properties collection to hold modified properties
   com.filenet.wcm.api.Properties docProps = ObjectFactory.getProperties();

   // Create new single-value Property objects and set values.
  Property docName = ObjectFactory.getProperty(com.filenet.Panagon.Document.idmName);
   docName.setValue("New Document name");
   Property docComment =
            ObjectFactory.getProperty(com.filenet.Panagon.Document.idmComment);
   docComment.setValue("New Document Comment");
   Property docVerProp =
           ObjectFactory.getProperty(com.filenet.Panagon.Document.idmVerCustom8);
   docVerProp.setValue(1000);

   // Add the single-value Property objects to the Properties collection
   docProps.add(docName);
   docProps.add(docComment);
   docProps.add(docVerProp);

   // Create new multi-value Property and set values (keywords)
   Property docMVPropk =
            ObjectFactory.getProperty(com.filenet.Panagon.Document.idmDocKeywords);
   // Create an empty Values collection for adding values
   Values valsk = ObjectFactory.getValues();
   // Create Value objects to add to the Values collection.
   Value val1k = ObjectFactory.getValue();
   Value val2k = ObjectFactory.getValue();
   // Set Value objects and add them to the Values collection
   val1k.setValue("keyword one");
   val2k.setValue("keyword two");
   valsk.add(val1k);
   valsk.add(val2k);
   // Add the Value objects to the multi-value Property object
   docMVPropk.setValue(valsk);
   // Add the multi-value Property object to the Properties collection
   docProps.add(docMVPropk);

   // Set the properties on the Document object
   doc.setProperties(docProps);
}
```

Class Descriptions

In the CS Java Connector, classes defined on Content Services are represented as ClassDescription objects. You can retrieve all of the classes stored in an object store (library), or all the classes of a particular object type, for example, Document, Folder, etc. Once you have a ClassDescription object, you can then access either the ClassDescription's properties or retrieve the ClassDescription's set of PropertyDescriptions. The PropertyDescriptions represent custom and/or system properties that define the ClassDescription.

This section provides the following topics:

For information on changing a document's class, see page 88.

Retrieving ClassDescription Objects

You cannot create a new ClassDescription object with the CS Java Connector, but you can instantiate an existing one in the following ways:

- Call an overloaded `getClassDescriptions` method on the ObjectStore, which creates a collection of ClassDescription objects. You can then return individual ClassDescription objects from the collection.
- Call the `getObject(...)` method on an ObjectStore object to return a specific class.
- Call the `getClassDescription()` method on the Document object to return a specific class.

ObjectStore.getClassDescriptions( )

You can retrieve a ClassDescriptions collection by calling the following methods on an ObjectStore object:

- `getClassDescriptions()`, which retrieves all classes from the object store.
- `getClassDescriptions(`*`objTypes`*`)` and `getClassDescriptions(`*`classDescs`*`)`, which retrieve classes of the specified object types and specified classes, respectively.

  Note: For this release of the CS Java Connector, the `getClassDescriptions(`*`objTypes`*`)` method returns class descriptions objects for the Document, Folder, and read-only versions of StoredSearch object types.

From a ClassDescriptions collection, you can:

- Use java.util.Collection and java.util.List methods to retrieve a ClassDescription from the collection.
- Call `filterByProperty` on the collection to retrieve a new ClassDescriptions collection, filtered by a specific property.
- Call `findByProperty` on the collection to retrieve a single ClassDescription, filtered by a specified property.

The following code fragment shows how object types are specified to return a collection of ClassDescription objects from the object store. The `findByProperty` method is then used to retrieve a ClassDescription object whose ID is known (“LoanApp”).

Listing 33: Returning a collection of ClassDescription objects

```
...
// Create String array to pass into getClassDescriptions
int[] types = new int[1];
// Populate the array with the type for Document
types[0] = Document.TYPE_DOCUMENT;
// Get the ClassDescription for Document
ClassDescriptions docClasses = myObjectStore.getClassDescriptions(types);
//Find the ClassDescription named "LoanApp"
ClassDescription newDocClass = (ClassDescription)docClasses.findByProperty
                               (com.filenet.Panagon.PanagonObject.idmName,
                                ReadableMetadataObjects.IS_EQUAL,
                                "LoanApp");
...
```

ObjectStore.getObject( )

To retrieve a specific ClassDescription from an object store, call getObject on an ObjectStore object. Specify the ClassDescription object type and the specifc object's Id property value, for example:

```
ClassDescription classDesc =
(ClassDescription)o.getObject(BaseObject.TYPE_CLASSDESCRIPTION, “Loan”);
```

Document.getClassDescription( )

To programmatically retrieve the class assigned to a specific document, call the getClassDescription method on the Document object, for example:

```
com.filenet.Panagon.Document doc =
   (com.filenet.Panagon.Document)os.getObject (Document.TYPE_DOCUMENT, "003676242" );
ClassDescription cd = doc.getClassDescription();
```

Getting Property Descriptions

As described in “Retrieving Metadata with PropertyDescriptions” on page 58, you can use the PropertyDescriptions object to retrieve metadata. The ClassDescription interface includes a `getPropertyDescriptions()` method for retrieving custom and/or system properties that define a ClassDescription object. This method is a shortcut to the ObjectStore interface's `getPropertyDescriptions()` method, which takes a ClassDescriptions collection as an argument. The following code fragfragmentfragmentment illustrates this shortcut method.

Listing 34: getPropertyDescriptions method for ClassDescription object

```
...
String ObjectStoreId = "pin2ms^Pinta";
ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
sess);
// Retrieve a Document class called "Loans"
ClassDescription classDesc = (ClassDescription) os.getObject
   BaseObject.TYPE_CLASSDESCRIPTION,"Loans");

// Put the property descriptions for the class into a collection
PropertyDescriptions pds = classDesc.getPropertyDescriptions();

// Create an Iterator object to loop through collection and print property
// descriptions
...
```

Getting Default Permissions on a Class

Using the `getDefaultPermissions()` method on a retrieved ClassDescription object, you can retrieve the default permissions for the class. Use this method (or the XML version of the method) when creating an object to determine if the default permissions are appropriate for your application.

In the code fragment, a ClassDescription object is retrieved and the default permissions returned.

```
...
ClassDescription cd = (ClassDescription)os.getObject(BaseObject.TYPE_CLASSDESCRIPTION,
                                                     "QA-Basic");
Permissions perms = cd.getDefaultPermissions();
...
```

For information on Permission objects, see "Access Rights to Documents and Folders" on page 33.

ClassDescription Code Samples

This section provides runnable code samples that demonstrate ways of working with ClassDescription objects. For these samples to work in your development environment, you must modify hard-coded references to servers, object stores, documents, and other resources stored on a Content Services server. You must also create a Session object to log into an object store.

The following code samples are included in this section:

- Listing 35 retrieves all class descriptions from an object store and prints the properties for each class description.
- Listing 36 retrieves all Document type class descriptions from an object store and prints the properties for each class description.
- Listing 37 code retrieves a specific class description from an object store, then retrieves and prints the property descriptions of the class.
- Listing 38 retrieves the default permissions for a ClassDescription object.

Listing 35: objectStoreGetClassDescriptions code sample

```
/* Retrieves all class descriptions from an object store and prints the class
   description properties */
public void objectStoreGetClassDescriptions()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
     sess);

   // Get all class descriptions
   ClassDescriptions allClasses = os.getClassDescriptions();

   try
   {
       Iterator iterator = allClasses.iterator();
       while(iterator.hasNext())
       {
          BaseObject bo = (BaseObject) iterator.next();
          System.out.println("\n*** Class Description ***");

          com.filenet.wcm.api.Properties props = null;

          ReadableMetadataObject cd = (ReadableMetadataObject) bo;
           try
           {
              props = cd.getProperties();
           }
           catch (Exception e){System.out.println(e);};

           for (int ip=0; ip < props.size(); ++ip)
           {
              Property prop = (Property)props.get(ip);
              System.out.println("Property Name:" + prop.getName() + "\tValue:" +
                prop.toString());
           }
       }
   }
         catch (Exception e){System.out.println(e);};
}
```

Listing 36: objectStoreGetClassDescriptionsByObjectType code sample

```
/* Retrieves all Document type class descriptions from an object store and prints
   the class description properties */
public void objectStoreGetClassDescriptionsByObjectType()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
     sess);

    // Get all class descriptions from object store
   int[] types =  new int[1];

   // Specify document type class descriptions to be returned
   types[0]=Document.TYPE_DOCUMENT;

   // Get all class descriptions for Document type
   ClassDescriptions docClasses = os.getClassDescriptions(types);

   try
   {
       Iterator iterator = docClasses.iterator();
       while(iterator.hasNext())
       {
            BaseObject bo = (BaseObject) iterator.next();
            System.out.println("\r\n*** Class Description ***");
            com.filenet.wcm.api.Properties props = null;

            ReadableMetadataObject cd = (ReadableMetadataObject) bo;
            try
            {
              props = cd.getProperties();
            }
            catch (Exception e){System.out.println(e);};

            for (int ip=0; ip < props.size(); ++ip)
            {
              Property prop = (Property)props.get(ip);
              System.out.println("Property Name:" + prop.getName() + "\tValue:" +
                prop.toString());
            }

       }
   }
        catch (Exception e){System.out.println(e);};
}
```

Listing 37: classDescriptionGetPropertyDescriptions code sample

```
/* Retrieves a specific class description from an object store,
   and retrieves and prints the property descriptions of the class. */
public void classDescriptionGetPropertyDescriptions()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
     sess);

   // Retrieve specific class
   ClassDescription classDesc = (ClassDescription)
     os.getObject(BaseObject.TYPE_CLASSDESCRIPTION,"QA-basic");

   // get the class' property descriptions
   PropertyDescriptions pds = classDesc.getPropertyDescriptions();
   Iterator iterator = pds.iterator();
   try
   {
      while(iterator.hasNext())
      {
         System.out.println("*** Property Description ***");
         BaseObject bo = (BaseObject) iterator.next();
         printProperties( (ReadableMetadataObject) bo);
      }
   }
   catch (Exception e){};
}

public void printProperties(ReadableMetadataObject ro)
{
   com.filenet.wcm.api.Properties props = null;
   try
   {
      props = ro.getProperties();
   }
   catch (Exception e)
   {
      return;
   }

   for (int ip=0; ip < props.size(); ++ip)
   {
      Property prop = (Property)props.get(ip);
      System.out.println("\tProperty Name:" + prop.getName() + "\tValue:" +
prop.toString());
   }
}
```

Listing 38: classDescriptionGetDefaultPermissions

```
// Get the default permissions on a ClassDescription object
// retrieved from a Document object.
public void classDescriptionGetDefaultPermissions()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
                     sess);

   // Get Document object.
   com.filenet.Panagon.Document doc = (com.filenet.Panagon.Document)
                                  os.getObject(Document.TYPE_DOCUMENT, "003676270" );

   // Get the document's ClassDescription object.
   ClassDescription cd = doc.getClassDescription();

   // Return a Permissions collection with the default permissions
   Permissions perms = cd.getDefaultPermissions();

   // Print the permissions
   printPermissions(perms);
}

public void printPermissions(Permissions perms)
{
   System.out.println("*** Permissions ***");
   Iterator iterator = perms.iterator();
    while(iterator.hasNext())
    {
       Permission perm = (Permission) iterator.next();
       System.out.println("Grantee Name:" + perm.getGranteeName());
       System.out.println("\tAccess:" + perm.getAccess());
       System.out.println("\tAccess type:" + perm.getAccessType());
       if (perm.getGranteeType() == BaseObject.TYPE_USER)
       {
          System.out.println("\tGrantee type:User");
       }
       else if (perm.getGranteeType() == BaseObject.TYPE_GROUP)
       {
          System.out.println("\tGrantee type:Group");
       }
       else
       {
          System.out.println("\tGrantee type:UNKNOWN!!!!!");
       }
    }
}
```

Containment

A folder represents a container that can hold other objects. Each Content Services object store (library) has an automatically-created root folder that represents the default root associated with the object store. Contained objects are child folders and documents.

Child folders are directly contained; that is, their containment models a 1:N, or one-to-many, relationship. A containing (parent) folder can contain multiple child folders, but each child folder is directly contained within at most one parent folder.

Document objects implement the ContainableObject interface, making them referentially-contained. Their containment models an N:M, or many-to-many, relationship. A referentially-contained object can be contained within multiple folders.

This section shows you how to work with folders and with documents as contained objects. It covers the following topics:

For code samples, see "Containment Code Samples" on page 75.

Folder Permissions

A folder has the following security characteristics:

- It is secured by its own ACL.
- It can serve as security parent to the ContainableObject objects it contains.

For more information, see "Retrieving and Setting Permissions" on page 35.

Creating a Folder Object

To create a Folder object, call the `addSubFolder(...)` method on the parent Folder object. Note that in Content Services, there is only one folder class — _BASE_FOLDER — for all folders. There is no inheritance of folder classes. There is, however, optional subfolder inheritance of permissions from the parent folder.

You are not required to set system properties when you create a folder, although the folder's ClassDescription may include custom properties that you are required to set.

In the example below, the newly-created subfolder will inherit permissions from its parent folder, although you can optionally set permissions for the new object when you create it. Default values are used for the folder properties.

```
...
// Retrieve the root folder, then add the subfolder, My Folder, to it.
// Properties that have default values in the Content Services object store take the
// defaults. Permissions are inherited from the parent folder, indicated by the null.
Folder rootFolder = objectStore.getRootFolder();
Folder newFolder = rootFolder.addSubFolder("My Folder",
                   com.filenet.wcm.api.Properties.ObjectFactory.getProperties(), null);
// Then add a subfolder to the folder you just created
Folder anotherFolder = newFolder.addSubFolder("AnotherFolder",
                   com.filenet.wcm.api.Properties.ObjectFactory.getProperties(), null);
...
```

Retrieving Folder Objects

You can retrieve a persisted Folder object in the following ways:

- By calling `getRootFolder()` on an ObjectStore object to retrieve the root folder for the object store. Note that you cannot create or delete a root folder.
- By calling `getObject()` on an ObjectStore object to retrieve a specific folder from the object store.
- By calling `getParentFolder()` on a Folder object to retrieve the folder's parent folder.

There are a number of methods that you can call on a Folder object, including moving it from one parent folder to another.

You can retrieve a Folders collection in the following ways:

- By calling one of the forms of the `getContainees(...)` method on a Folder object (see “Retrieving Objects from a Folder” on page 73).
- By calling the `getContainers()` method on an object implementing the ContainableObject interface, which returns the list of folders that contain the object (see “Working with Contained Objects” on page 74).

From a Folders collection, you can:

- Use java.util.Collection and java.util.List methods to retrieve a Folder from the collection.
- Call `filterByProperty()` on the collection to retrieve a new Folders collection, filtered by a specific property.
- Call `findByProperty()` on the collection to retrieve a single Folder, filtered by a specify property.
- Call the `getTopFoldersXML()` method on an ObjectStore object.

Deleting a Folder Object

To delete a Folder object, call the `delete()` method on it. For example:

```
Folder fld = (Folder) os.getObject(Folder.TYPE_FOLDER, "1027700919" );
fld.delete();
```

If you delete a folder that contains subfolders, all subfolders are deleted as well, assuming security is sufficient. Other objects, such as documents, are automatically un-filed from the folder when you delete it.

You cannot delete the root folder. Also, you must have the appropriate permissions to delete a folder. See “Folder Access Rights” on page 35.

Retrieving Folder Properties

The Folder interface extends ReadableMetadataObject; therefore, a Folder implementation supports the property retrieval and caching methods of ReadableMetadataObject. For general information on properties, see page 52.

In the following code fragment, all of the properties of a Folder object are returned into a Properties collection. Iterating through the collection, the code prints out the name and value of each Folder property.

Listing 39: Retrieving folder properties

```
...
Folder fld = (Folder) objectStore.getObject
   (BaseObject.TYPE_FOLDER, "988143960" );
try
{
   com.filenet.wcm.api.Properties fldProps = fld.getProperties();
   Iterator pi = fldProps.iterator();
   while (pi.hasNext())
   {
     com.filenet.wcm.api.Property fldProp = (com.filenet.wcm.api.Property) pi.next();
     String strPropVal = "null";
     Object objPropVal = fldProp.getValue();
     if (objPropVal != null)
        strPropVal = objPropVal.toString();
     System.out.println("Property: \"" + fldProp.getName() +
        "\" has value: \"" + strPropVal + "\"");
   }
...
```

Retrieving Objects from a Folder

To retrieve objects filed in a folder, call the overloaded `getContainees` method on the Folder object. The `getContainees()` form of the method returns all objects in the folder, along with a default set of properties.

In the other forms of the method, you specify the object types to return and, optionally, pass in a String array with the symbolic names of the properties to retrieve. For example, the following code fragment retrieves all documents in the root folder along with the `idmName` and `idmId` properties.

Listing 40: Retrieving documents in the root folder

```
...
// Create integer array and specify Document type
int[] objTypes = new int[1];
objTypes[0] = Document.TYPE_DOCUMENT;

// Create string array and populate it with the Document properties to retrieve.
String[] propNames = new String[2];
propNames[0] = com.filenet.Panagon.PanagonObject.idmName;
propNames[1] = com.filenet.Panagon.PanagonObject.idmId;

// Get the root folder
Folder rootfolder = objStore.getRootFolder();

// Get document in root folder
BaseObjects rootDocuments = rootfolder.getContainees(objTypes, propNames);
```

Working with Contained Objects

A referentially-contained object implements the ContainableObject interface. In this release of the CS Java Connector, only Document objects implement methods defined in ContainableObject. These methods allow you to:

- File a document into a folder and unfile it from a folder.

  The following statement files the document, newDoc, into the folder, someFolder. Note that the document must be checked in to file it.

  ```
  newDoc.file(someFolder, true);
  ```

  The following statement unfiles the document.

  ```
  newDoc.unfile(someFolder);
  ```

- Return the folders into which a document is filed. The following statement returns a collection of folders into which the document is contained. As a referentially-contained object, the document may be contained in one or multiple folders. You can iterate through the Folders collection and retrieve properties for each Folder object.

  ```
  Folders objs = doc.getContainers();
  ```

- Retrieve the containment full path(s) where a document is filed by calling `getContainmentPaths` on the ContainableObject object. The root folder's path is "/". For example, "`/myTopFolder`" is the containment path for a document filed in the myTopFolder folder. (Note that a containment path uses forward slashes.)

  In the code fragment below, the containment paths for a specified document are retrieved and printed.

Listing 41: Retrieving containment paths for a document

```
...
ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
sess);
// Get document
Document doc     = (Document) os.getObject(Document.TYPE_DOCUMENT, "003682302" );
// Get the containtment paths for the document
String [] containmentPaths = doc.getContainmentPaths();
// Print the paths
System.out.println("Containment paths:");
for (int lp = 0; lp < containmentPaths.length; lp++)
  {
     System.out.println("\t" + containmentPaths[lp]);
  }
```

For more information about the ContainableObject interface, refer to the *CS Java Connector v.3.0 Javadoc*.

Containment Code Samples

This section provides runnable code samples that demonstrate ways of working folders and objects that implement the ContainableObject interface. For these samples to work in your development environment, you must modify hard-coded references to servers, object stores, documents, and other resources stored on a Content Services server. You must also create a Session object to log into an object store.

The following code samples are included:

- Listing 42 creates a new folder.
- Listing 43 files and unfiles a document.
- Listing 44 retrieves all Document and Folder objects from a folder and prints the objects' properties.
- Listing 45 moves a subfolder from one folder to another.
- Listing 46 prints the containment paths of a specified document, and gets the folders that the document is stored in and prints the properties of each folder.
- Listing 47 exercises various XML-return methods for a Folder object.

Listing 42: containmentAddSubFolder code sample

```
// Get the root folder and create a subfolder under it.
public void containmentAddSubFolder()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
                     sess);

   Folder rootFolder = (Folder) os.getRootFolder();

   String subFolderName = "A New Subfolder";
   String folderComment = "Subfolder under Root";

   // Create the properties for the new subfolder.
   com.filenet.wcm.api.Properties fldProps = ObjectFactory.getProperties();
   Property fldComment = ObjectFactory.getProperty
                        (com.filenet.Panagon.Folder.idmComment);

   // Set a value for the property and add it to the Properties collection.
   fldComment.setValue(folderComment);
   fldProps.add(fldComment);

   // Create the permissions for the new subfolder
   Permissions perms = ObjectFactory.getPermissions();

   Permission newPerm = ObjectFactory.getPermission
                                     (com.filenet.Panagon.Permission.idmAccessViewer,
                                       Permission.TYPE_ALLOW,
                                       "agroup",
                                       BaseObject.TYPE_GROUP);
   perms.add(newPerm);

   newPerm = ObjectFactory.getPermission
                                      (com.filenet.Panagon.Permission.idmAccessOwner,
                                       Permission.TYPE_ALLOW,
                                       "General Users",
                                       BaseObject.TYPE_GROUP);
   perms.add(newPerm);

   newPerm = ObjectFactory.getPermission
                                     (com.filenet.Panagon.Permission.idmAccessAuthor,
                                       Permission.TYPE_ALLOW,
                                       "auser",
                                       BaseObject.TYPE_USER);
   perms.add(newPerm);

   newPerm = ObjectFactory.getPermission
                                      (com.filenet.Panagon.Permission.idmAccessOwner,
                                       Permission.TYPE_ALLOW,
                                       "tester",
                                       BaseObject.TYPE_USER);
   perms.add(newPerm);

   // Create the new subfolder
   Folder subFldr = rootFolder.addSubFolder(subFolderName, fldProps, perms);
}
```

Listing 43: containmentFileUnFile code sample

```
/* 1) Prints the containment paths of a specified document;
   2) Files the document into another folder and reprints the containment paths;
   3) Unfiles the document into another folder and reprints the containment paths.*/
public void containmentFileUnFile()
{
     String ObjectStoreId = "pin2ms^Pinta";
    ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
sess);
     Document doc     = (Document) os.getObject(Document.TYPE_DOCUMENT, "003682302" );

     // Returns containment paths of document
     String [] containmentPaths = doc.getContainmentPaths();
     System.out.println("Containment paths:");
     for (int lp = 0; lp < containmentPaths.length; lp++)
     {
         System.out.println("\t" + containmentPaths[lp]);
     }

     ////////////////////////////////////
     System.out.println("Filing document into folder: 1013124602");
     Folder fld = (Folder) os.getObject(Folder.TYPE_FOLDER, "1013124602");
     doc.file(fld, false);
     containmentPaths = doc.getContainmentPaths();
     System.out.println("Containment paths after filing into the folder:");
     for (int lp = 0; lp < containmentPaths.length; lp++)
     {
        System.out.println("\t" + containmentPaths[lp]);
     }

     ////////////////////////////////////////////
     System.out.println("Unfiling document from folder");
     doc.unfile(fld);
     containmentPaths = doc.getContainmentPaths();
     System.out.println("Containment paths after Unfiling into the folder:");
     for (int lp = 0; lp < containmentPaths.length; lp++)
     {
        System.out.println("\t" + containmentPaths[lp]);
     }
}
```

Listing 44: containmentGetContainees code sample

```
/* 1) Gets a specified folder and prints its name and ID;
   2) Gets Document and Folder objects from the folder, along with specified properties
of the objects;
   3) Iterates through the objects and prints properties for each object. */
public void containmentGetContainees()
{
    String ObjectStoreId = "pin2ms^Pinta";
    ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
                      sess);

    sess.changeGroup("General Users");

    Folder fld = (Folder) os.getObject(Folder.TYPE_FOLDER, "988143960");

    String fldName = fld.getName();
    String fldId = fld.getId();

    System.out.println("Folder name:" + fldName);
    System.out.println("Folder id:" + fldId);

    int[] types = new int[2];
    types[0] = Document.TYPE_DOCUMENT;
    types[1] = Folder.TYPE_FOLDER;

    String [] descNames = new String[2];
    descNames[0] = "com.filenet.Panagon.Document.idmDocOwner";
    descNames[1] = "com.filenet.Panagon.Document.idmComment";
    BaseObjects objs = fld.getContainees(types, descNames);

    if (objs == null)
        System.out.println("Folder getContainees returned NULL!!!");

    Iterator iterator = objs.iterator();

    while(iterator.hasNext())
    {
        BaseObject bo = (BaseObject) iterator.next();

        if (bo.getObjectType() == bo.TYPE_DOCUMENT)
        {
            Document doc = (Document) bo;
            try
            {
                String Id = doc.getPropertyStringValue(Document.idmId);
                String name = doc.getPropertyStringValue(Document.idmName);
              String dateCreated = doc.getPropertyStringValue(Document.idmDateAdded);
                String lastModifier =
                   doc.getPropertyStringValue(Document.idmModifiedByUser);
                String dateLastMod =
                   doc.getPropertyStringValue(Document.idmDateModified);
```

continued on next page

Listing 44: containmentGetContainees code sample *(continued)*

```
                System.out.println("*** Id:" + Id + " ***");
                System.out.println("*** name:" + name + " ***");
                System.out.println("*** dateCreated:" + dateCreated + " ***");
                System.out.println("*** lastModifier:" + lastModifier + " ***");
                System.out.println("*** dateLastMod:" + dateLastMod + " ***");
            }
            catch( PropertyNotFoundException e)
            {
                e.printStackTrace();
            }
            printProperties((ReadableMetadataObject) doc);
        }
        else if (bo.getObjectType() == bo.TYPE_FOLDER)
        {
            Folder subFld = (Folder) bo;
            String subFldName = subFld.getName();
            String subFldId   = subFld.getId();

            System.out.println("Subfolder Name: " + subFldName + "\tId:" + subFldId);
            printProperties( (ReadableMetadataObject) subFld);
        }
        else
            System.out.println("Unknown object type:" + bo.getId() );
    }
}


public void printProperties(ReadableMetadataObject ro)
{
   com.filenet.wcm.api.Properties props = null;
   try
   {
      props = ro.getProperties();
   }
   catch (Exception e)
   {
      return;
   }

   for (int ip=0; ip < props.size(); ++ip)
   {
      Property prop = (Property)props.get(ip);
      System.out.println("\tProperty Name:" + prop.getName() + "\tValue:" +
        prop.toString());
   }
}
```

Listing 45: containmentFolderMove code sample

```
// Gets a source folder, gets a destination folder, then moves the source folder to
   the destination.
 public void containmentFolderMove()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
     sess);

   Folder fldSrc  = (Folder) os.getObject(Folder.TYPE_FOLDER, "1013800640" );
      System.out.println("Name of folder: "+fldSrc.getName());
   System.out.println("Original parent folder: "+fldSrc.getParentFolder().getName());

   Folder fldDesc = (Folder) os.getObject(Folder.TYPE_FOLDER, "1013124602" );
   fldSrc.move(fldDesc);
   System.out.println("New parent folder: "+fldSrc.getParentFolder().getName());
}
```

Listing 46: containmentGetContainmentPaths code sample

```
/*  1) Prints the containment paths of a specified document.
    2) Gets the folders that the document is stored in and prints the properties of
       each folder.
*/
public void containmentGetContainmentPaths()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
     sess);

   Document doc     = (Document) os.getObject(Document.TYPE_DOCUMENT, "003682302" );
   String [] containmentPaths = doc.getContainmentPaths();

   System.out.println("Containment paths:");
   for (int lp = 0; lp < containmentPaths.length; lp++)
   {
      System.out.println("\t" + containmentPaths[lp]);
   }

   Folders objs = doc.getContainers();

   Iterator iterator = objs.iterator();

   while(iterator.hasNext())
   {
       System.out.println("*** Document is filed in folder: ***");
       BaseObject bo = (BaseObject) iterator.next();
       printProperties( (ReadableMetadataObject) bo);
    }
}

public void printProperties(ReadableMetadataObject ro)
{
   com.filenet.wcm.api.Properties props = null;
   try
   {
      props = ro.getProperties();
   }
   catch (Exception e)
   {
      return;
   }

   for (int ip=0; ip < props.size(); ++ip)
   {
      Property prop = (Property)props.get(ip);
      System.out.println("\tProperty Name:" + prop.getName() + "\tValue:" +
        prop.toString());
   }
}
```

Listing 47: containmentFolderXMLOutput sample code

```
// Exercies various XML-return methods for a Folder object.
public void containmentFolderXMLOutput()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
                     sess);
   Folder fld     = (Folder) os.getObject(Folder.TYPE_FOLDER, "988143960" );

   String fldName = fld.getName();
   String fldId   = fld.getId();
   System.out.println("Folder name:" + fldName);
   System.out.println("Folder id:" + fldId);

   // getContaineesXML for the following objects. Get all properties.
   int[] types = new int[3];
   types[0] = Document.TYPE_DOCUMENT;
   types[1] = Folder.TYPE_FOLDER;
   types[2] = StoredSearch.TYPE_STORED_SEARCH;
   String xml = fld.getContaineesXML(types, (String[]) null);
   System.out.println("Folder getContaineesXML:\n" + xml);

   // getParentFolderXML
   xml = fld.getParentFolderXML();
   System.out.println("Folder getParentFolderXML:\n" + xml);

   // getPermissionsXML
   String fldPerms = fld.getPermissionsXML();
   System.out.println("Folder getPermissionsXML:\n" + fldPerms);

  // getPropertiesXML, derviced from ReadableMetadataObject. Retrieve all properties.
   String fldProps = fld.getPropertiesXML((String[])null);
   System.out.println("Folder getPropertiesXML:\n" + fldProps);
}
```

Documents

A document — a word processing file, a graphic, a spreadsheet, an HTML file, etc. — is stored either in the object store's (library's) content store or stored externally. A Document object represents a single version of a document. In Content Services, a Document object is stored as a row in a database. The content in a document is the content element in a Document object.

A Document object possesses the following characteristics:

- You can retrieve a Document object or a collection of Document objects, each of which represents a version of the document. See "Retrieving Document Objects" on page 84.
- A Document object implements the ReadableMetadataObject interface, which defines methods to retrieve a document's properties and to refresh cached properties. See "Retrieving Document Properties" on page 85.
- A Document object can have zero or one content elements associated with it. If a document has content, the content is either stored in a Content Services content store (referred to as a content transfer element) or stored externally (a content reference element). For content-carrying documents, the CS Java Connector supports retrieving a document with one content element. For more information, see "Retrieving a Document's Content" on page 86.
- You can create a Document object, set its content, check it in, and file it into a folder. See "Creating a Document Object" on page 86.
- The Document class implements the `getClassDescription()` method, which returns the class assigned to a specific document. See "Document.getClassDescription( )" on page 65. The Document class also implements changeClass(...) method, which changes the class with which a Document object is associated in the object store. "Changing a Document's Class" on page 88.
- The Document class implements the `getContainers()` method, which returns the folders into which a document is filed. In addition, the Document interface extends the ContainableObject interface, which includes methods for filing and unfiling a document, returning the folders into which a document is filed, and retrieving the containment paths where a document is filed. For more information, see "Containment" on page 71.
- The Document interface extends the VersionableObject interface for checking documents in and out, getting multiple versions of a document, and returning the reservation object of a checked-out document.

  In addition, you can get a Document's VersionSeries object, which represents multiple versions of the document. The VersionSeries object allows you to determine whether your application is working with the latest version of a document. It also includes many of the same methods for acting on previous versions of a document that the Document object includes for acting on a current version of a document. For more information, see "Versioning" on page 98.
- A document is independently securable on a Content Services server. In the CS Java Connector, the Document interface extends the ReadableSecurityObject interface, which defines methods for retrieving access rights to a Document object defined in the object's Access Control List (ACL). See "Security" on page 31.
- The Document class implements the setVersionIndex() method, which marks the version for indexing in the content search server. The indexing action will be queued and the version will be indexed the next time the indexing daemon runs. For more information, see "Indexing a Document" on page 88.

- The Document class implements the setVersionDeindex() method, which marks the version for de-indexing in the Content Search server. The deindexing action will be queued and the version will be de-indexed the next time the indexing daemon runs. For more information, see "De-indexing a Document" on page 88.
- The Document class implements the getVersionIndexingState() method, which returns the actual indexing state of the document. For more information, see "Getting Indexing state of a Document" on page 89.
- The Document class implements the isVersionIndexed() method, which returns true if the document is indexed else it returns false. For more information, see "Getting Indexing status of a Document" on page 89.
- For code samples on using the Document object, see page 89.

Retrieving Document Objects

You can retrieve a Document object by calling the `getObject()` method on an ObjectStore object, which returns a document stored in the object store. For example:

```
Document doc = (Document) os.getObject(Document.TYPE_DOCUMENT, "003677005" );
```

Other ways to retrieve a Document object are as follows:

- By calling the `getReservation()` or `checkout()` method on an object that implements the VersionableObject interface, both of which return a reservation object. See "Retrieving a Reservation Object" on page 100.
- By calling the `getCurrentVersion()` method on a VersionSeries object, which returns the current document version in the version series. See "Retrieving the Current Document in the Version Series" on page 102.

You can retrieve a Documents collection in the following ways:

- By calling the `getVersions()` method on a VersionableObject. See "Retrieving All Documents in a Version Series" on page 102.
- By calling the `getContainees()` method on a Folder object, which returns the documents filed in a folder. See "Retrieving Objects from a Folder" on page 73.
- By calling one of the overloaded `getCheckOutList` methods on an ObjectStore object, which returns the documents checked out by the logged-in user. See "Retrieving Checked-Out Documents" on page 101.

From a Documents collection, you can:

- Use java.util.Collection and java.util.List methods to retrieve a Document from the collection.
- Call `filterByProperty()` on the collection to retrieve a new Documents collection, filtered by a specific property.
- Call `findByProperty()` on the collection to retrieve a single Document, filtered by a specific property.

Retrieving Document Properties

The ReadableMetadataObject interface, which the Document interface extends, has methods to retrieve a document's properties and to refresh cached properties.

With the overloaded `getProperties` method, you can retrieve all of the Document properties or a specified set of properties. The code fragment below shows how a String array is used to retrieve selected properties of a Document object. After being populated with the desired properties, the array is passed in the `getProperties(propNames)` call, and the method returns a Properties collection. Once you have the Properties collection, you can iterate through it and get the values of the specified properties (not shown).

Listing 48: Retrieving Document object properties

```
...
String ObjectStoreId = "pin2ms^Pinta";
ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
sess);
Document doc = (Document) os.getObject(Document.TYPE_DOCUMENT, "003676270");

// Create string Array
   String [] descNames = new String[3];

   // Populate array with document properties to retrieve
   descNames[0] = com.filenet.Panagon.Document.idmName;
   descNames[1] = com.filenet.Panagon.Document.idmComment;
   descNames[2] = com.filenet.Panagon.Document.idmDateAdded;

   com.filenet.wcm.api.Properties docProps = null;
   try
   {
      // Get the properties
      docProps = doc.getProperties(descNames);
   }
   catch (Exception e)
   {
      return;
   }
...
```

In addition, you can call a number of getProperty[*type*]Value methods, to return the value of a named property. (The property's data *type* is reflected in the type part of the method name; for example, getPropertyBooleanValue.) For example:

```
// Get the date and time the document's current version was added to the object store
String curVerDate =
    vDoc.getPropertyStringValue(com.filenet.Panagon.Document.idmDocCurVerDate);
```

For more information about properties, refer to "Properties" on page 52.

Retrieving a Document's Content

To retrieve a document's content, use the Document object's `getContent()` method. When you call `getContent()`, the method returns a TransportInputStream that contains a single content element, as shown in the following statements. (For a code sample, see Listing 49.)

```
Document doc = (Document) os.getObject(Document.TYPE_DOCUMENT, "003677005" );
TransportInputStream ts = doc.getContent();
```

To conserve disk space, the CS Java Connector deletes downloaded files based on an expire time, which begins as soon as the file is downloaded and lasts for five minutes. When the expire time has been met, the file is deleted. If a subsequent request is made for a file that already exists in the download cache, the expire time is reset.

Note: Make sure that you close the transport input stream (`TransportInputStream.close()`). Otherwise, the delete will fail while the transport input stream is holding the file open, and the file will remain open until the Java garbage collector cleans up the orphaned transport input stream.

By default, downloaded files are cached to the current working directory. You can, however, set the cache directory structure. See "Setting Cache Directory Structure" on page 20.

TransportInputStream also has methods to return the content's MIME type, size, element number, and filename. Similarly, the Document class includes methods that provide content element information. You can return the filename for a document's content element by calling `getFileName()` method on the Document object. To determine the type of content elements associated with a document, call `getContentElementObjectTypes()` on the Document object, which returns an array that contains the object types (ContentTransfer or ContentReference) for the content elements. And with `getContentReferenceMimetype()`, you can return the MIME type of a content reference element.

Creating a Document Object

You can create a Document object with the `objectStore.createObject(...)` method, as shown in the following code fragment:

```
Document DocumentObj = (Document)objStore.createObject("Doc-Basic", props,
perms);
```

The first parameter specifies a class on the Content Services server. The second parameter is a Properties collection containing the properties to assign to the created object. You are not required to set any system properties when you create a document, although the document's class might include custom properties that require setting.

You must pass a valid Permissions collection object to the createObject and method. A valid collection is one that contains elements or is null. If you specify null for the collection, the user's Default Item Access List is applied. If the user's Default Item Access is empty, then the user gets Owner access. An empty collection is invalid.

Note that it's possible for a user to create a Document object that the user has insufficient permission to access. This situation could occur if you assign permissions in the createObject(...) call that are subsequently insufficient for the user to access the object just created. It can also occur if the user's Default Item Access List does not include the user's ID in the list, thereby excluding the user from access. On the other hand, the Default Item Access List may provide access to a group to which the user belongs.

The createObject method returns a reference to a Document object that is initially a reservation object with permissions and properties but without content. After you create the document using createObject, you can optionally set its content, check it in, and file it into a folder.

The following sections show code fragments for the various steps you would typically take to create a document, set its content, check it into the Content Services object store, and file it into a folder.

Set Properties

You are not required to set system properties, but you do have to set other properties that require settings. The following code fragment shows setting a document's DocumentTitle property.

```
// Create an empty Properties collection
Properties docProps = ObjectFactory.getProperties();
// Create a new Property object, then set its value and add to collection
Property docName = ObjectFactory.getProperty(Property.DOCUMENT_TITLE);
docName.setValue("My New Document");
docProps.add(docName);
```

To work with a class with required settings, you'll want to retrieve the ClassDescription object and filter out properties that require settings. See "Retrieving ClassDescription Objects" on page 64 and "Retrieving Metadata with PropertyDescriptions" on page 58.

Create the Document

The `createObject` method returns a reference to a Document object that is initially a reservation object with permissions and properties but without content. The reservation object essentially records the intention to check in the document. A reservation object is not a separate class of object. Rather, the reservation object represents the new, unchecked-in version of the document.

The following code fragment uses the createObject method. The newly-created document will belong to the Doc-Basic class and will inherit its properties and permissions from this class. You can optionally set permissions for the new object when you create it. See "Retrieving and Setting Permissions" on page 35.

```
// Create the object (returned newDoc is a reservation object);
// pass in null for permissions to use user's Default Item Access List.
   Document newDoc = (Document)objectStore.createObject("Doc-Basic", docProps, null);
```

After a successful call to createObject, the values for the following Document properties are set:

- VersionStatus is set to VersionableObject.VERSION_STATUS_RESERVATION
- MajorVersionNumber is set to 0

You can return a reference to a reservation object with the `VersionableObject.getReservation()` call.

Set Content and Check in the Document

After you create the new Document object, you can optionally set content for it. The following code fragment shows setting a single content transfer element and checking in the document.

```
// Set a single content element from a file
   TransportInputStream docContent = new TransportInputStream(
      new java.io.FileInputStream("C:\\README.txt"));
   docContent.setFilename("A Readme File");
// Set doCheckin to true to check in the doc as a major version on the same
// roundtrip as the setContent call; set doAutoClassify to false to disable
// automatic document classification
   newDoc.setContent(docContent, true, false);
```

In the `setContent(...)` call in the code fragment, the `doCheckin` parameter is set to true. Checking in the document via `setContent` saves a round trip to the Content Engine server. (Alternatively, you can set the `doCheckin` parameter to false and call the derived `checkin(...)` method.)

Note: The third parameter in the `setContent(...)` call is ignored by the CS Java Connector.

File the Document into a Folder

After you create the document, you can optionally file it into a folder. For more information, see Filing an Object into a Folder.

```
// File the document into a folder with resolveUniqueness parameter set to true
   Folder aFolder = (Folder)objectStore.getObject(BaseObject.TYPE_FOLDER,
      "/My Folder/AnotherFolder");
   newDoc.file(aFolder, true);
```

Changing a Document's Class

With the `changeClass(...)` method, you can change the class with which a Document object is associated in the object store. The new class and the old class must have the Document base class type.

Since the required properties for the two classes might differ, this method takes an optional Properties collection as one of its parameters. The collection can be empty or the parameter can be null if no changes are required. This method also takes an optional Permissions collection as a parameter. If this parameter is null, the Document object's security is not modified.

For a code sample, see Listing 54 on page 95.

Indexing a Document

A document needs to be indexed before it could be searched using Content Based Retrieval (CBR) feature. You can index a document in Content Search server with setVersionIndex() method, as shown in the following code fragment:

```
Document documentObject = (Document)
objectStores.getObject(Document.TYPE_DOCUMENT, "003677005" );
documentObject.setVersionIndex();
```

The indexing action will be queued and the version will be indexed the next time the indexing daemon runs.

De-indexing a Document

You can de-index a document in content search server with setVersionDeIndex() method, as shown in the following code fragment:

```
Document documentObject = (Document)
objectStores.getObject(Document.TYPE_DOCUMENT, "003677005" );
documentObject.setVersionDeindex();
```

The de-indexing action will be queued and the version will be de-indexed the next time the indexing daemon runs.

Getting Indexing state of a Document

You can get the actual indexing state of a document with getVersionIndexState() method, as shown in the following code fragment:

```
Document documentObject = (Document)
objectStores.getObject(Document.TYPE_DOCUMENT, "003677005" );
int indexingState = documentObject.getVersionIndexState();
```

The getVersionIndexState() method returns one of these integer values from Document interface:

```
idmVerCsiStatusNotIndexed, idmVerCsiStatusIndexed,
idmVerCsiStatusIndexPending, idmVerCsiStatusReindexPending,
idmVerCsiStatusIndexLost, idmVerCsiStatusDeindexPending,
idmVerCsiStatusFilterFail, idmVerCsiStatusIndexFail.
```

Getting Indexing status of a Document

You can get the indexing status of a document with isVersionIndexed() method, as shown in the following code fragment:

```
Document documentObject = (Document)
objectStores.getObject(Document.TYPE_DOCUMENT, "003677005" );
boolean indexingStatus = documentObject.isVersionIndexed();
```

This function returns true if the document is indexed else it returns false.

Note: This method considers the document to be indexed. If the indexing state of the document is idmVerCsiStatusIndexed or idmVerCsiStatusIndexPending or idmVerCsiStatusReindexPending, then the method returns true. Otherwise, for the rest of the indexing states, the document is considered as not indexed and the methods returns false.

Document Code Samples

This section provides runnable code samples that demonstrate ways of working with the Document object. For these samples to work in your development environment, you must modify hard-coded references to servers, object stores, documents and other resources stored on a Content Services server. You must also create a Session object to log into an object store.

The following code samples are included:

- Listing 49 downloads the content element of a document.
- Listing 50 determines if a specified document is checked out.
- Listing 51 prints specified properties of a document.
- Listing 52 prints specified document properties in an XML format.
- Listing 53 creates a new document and files it into a folder.
- Listing 54 changes the class of a document.
- Listing 55 marks the document for indexing.
- Listing 56 marks the document for de-indexing.

Listing 49: documentDownloadContent code sample

```
// Downloads the content element of a document
public void documentDownloadContent()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
sess);
   Document doc     = (Document) os.getObject(Document.TYPE_DOCUMENT, "003677005" );
   TransportInputStream ts = doc.getContent();

   System.out.println("TransportInputStream File Name: " + ts.getFilename() + "\n"
         + "Document Name: " + ts.getFilename() + "\n"
         + "Size: " + ts.getContentSize() + "\n"
         + "Content type: " + ts.getMimeType() );

   String destPath = "c:\\FileNet\\CSJavaApi\\testApplication\\";

   try
   {
      FileOutputStream outToFile = new FileOutputStream(destPath.toString()
        +doc.getFilename());
      int i;
      do
      {
          i = ts.read();
          if(i != -1) outToFile.write(i);
      } while (i != -1);
            ts.close();
            outToFile.close();
            System.out.println("\n*** File Download Successful ***");
   }
   catch (IOException e) {
      System.out.println("File Download Error");
   }
}
```

Listing 50: documentCheckoutStatus code sample

```
// Determines if a specified document is checked out
public void documentCheckoutStatus()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
     sess);
   Document doc     = (Document) os.getObject(Document.TYPE_DOCUMENT, "003677005" );

   System.out.println("Document name: " + doc.getName());
   System.out.println("Document id: " + doc.getId());

   String idmCheckoutUser = "None at this time";
   String idmCheckoutWsAddress = "None at this time";
   int idmDocCurVerCheckedOut = 0;
   try
   {
      idmDocCurVerCheckedOut =
        doc.getPropertyIntValue(com.filenet.Panagon.Document.idmDocCurVerCheckedOut);
      if (idmDocCurVerCheckedOut == 1)
      {
         System.out.println("Document checked out");
         idmCheckoutUser =
           doc.getPropertyStringValue(com.filenet.Panagon.Document.idmCheckoutUser);
         idmCheckoutWsAddress =
       doc.getPropertyStringValue(com.filenet.Panagon.Document.idmCheckoutWsAddress);
      }
      else System.out.println("Document not checked out");
    }
    catch( PropertyNotFoundException e)
    {
      System.out.println(e.toString());
    }

    System.out.println("idmCheckoutUser: " + idmCheckoutUser);
    System.out.println("idmCheckoutWsAddress: " + idmCheckoutWsAddress);
}
```

Listing 51: documentGetProperties code sample

```
// Prints specified properties of a document
public void documentGetProperties()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
     sess);
   Document doc = (Document) os.getObject(Document.TYPE_DOCUMENT, "003676270" );

   // Create a String array
   String [] descNames = new String[7];
   descNames[0] = com.filenet.Panagon.Document.idmDocOwner;
   descNames[1] = com.filenet.Panagon.Document.idmDocKeywords;
   descNames[2] = com.filenet.Panagon.Document.idmComment;
   descNames[3] = com.filenet.Panagon.Document.idmVerFileDate;
   descNames[4] = com.filenet.Panagon.Document.idmDateAccessed;
   descNames[5] = com.filenet.Panagon.Document.idmDateAdded;
   descNames[6] = com.filenet.Panagon.Document.idmDateModified;

   System.out.println("Name: "+doc.getName());
   System.out.println("Id: "+doc.getId());
   System.out.println("ObjectStoreID: "+doc.getObjectStoreId());

   com.filenet.wcm.api.Properties props = null;

   try
   {
      System.out.println("idmId:
              "+doc.getPropertyStringValue(com.filenet.Panagon.PanagonObject.idmId));
      // Get the properties
      props = doc.getProperties(descNames);
   }
   catch (PropertyNotFoundException e)
   {
     System.out.println(e.getMessage());
   }

   for (int ip=0; ip < props.size(); ++ip)
   {
      Property prop = (Property)props.get(ip);
      System.out.println("Property Name:" + prop.getName() + "\tValue:"
                         + prop.getValue());
   }
}
```

Listing 52: documentGetPropertiesXML code sample

```
//Prints specified document properties in an XML format
public void documentGetPropertiesXML()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
     sess);
   Document doc     = (Document) os.getObject(Document.TYPE_DOCUMENT, "003676270" );

   String [] descNames = new String[7];
   descNames[0] = "com.filenet.Panagon.Document.idmDocType";
   descNames[1] = "com.filenet.Panagon.Document.idmDocKeywords";
   descNames[2] = "com.filenet.Panagon.Document.idmComment";
   descNames[3] = "com.filenet.Panagon.Document.idmDateAccessed";
   descNames[4] = "com.filenet.Panagon.Document.idmDateAdded";
   descNames[5] = "com.filenet.Panagon.Document.idmDateModified";
   descNames[6] = "com.filenet.Panagon.Document.idmVerFileDate";

   //Create PropertyDescriptions collection with property elements
   PropertyDescriptions desc = ObjectFactory.getPropertyDescriptions(descNames);
   String docProps = doc.getPropertiesXML(desc);
   System.out.println("Document getPropertiesXML:\n" + docProps);
}
```

Listing 53: createDocument code sample

```
// Creates a new document with two required properties, title and comment.
   The values for these properties are passed in. The sample also checks in the
   document and saves it in a folder. Default class permisssions are used.
public void createDocument(sTitle,sComment)
{
   com.filenet.wcm.api.Properties props = null;
   com.filenet.wcm.api.Properties docProps =
      com.filenet.Panagon.ObjectFactory.getProperties();
   Property docTitle =
      com.filenet.Panagon.ObjectFactory.getProperty(Property.DOCUMENT_TITLE);
   docTitle.setValue("My New Document");
   docProps.add(docTitle);

   Property docComment =
      com.filenet.Panagon.ObjectFactory.getProperty(
         com.filenet.Panagon.Document.idmComment);
   docComment.setValue("My New Comment");
   docProps.add(docComment);

   Document nDoc = (Document) os.createObject("QA-Basic", docProps, null);

   TransportInputStream t = null;
   String filename = "results.xml";
   try
   {
      t = new TransportInputStream(new FileInputStream(filename));
   }
   catch (Exception e)
   {
      e.printStackTrace();
   }

   nDoc.setContent(t, false, false);

   nDoc.checkin(false);

   Folder fldr = (Folder) os.getObject(Folder.TYPE_FOLDER, "1027700912" );
   nDoc.file(fldr, true);

   System.out.println("New Document:" + nDoc.getId() + " was created,checked in,
      and filed");
}
```

Listing 54: documentChangeClass code sample

```
// Changes the class of a document and applies new permissions.
// The properties are not changed.
public void documentChangeClass()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(
                     ObjectStoreId, sess);

   com.filenet.Panagon.Document doc = (com.filenet.Panagon.Document)
                  os.getObject(Document.TYPE_DOCUMENT, "003676270" );

   System.out.println("Existing Document Class is
                       "+doc.getClassDescription().getName());

   // Set up the permissions to pass in.
   Permissions perms = ObjectFactory.getPermissions();

   Permission newPerm =
    ObjectFactory.getPermission(com.filenet.Panagon.Permission.idmAccessViewer,
                                                    Permission.TYPE_ALLOW,
                                                    "agroup",
                                                    BaseObject.TYPE_GROUP);
   perms.add(newPerm);

   newPerm =
    ObjectFactory.getPermission(com.filenet.Panagon.Permission.idmAccessOwner,
                                                    Permission.TYPE_ALLOW,
                                                    "General Users",
                                                    BaseObject.TYPE_GROUP);
   perms.add(newPerm);

   newPerm =
    ObjectFactory.getPermission(com.filenet.Panagon.Permission.idmAccessAuthor,
                                                    Permission.TYPE_ALLOW,
                                                    "auser",
                                                    BaseObject.TYPE_USER);
   perms.add(newPerm);

   newPerm =
    ObjectFactory.getPermission(com.filenet.Panagon.Permission.idmAccessOwner,
                                                    Permission.TYPE_ALLOW,
                                                    "tester",
                                                    BaseObject.TYPE_USER    );
    perms.add(newPerm);

    // Changes the class and applies new permissions.
    // The null parameter indicates no change to the properties.
    doc.changeClass("General", null, perms);

    System.out.println("New Class is "+doc.getClassDescription().getName());
}
```

Listing 55: setVersionIndexTest code sample

```
public void setVersionIndexTest() {
     String ObjectStoreId = "pin2ms^Pinta";
     ObjectStore os = com.filenet.Panagon.ObjectFactory.getObjectStore(
     ObjectStoreId, sess);

     // protected document - Word File
     com.filenet.Panagon.Document doc = (com.filenet.Panagon.Document)
     os.getObject(Document.TYPE_DOCUMENT, "003670062" );
     doc.setVersionIndex();


     // external document
     doc =
     (com.filenet.Panagon.Document)os.getObject(Document.TYPE_DOCUMENT,
     "003670257");
     doc.setVersionIndex();
}
```

Listing 56: setVersionDeindexTest code sample

```
public void runVersionDeindexTest() {
     String ObjectStoreId = "pin2ms^Pinta";
     ObjectStore os = com.filenet.Panagon.ObjectFactory.getObjectStore(
     ObjectStoreId, sess);

     // protected document - Word File
     com.filenet.Panagon.Document doc = (com.filenet.Panagon.Document)
     os.getObject(Document.TYPE_DOCUMENT, "003670062" );
     doc.setVersionDeindex();


     // external document
     doc =
     (com.filenet.Panagon.Document)os.getObject(Document.TYPE_DOCUMENT,
     "003670257");
     doc.setVersionDeindex();
}
```

Versioning

If a document is versionable, you can check in the new document, at which point the document becomes version one in its version series. Thereafter, each time you check out and check in the document, the Content Services server creates another version of the document. The new version of the document contains properties (such as `idmVerId`) that are unique to that particular version.

A class implementing the VersionableObject interface is an object that can be versioned. In this release of the CS Java Connector, the Document class and the VersionSeries class implement the VersionableObject interface.

A VersionSeries object represents multiple versions of a document, and is especially useful for determining whether your application is working with the latest version of a document. A VersionSeries object is read-only and has no security attached to it; it is secured by the documents contained within it. A VersionSeries method operates on the latest document version.

This section shows you how to work with document versions and the VersionSeries object. It covers the following topics:

Checking Out a Document

You can call the `checkout()` method on a Document or VersionSeries object, which implement the VersionableObject interface. To successfully check out an object, the object must not be already checked out (the value of its STATE_CHECKED_OUT property is `false`). If you call the `checkout()` method on a Document object, it must be the current version (STATE_LATEST_VERSION is `true`). If you call the `checkout()` method on a VersionSeries object, the method checks out the appropriate version (the current version) in the version series. See "Retrieving a VersionSeries Object" on page 101.

The user must have the appropriate access rights to check out a document. See "Retrieving and Setting Permissions" on page 35.

For example, the following code fragment retrieves a Document object, tests if it's the current version and that it's not already reserved, then checks it out of the object store. The returned resDoc is the Document object that represents the reservation object.

```
Document doc = (Document) os.getObject(Document.TYPE_DOCUMENT, "003676270");
boolean canCheckOut =
   doc.getPropertyBooleanValue(com.filenet.Panagon.Document.STATE_CAN_CHECKOUT)
boolean isLatestVersion =
   doc.getPropertyBooleanValue(com.filenet.Panagon.Document.STATE_LATEST_VERSION)

if ((canCheckOut == true) && (isLatestVersion == true))
Document resDoc = doc.checkout();
...
```

The checkout() method returns a reference to a Document object that is initially a reservation object. The reservation object essentially records the intention to check in the document. A reservation object is not a separate class of object. Rather, the new, unchecked-in version of the document represents the reservation object, with permissions and properties but without content. To get the content, if any, see "Retrieving a Document's Content" on page 86.

Checking in a Document

You can check in a document with the CS Java Connector in the following ways:

- By calling the `checkin(...)` method on a Document object, which implements the VersionableObject interface.
- By calling the `checkin(...)` method on a Document object, which implements the Document interface.
- By calling the `setContent(...)` method on a Document object and setting the `doCheckin` parameter to true. Checking in the object via `setContent(...)` saves an additional round trip to the Content Engine server.

You check in the document that represents the reservation object (in other words, the document must either be a newly-created document that has not been checked in, or it must be a checked-out document). If a document is a reservation object, the value of its VersionStatus property is `VersionableObject.VERSION_STATUS_RESERVATION`. If you call `checkin(...)` on a VersionSeries object, the appropriate document in the VersionSeries (the document that represents the reservation object) is checked in. See "Retrieving a Reservation Object" on page 100 for ways to return a reservation object.

If the document has content, you must set its content before you check it in. See "Setting a Document's Content" on page 100.

If you need to index the document after it has been checked in, you must use the `checkin(...)` method implemented from the document interface. See "Checking in Document with indexing option" on page 100.

The user must have the appropriate access rights to check in a document. See "Retrieving and Setting Permissions" on page 35.

The following example retrieves the reservation object from a VersionSeries, then calls the `checkin(autoClassify)` method on the reservation object. The autoClassify parameter is ignored by the CS Java Connector.

```
Document resDoc = objVersionSeries.getReservation();
   if (resDoc != null)
```

```
resDoc.checkin(false); //checks in the document
```

After a successful check-in, the reservation object becomes the new current version of the document. Its IsCurrentVersion property is set to `true`, and its VersionStatus property is set to `VersionableObject.VERSION_STATUS_RELEASED`. In addition, the values of a number of document properties change after a successful check-in, for example, idmVerID, idmVerFileDate, and idmDocLastVerId.

Checking in Document with indexing option

If you need to index the document after it has been checked in, you must use the `checkin(boolean autoClassify, String options[])` method implemented from the document interface.

For example, the following code fragment retrieves a document object, tests if the user can check it in, and if yes then checks it in to the object store.

```
Document doc = (Document) os.getObject(Document.TYPE_DOCUMENT, "003676070");
boolean canCheckin =
   doc.getPropertyBooleanValue(com.filenet.Panagon.Document.STATE_CAN_CHECKIN)

String options = new String[]{com.filenet.Panagon.Document.idmOptionIndex};
if (canCheckin)
Document resDoc = doc.checkin( false, options );
...
```

Retrieving a Reservation Object

You can retrieve a Document object that represents a reservation object by:

- Calling the `checkout()` method on an object that implements the VersionableObject interface, which checks out a document and returns a reservation object (see “Checking Out a Document” on page 98).
- Calling the `getVersions()` method on an object that implements the VersionableObject interface. If the VersionableObject object is checked out, the last element in the returned collection is the reservation object (see “Retrieving All Documents in a Version Series” on page 102).
- Calling the `getReservation()` method on an object that implements the VersionableObject interface. An example of calling the `getReservation()` method is shown below.

For example, the following code fragment shows calling `getReservation()` on a Document object:

```
Document doc = (Document)ObjectStore.getObject(BaseObject.TYPE_DOCUMENT, "/Folder/
SomeDoc");
if (doc != null)
{
    Document reservedDoc = doc.getReservation();
    if (reservedDoc != null)
    {
      ...
```

Setting a Document's Content

For a document checked out from Content Services, the reservation object initially has no content. Therefore, you must set its content before you can check in the document. When you set the content, you are setting the content on the next version of the document that the reservation object represents.

Note: You cannot set content on an external document.

To set the content element (known as a content transfer element, you use the Document interface's `setContent(...)` method), passing in an instance of a TransportInputStream, which is a stream that contains a single content element. TransportInputStream includes methods to set the following content attributes:

- Size - You are not required to supply the size unless you are calling `setContent(...)` from a client (not an Application Server) and you construct the TransportInputStream from any type of input stream other than a ByteArrayInputStream. In this case, you must set the content size using the `setContent(...)` method on the TransportInputStream. (The ByteArrayInputStream implicitly tells its own content size; for other types of input streams--such as FileInputStream--the stream's length cannot be easily determined.)
- Filename - If you do not supply a filename and the document does not have a document title, the filename associated with the content is set to the GUID of the document.
- Element number - The setContent method defaults to setting the first content transfer element. Note that the CS Java Connector supports only the first content transfer element.

The following code fragment sets a content transfer element for a document and checks in the document with the `setContent(...)` call.

```
...
TransportInputStream t = new TransportInputStream(new FileInputStream
                         ( "C:\\README.txt"));
t.setFilename("A Readme File");

// newDoc is a reservation objects
// Set the doCheckin parameter to true to check in this document.
// The third parameter is ignored by the CS Java API.
newDoc.setContent(t, true, false);
...
```

Retrieving Checked-Out Documents

To retrieve a collection of Document objects that are checked out by the current user, call getCheckoutList(...) on an ObjectStore object. You can optionally specify properties to retrieve for each document in the collection. For example:

```
...
// Populate a String[] with the properties to retrieve
String[] propNames = new String[] {Document.idmName};
Documents checkedOutDocs = objectStore.getCheckoutList(propNames);
Iterator pi = checkedOutDocs.iterator();
while (pi.hasNext())
{
   Document aDocument = (Document) pi.next();
   System.out.println(aDocument.getPropertyStringValue(Document.idmName) + "\n");
}
...
```

Retrieving a VersionSeries Object

You cannot create a new VersionSeries object but you can instantiate one in the following ways:

- Call the `getVersionSeries()` method on a Document object, for example:

```
VersionSeries objVS = doc.getVersionSeries();
```

- Call the `getObject()` method on an ObjectStore object.

```
VersionSeries objVS = (VersionSeries)objectStore.getObject
    (BaseObject.TYPE_VERSIONSERIES,"03676956");
```

Retrieving the Current Document in the Version Series

To retrieve the current Document object in a version series, call `getCurrentVersion()` on the VersionSeries object, for example:

```
Document curDoc = objVS.getCurrentVersion();
```

Retrieving All Documents in a Version Series

To retrieve all documents in a version series, call `getVersions()` on a VersionableObject object, which returns a Documents collection. Many properties are returned by default for each object in the collection, including: `IdmVerId`, `idmName`, and `idmDocOwner`. You can optionally specify the properties to retrieve with `getVersions(String[] propName)`. You can also call an XML version of the method on the Document or VersionSeries object.

In the following code fragment, the `getVersions()` method is used to retrieve a documents collection. The code then iterates through the collection and prints specified Document property values for each version in the collection.

Listing 57: Retrieving all Documents in a version series

```
...
ObjectStore os = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
sess);
Document doc = (Document) os.getObject(Document.TYPE_DOCUMENT, "003676956");

// Get all versions of the document
Documents docs = doc.getVersions();

// Iterate through the collection and print the specified property values
Iterator iterator = docs.iterator();
while(iterator.hasNext())
{
   Document vDoc = (Document) iterator.next();
   try
   {
      String idmVerDocId =
        vDoc.getPropertyStringValue(com.filenet.Panagon.Document.idmVerDocId);
      String idmName =
        vDoc.getPropertyStringValue(com.filenet.Panagon.Document.idmName);
      String idmVerId =
        vDoc.getPropertyStringValue(com.filenet.Panagon.Document.idmVerId);
      String idmDocCurVerNum =
        vDoc.getPropertyStringValue(com.filenet.Panagon.Document.idmDocCurVerNum);

      System.out.println( "DocumentId:" + idmVerDocId +
                          " Name:" + idmName +
                          " VersionId:" + idmVerId +
                          " CurrentVersion:" + idmDocCurVerNum);
   }
   catch( PropertyNotFoundException e)
   {
      System.out.println(e.toString());
   }
```

Retrieving VersionSeries Properties

The VersionSeries interface extends ReadableMetadataObject; therefore, a VersionSeries implementation supports the property retrieval and caching methods of ReadableMetadataObject. (For general information on properties, see page 52.)

In the following code fragment, all of the properties of a VersionSeries object are returned into a Properties collection. Iterating through the collection, the code prints out the name and value of each VersionSeries property. Because VersionSeries methods operate on the latest document version, the property values are those of the latest document version.

Listing 58: Retrieving VersionSeries properties

```
...
VersionSeries objVS = (VersionSeries)objectStore.getObject
   (BaseObject.TYPE_VERSIONSERIES,"003676956");
Properties vsProps = objVS.getProperties();
Iterator pi = vsProps.iterator();
while (pi.hasNext())
{
    com.filenet.wcm.api.Property vsProp = (com.filenet.wcm.api.Property) pi.next();
    String strPropVal = "null";
    Object objPropVal = vsProp.getValue();
    if (objPropVal != null)
        strPropVal = objPropVal.toString();
    System.out.println("Property: \"" + vsProp.getName() + "\"
        has value: \"" + strPropVal + "\"");
}
```

Deleting a VersionSeries Object

As shown in the code fragment, you can delete a VersionSeries object and all document versions in it. You must have Delete permission on the current version of the document. See “Document Access Rights” on page 34.

```
...
VersionSeries objVS = docObject.getVersionSeries();
vs.delete();
```

Versioning Code Samples

This section provides runnable code samples that demonstrate ways of working with Document and VersionSeries objects. For these samples to work in your development environment, you must modify hard-coded references to servers, object stores, documents, and other resources stored on a Content Services server. You must also create a Session object to log into an object store.

The following code samples are included:

- Listing 59 determines if the retrieved document version is the current one.
- Listing 60 retrieves a VersionSeries object and prints all of its properties.
- Listing 61 checks out a document and tests if it’s stored externally.
- Listing 62 checks out a document, sets it content, and checks it back in.
- Listing 63 checks out a document, sets it content, and checks it back in with indexing.
- Listing 64 gets a list of checked-out documents for the logged-in user.

Listing 59: versionSeriesTestCurrent code sample

```
// Determines if the retrieved document version is the current one.
public void versionSeriesTestCurrent()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
     sess);

   // Get a version of the document that is known to not be the current
   Document doc = (Document) os.getObject(Document.TYPE_DOCUMENT, "003678596:4" );

   // Get the version series
   VersionSeries vs = doc.getVersionSeries();

   /* Compare the ID of the retrieved version (idmVerId)
      with the ID of the current version (idmDocCurVerNum)*/
   try
   {
      String idmVerId =
         vs.getPropertyStringValue(com.filenet.Panagon.Document.idmVerId);
      String idmDocCurVerNum  =
         vs.getPropertyStringValue(com.filenet.Panagon.Document.idmDocCurVerNum);
      if (idmVerId.equalsIgnoreCase(idmDocCurVerNum))
         System.out.println("Version series returned is the current");
      else
         System.out.println("Version series returned is NOT the current");
   }
   catch (PropertyNotFoundException e)
   {
            System.out.println(e.toString());
   }
}
```

Listing 60: versionSeriesGetProperties code sample

```
// Retrieves VersionSeries object and prints all of its properties
public void versionSeriesGetProperties()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
     sess);

   // Get a VersionSeries object.
   VersionSeries objVS = (VersionSeries)os.getObject
    (BaseObject.TYPE_VERSIONSERIES,"003676956");

   try
   {
      com.filenet.wcm.api.Properties vsProps = objVS.getProperties();
      Iterator pi = vsProps.iterator();
      while (pi.hasNext())
      {
         com.filenet.wcm.api.Property vsProp = (com.filenet.wcm.api.Property)
pi.next();
         String strPropVal = "null";
         Object objPropVal = vsProp.getValue();
         if (objPropVal != null)
            strPropVal = objPropVal.toString();
         System.out.println("Property: \""  + vsProp.getName() + "\" has value: \"" +
            strPropVal + "\"");
      }
    }
    catch (Exception e) { }
}
```

Listing 61: documentCheckout code sample

```
// Checks out a document and tests if it is stored externally.
// If not, gets the content as a TransportInputStream object,
// which is used to retrieve information about the content.
public void documentCheckout()
{
    String ObjectStoreId = "pin2ms^Pinta";
    ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
                      sess);

    // Get the document, then check it out.
    Document doc     = (Document) os.getObject(Document.TYPE_DOCUMENT, "003684759");
    Document cDoc = doc.checkout();

    // Test if the document is stored externally.
    boolean isExt = false;
    try
    {
       isExt = cDoc.getPropertyBooleanValue
                  (com.filenet.Panagon.Document.STATE_IS_EXTERNAL);
    }
    catch (Exception e)
    {
       e.printStackTrace();
    }

    // If not stored externally, get the document's content and print
    // information about it.
    if (!isExt)
    {
       // subsequent get content call should retrieve the document from cache...
       TransportInputStream ts = cDoc.getContent();
       System.out.println("File name:" + ts.getFilename() + " Length:" +
                         ts.getContentSize() + " Content type:" + ts.getMimeType() );

       try
       {
          ts.close();
       }
       catch (Exception e)
       {
          e.printStackTrace();
       }
    }
}
```

Listing 62: documentSetContent code sample

```
// Checks out a document, sets it content, and checks it back in.
public void documentSetContent()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
                     sess);

   com.filenet.Panagon.Document doc = (com.filenet.Panagon.Document)
                                  os.getObject(Document.TYPE_DOCUMENT, "003682168");

   // If already checked out, check it back in.
   try
   {
     if (doc.getPropertyBooleanValue(com.filenet.Panagon.Document.STATE_CHECKED_OUT))
          doc.checkin(false);
   }
   catch (Exception e)
   {
      e.printStackTrace();
   }

   // Check out document.
   doc.checkout();
   System.out.println("Document checked out");

   // File to be used for new content
   String filename = "D:/content.xml";

   // Create TransportInputStream object
   TransportInputStream t = null;
   try
   {
      t = new TransportInputStream(new FileInputStream(filename));
      t.setFilename("Sample XML file.xml");
   }
   catch (Exception e)
   {
      e.printStackTrace();
   }
   // Set the content. Also check in the document, as specified by
   // "true" parameter value. (The third value is ignored.)
   doc.setContent(t, true, false);
   System.out.println("Content set\nDocument checked in");
```

Listing 63: documentSetContentWithIndexing code sample

```
// Checks out a document, sets it content, and checks it back in with indexing.
public void documentSetContentWithIndexing()
{
    String ObjectStoreId = "pin2ms^Pinta";
    ObjectStore os = com.filenet.Panagon.ObjectFactory.getObjectStore( ObjectStoreId,
                      sess);
    com.filenet.Panagon.Document doc = (com.filenet.Panagon.Document)
                            os.getObject(Document.TYPE_DOCUMENT, "003682168");
    // If already checked out, check it back in.
    try
    {
      if(doc.getPropertyBooleanValue(com.filenet.Panagon.Document.STATE_CHECKED_OUT))
      doc.checkin(false);
    }
    catch (Exception e)
    {
       e.printStackTrace();
    }
    // Check out document.
    doc.checkout();
    System.out.println("Document checked out");
    // File to be used for new content
    String filename = "D:/content.xml";
    // Create TransportInputStream object
    TransportInputStream t = null;
    try
    {
       t = new TransportInputStream(new FileInputStream(filename));
       t.setFilename("Sample XML file.xml");
    }
    catch (Exception e)
    {
       e.printStackTrace();
    }
    // Set the content. But do not check in the document, as specified by
    // "false" parameter value. (The third value is ignored.)
    doc.setContent(t, false, false);
    System.out.println("Content set");

    String options[] = new String[] {com.filenet.Panagon.Document.idmOptionIndex};
    // Checkin the document with checkin(...) method from Document interface.
    // This method will checkin the document and will take the indexing
    // action as specified in options array.
    doc.checkin( false, options );
    System.out.println("\nDocument checked in");
}
```

Listing 64: getCheckoutList

```
// Gets the list of checked-out documents for the logged-in user.
// The ObjectStore.getCheckoutList(propNames) call returns a collection of Document
// objects plus specified properties. This method prints the name of each
// checked-out document and the value of one of its properties.
// Then the ObjectStore. getCheckoutListXML(propNames) call returns a list of
// checked-out documents and specified properties in XML format.
// This method writes the string to a file.
public void getCheckoutList()
{
   String ObjectStoreId = "pin2ms^Pinta";
   ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
sess);

   String [] descNames = new String[5];
   descNames[ 0 ] = Document.idmId;
   descNames[ 1 ] = Document.idmName;
   descNames[ 2 ] = Document.idmDateAdded;
   descNames[ 3 ] = Document.idmModifiedByUser;
   descNames[ 4 ] = Document.idmDateModified;

   // Get the list of checked-out documents plus the requested properties.
   Documents docs = os.getCheckoutList(descNames);

   // if logged-in user has no checked-out files, return.
   if (docs.isEmpty())
   {
      System.out.println("No documents checked out");
      return;
   }

   // Identify documents checked out.
   Iterator iterator = docs.iterator();
   while(iterator.hasNext())
   {
      Document vDoc = (Document) iterator.next();
      System.out.println( "\nDocumentId:" + vDoc.getId() +
                                " Name:" + vDoc.getName());
      // Print the value of the idmDateModified property.
      try
      {
         System.out.println("Date last modified is
"+vDoc.getPropertyValue(descNames[5]));
      }
      catch (PropertyNotFoundException e)
      {
        System.out.println(e.getMessage());
      }
   }
```

Listing 64: getCheckoutList *(Continued)*

```
// Get checked-out documents in XML-formatted string.
   String docXML = os.getCheckoutListXML(descNames);

   // Send to file
   String fileName = "getCheckoutList.xml";
   try
   {
        java.io.OutputStream fos = new java.io.FileOutputStream(fileName);
        java.io.Writer w = new java.io.BufferedWriter(new
java.io.OutputStreamWriter(fos, "Unicode"));
        w.write(docXML);
        w.close();
   }
   catch(Exception e)
   {
        System.out.println(e.toString());
   }

   System.out.println("Document getCheckoutListXML sent to file:\n" + fileName);
}
```

Searching

In CS Java Connector 3.0 Release, documents can be searched by property values and/or their content in a single object store (library).

Note that the following search features are NOT supported:

- Stored searches
- Searching across two or more object stores
- Property name alias
- SearchPropertyDescription object

Searches are ad hoc, that is, free from queries of the Content Services Property Repository. Each search is used to find documents that have common property values and/or have common content in a specific order (in order or any order) and in a specific range. You can express the search criteria in a Structured Query Language (SQL) statment contained in an XML formatted string. The result set is returned as an XML string representation of columns and rows of data, formatted according to the Microsoft ADO XML schema.

- For information on SQL support by the CS Java Connector, see the “Supported SQL Grammar” on page 112.
- For an overview of Property Based Search,see “Property Based Search” on page 117.
- For an overview of Content Based Retrieval (CBR), see “Content Based Retrieval (CBR)” on page 119.
- For an overview of Combined Search, that is, combination of property based search and content based retrieval, see “Combined Search” on page 121.
- For detailed XML support by the CS Java Connector, see the “Execute Search XML Schema” on page 122.
- For Code Samples, see “Search Code Samples” on page 129.

Note that a Search object is created with no information about any previously defined objects. Each method of the Search class is independent of the others. That is, unlike other classes that inherit object references during the creation process, the Search class inherits no object references so has no access to objects that its methods need to perform their functions. Therefore, for each method you call on the Search object, you must specifically pass information about other objects that the method needs. Also, a Search object has no metadata properties associated with it.

Supported SQL Grammar

In your queries, use the syntax and options as specified in the following SQL SELECT statement:

```
SELECT <select_list> FROM Document WHERE <where_condition> [CONTAINS
<contains_conditions>] [INFOLDER <folder_id>] [ORDER BY <order_by_list>]

<select_list> ::=
    {
        idm_property
    }   [ ,...n ]

<where_condition> ::=
```

```
    {
        <where_expression> | (<where_condition>)
        [
            { AND|OR }
            { <where_expression> | (<where_condition>) }
            }
        ]
    }

<where_expression> ::=
    {
        idm_property
        { = | < > | > | > = | < | < = | like }
        value
    }

<contains_condition> ::=
    {
        <contains_expression> | (<contains_condition>)
        [
            { AND|OR }
            { <contains_expression> | (<contains_condition>) }
        ]
    }

<contains_expression> ::=
    {
        'word_or_phrase'
        { inorder | anyorder }
        'range'
    }

<folder_id> ::=
    {'folderid'}

<order_by_list> ::=
    {
        idm_property
} [ ,...n ]
```

Arguments

```
<select_list>
    The columns to be selected for the result set. The select list is a series of idm
    properties separated by commas.

    idm_property
    This is name of a column to be returned. For example: idmId, idmDateAdded,   etc.

<where_condition>
    Specifies the property-based conditions for the rows returned in the result set for
    a SELECT statement.

    AND
    Combines two conditions and evaluates to TRUE when both of the conditions are TRUE.

    OR
    Combines two conditions and evaluates to TRUE when either condition is TRUE.

<where_expression>
    It is an expression that returns TRUE or FALSE.
```

idm_property
It's a column name connected by an operator to an expected value.

=
It's the operator used to test the equality between two expressions.

<>
It's the operator used to test the condition of two expressions not being equal to each other.

>
It's the operator used to test the condition of one expression being greater than the other.

>=
It's the operator used to test the condition of one expression being greater than or equal to the other expression.

<
It's the operator used to test the condition of one expression being less than the other.

<=
It's the operator used to test the condition of one expression being less than or equal to the other expression.

like
It's the operator to determine, whether or not given characters string matches a specified pattern.

value
It's a string of characters with/without wildcard characters. Wildcard characters can be % for like operator.

<contains_condition>
Specifies the content-based conditions for the rows returned in the result set for a SELECT statement.

AND
Combines two conditions and evaluates to TRUE when both of the conditions are TRUE.

OR
Combines two conditions and evaluates to TRUE when either condition is TRUE.

<contains_expression>
It is an expression that represents the content-based search criteria.

word_or_phrase
It's a string of characters with/without wildcard character. Wildcard character can be asterisk (*) and question mark (?), where * represents zero or more characters in a word and the question mark represents one character in a word.

anyorder
To find words that appear in sequence with possible intervening words, use this operator and set the range to the number of characters in the intervening words, again skipping over common words and white space.

inorder
To find a word or an exact phrase, use this operator and set the range to 0. This finds the exact sequence of words, skipping over common words and white space.

range

Specifies a proximity value, which is an integer number starting from 0 through 999999999. This specifies the maximum number of words in the intervening words of a phrase.

<folder_id>
This is specified to search documents in a particular folder in Content Services.

folderid
This is the folder id of that folder, from which the documents are expected to appear in search result.

<order_by_list>
The list of columns on which the result set should be sorted. The order by list is a series of column names separated by commas.

idm_property
This is name of a column, on which the result set should be sorted. For example: idmId, idmDateAdded, etc.

For Example:

```
SELECT idmId, idmVerFileName, FROM Document WHERE idmAddedByUser = 'John Smith' INFOLDER
'00367981' ORDER BY idmId desc

SELECT idmId, idmDocType, idmDocOwner FROM Document CONTAINS 'central processing' inorder
'10' AND 'unit' anyorder '0' ORDER BY idmId desc

SELECT idmId FROM Document WHERE idmVerCsiStatus = '1' AND idmAddedByUser = 'Admin'
CONTAINS 'tree construction' anyorder '35'
```

The following rules apply:

- A valid query must contain the SELECT and FROM clauses. The CS Java Connector does NOT support EXISTS, IN, BETWEEN, and JOIN.
- The right side of the condition is always interpreted as a literal value.
- The maximum number of elements placed in a SELECT is forty-two.
- Only single-value document custom properties and single-value version custom properties can be used in the SELECT clause.
- The following reserved custom property names cannot be used in the Select clause : idmCDBehaviorId, idmPublish, idmAnnotation, and idmVWVersion. For example, if idmCDBehaviorId is implemented as idmCustom01, using idmCustom01 in the Select clause would yield an error.
- Selecting derived document properties will be ignored: idmDocIsReplica, idmDocOriginLibrary, idmDocOriginID, idmAccessLevel.
- Multi-value properties in a SELECT clause will generate an error because they are not selectable.
- Only documents can be selected (FROM Document).
- "INFOLDER" is used to optionally scope a document search to a specific folder (by folder id). It consumes WHERE and CONTAINS conditions.
- By default, comparisons are case-insensitive.
- For case-insensitive searches, you can use the standard comparison operators of =, !=, <>, >, <=, like
- For case-sensitive searches, use the following comparison operators:
  - EQ - case sensitive equal

- GT - case sensitive greater than
- LT - case sensitive less than
- NE - case sensitive NOT equal
- GE - case sensitive greater than equal
- LE - case sensitive less than equal
- LK - case sensitive like
- NL - case sensitive NOT like

For a case-insensitive comparison operator, you can add an "I" to the end of an equivalent case-sensitive operator list above, for example, "EQI" .

- The SQL "NOT" is not supported. Use the equivalent "NE" or "NL" operator.
- The wildcards characters for a like operator ("like", "LK", "LKI") are:
  - _ or ?

    Single character wildcard match, for example 98000_234 or 98000?234
  - % or *

    Multiple character wildcard match, for example, 98120% or 98120*
- Depending on the data type of the property only certain operators are valid. Property descriptions define valid data type and operator search combinations.
- Date properties are only valid with >, >=, <, <=. Therefore, you do not have to fully specify the date
- The date format must be in ISO format. See “Specifying Date and Time as Search Criteria” on page 116.
- The ORDER BY clause is an optional comma separated list of symbolic property names.
- Property Based Search can consume 59 clauses (58 conjunctions).
- Content Based Retrieval can consume one phrase of maximum 57 words. If multiple phrases are given, then maximum number word for each phrase decreases.
- Each word in CBR can be of maximum 32 characters.
- CBR can consume 59 clauses (58 conjunctions) having words (but not phrase) in each clause. If phrases are given, then maximum number of conjunction decreases.
- In CBR, the range can be any integer value between 0 to 999999999.
- To specify a single quote in a word for Content Based Retrieval, you need to give the single quote twice or you can give ' instead of single quote.
- In Property Based Search or Content Based Retrieval, maximum of 14 nested conditions can be expressed in parentheses.

Specifying Date and Time as Search Criteria

Ad hoc searches accept date and time values in ISO standard 8601 format (except to thousandths, rather than millionths):

ISO format is *yyyymmddThhmmss,ttt* where:

- *yyyy* represents the year
- *mm* represents the month
- *dd* represents the day
- *T* indicates the start of the time value
- *hh* represents the hour
- *mm* represents the minute
- *ss* represents the second
- *ttt* represents the fraction of the second

For example, 9:45:45 a.m. on July 27th, 2002 is represented in ISO format as 20020727T094545.

An ad hoc search condition that specifies an ISO format for a DateTime-type property will retrieve documents whose specified property value exactly matches the specified date and time value. You should note, however, that the fractional time component is stored with a document's DateTime-type properties and it is therefore extremely unlikely that your date/time criteria will find an exact match. Therefore, you should use the greater-than and less-than operators (>, <).

Property Based Search

You can search documents from Content Services, which have common property values. You express the search criteria in a Structured Query Language (SQL) statement contained in an XML formatted string. To perform a property based search, the search SQL should have expressions in WHERE clause.

- For information on SQL support by the CS Java Connector, see “Supported SQL Grammar” on page 112.
- For an overview of XML formatted queries for Property Based Search, see “XML Query Examples” on page 117.
- For an example result set using the ADO XML schema, see “XML Result Set Example” on page 128.
- For a code sample, see “Search Code Samples” on page 129.

XML Query Examples

The code fragments in this section introduce XML formatted queries for Property Based Search.

The following query retrieves the document id, file name and date for documents added to the searchLibrary object store by John Smith.

Listing 65: Example XML query

```
...
String searchStatement = "<request>"+
   "<objectstores>"+
   "<objectstore id=\"searchLibrary^testHost\"/>"+
   "</objectstores>"+
    "<querystatement>SELECT idmId, idmVerFileName, idmDocCurVerDate FROM Document "+
   "WHERE idmAddedByUser = 'John Smith'</querystatement>"+
   "<options maxrecords=\"100\"/>"+
   "</request>";
String resultString = oSearch.executeXML(searchStatement);
```

The following query retrieves the document id, type, owner, protection status, and date added for documents added to the pin2ms object store on March 19, 2002 at 7:03 p.m. or later.

Listing 66: Example XML query

```
...
String searchStatement = "<request>"+
   "<objectstores>"+
   "<objectstore id=\"pin2ms^Pinta\"/>"+
   "</objectstores>"+
    "<querystatement>SELECT idmId, idmDocType, idmDocOwner, idmDocProtected,"+
   "idmDateAdded FROM Document WHERE idmDateAdded >= '2002-03-19T09:19:03'"+
   "</querystatement>"+
   "<options maxrecords=\"100\"/>"+
   "</request>";
String resultString = oSearch.executeXML(searchStatement);
```

The next code fragment shows how to define an XML query string that retrieves the property values for document id, type, owner, protection status, and date added for documents that meet one of the following search criteria: 1) a document ID of 003676744 AND for all documents whose file name starts with the "a" character that have not been added by the Admin user; 2) a document ID of 003676270; 3) for all documents whose file name starts with the "a" character.

Listing 67: Example XML query

```
...
String searchStatement = "<request>"+
   "<objectstores>"+
   "<objectstore id=\"pin2ms^Pinta\"/>"+
   "</objectstores>"+
    "<querystatement>SELECT idmId, idmDocType, idmDocOwner, idmDocProtected, "+
   "idmDateAdded FROM Document "+
    "WHERE (idmId = '003676744' and (idmVerFileName like 'a%' and idmAddedByUser  "+
   "!= 'Admin') or idmId = '003676270') or idmVerFileName like 'a%' "+
   "</querystatement>" +
   "<options maxrecords=\"100\"/>"+
   "</request>";
String resultString = oSearch.executeXML(searchStatement);
```

Content Based Retrieval (CBR)

You can search documents, which have common content in order or any order, and in a specific range. You express the search criteria in a Structured Query Language (SQL) statement contained in an XML formatted string. To perform a Content Based Retrieval, the search SQL should have expressions in CONTAINS clause.

- For information on SQL support by the CS Java Connector, see “Supported SQL Grammar” on page 112.
- For an overview of XML formatted queries for Content Based Retrieval, see “XML Query Examples” on page 119.
- For an example result set using the ADO XML schema, see “XML Result Set Example” on page 128.
- For a code sample, see “Search Code Samples” on page 129.

XML Query Examples

The code fragments in this section introduce XML formatted queries.

The following query retrieves the document id, file name and date of creation for documents added to the csdb object store having phrase 'central processing unit' in order and within range 0.

Listing 68: Example XML query

```
String searchStatement = "<request>"+
   "<objectstores>"+
   "<objectstore id=\"csdb^fn-test\"/>"+
   "</objectstores>"+
    "<querystatement>SELECT idmId, idmVerFileName, idmDateAdded FROM Document "+
   "CONTAINS 'central processing unit' inorder '0'</querystatement>"+
   "<options maxrecords=\"100\"/>"+
   "</request>";
String resultString = oSearch.executeXML(searchStatement);
```

The following query retrieves the document id, indexing status and group name for documents that have one of the following words or phrases: 1) having word 'tree'; 2) having phrase 'central unit' in any order within range 10 and having word 'stomp'.

Listing 69: Example XML query

```
String searchStatement = "<request>"+
   "<objectstores>"+
   "<objectstore id=\"csdb^fn-test\"/>"+
   "</objectstores>"+
    "<querystatement>SELECT idmId, idmVerCsiStatus, idmAddedByGroup FROM Document "+
    "CONTAINS 'tree' inorder '0' OR ( 'central unit' anyorder '10' AND 'stomp' "+
   "anyorder '0' )"+
   "</querystatement><options maxrecords=\"100\"/>"+
   "</request>";
String resultString = oSearch.executeXML(searchStatement);
```

The following query retrieves the document id, indexing status, and group name for documents that have one of the following words or phrases: 1) having phrase 'construction tree' in any order within range 50; 2) having phrase 'central unit' in any order within range 10 and having word 'stomp'; 3) having word 'house'

Listing 70: Example XML query

```
...
String searchStatement = "<request>"+
   "<objectstores>"+
   "<objectstore id=\"csdb^fn-test\"/>"+
   "</objectstores>"+
    "<querystatement>SELECT idmId, idmVerCsiStatus, idmAddedByGroup FROM Document "+
    "CONTAINS ('construction tree' anyorder '50' OR ( 'central unit' anyorder '10' "+
   "AND 'stomp' anyorder '0'  ) "+
   "OR 'house' inorder '0' "+
   "</querystatement><options maxrecords=\"100\"/>"+
   "</request>";
String resultString = oSearch.executeXML(searchStatement);
```

Combined Search

You can search documents, which have common property values and common content in a specific order (in order or any order) and in a specific range. You can express the search criteria in a Structured Query Language (SQL) statement contained in an XML formatted string. To perform a combined search, the search SQL should have expressions in WHERE and CONTAINS clauses.

- For information on SQL support by the CS Java Connector, see “Supported SQL Grammar” on page 112.
- For an overview of XML formatted queries for Combined Search, see “XML Query Examples” on page 121.
- For an example result set using the ADO XML schema, see “XML Result Set Example” on page 128.
- For a code sample, see “Search Code Samples” on page 129

XML Query Examples

The code fragments in this section introduce XML formatted queries.

The following query retrieves the document id, file name and date of creation for documents added to the csdb object store by Smith and having phrase 'central processing unit' in order and within range 0.

Listing 71: Example XML query

```
...
String searchStatement = "<request>" +
   "<objectstores>" +
   "<objectstore id=\"csdb^fn-test\"/>" +
   "</objectstores>" +
    "<querystatement>SELECT idmId, idmVerFileName, idmDateAdded FROM Document " +
   "WHERE idmAddedByUser = 'Smith' " +
   "CONTAINS 'central processing unit' inorder '0' " +
   "</querystatement>" +
   "<options maxrecords=\"100\"/>" +
   "</request>";
String resultString = oSearch.executeXML(searchStatement);
```

The next code fragment shows how to define an XML query string that retrieves the property values for document id, type, owner, protection status, and date added for documents that meet one of the following search criteria: 1) a document ID of 003676744 AND for all documents whose file name starts with the "a" character that have not been added by the Admin user; 2) a document ID of 003676270; 3) for all documents whose file name starts with the "a" character 4) having phrase 'construction tree' in any order within range 50; 5) having phrase 'central unit' in any order within range 10 or having word 'stomp'.

Listing 72: Example XML query

```
...
String searchStatement = "<request>" +
    "<objectstores>" +
    "<objectstore id=\"csdb^fn-test\"/>" +
     "</objectstores>" +
    "<querystatement>" +
    "SELECT idmId, idmDocType, idmDocOwner, idmDocProtected, " +
    "idmDateAdded FROM Document " +
     "WHERE (idmId = '003676744' and (idmVerFileName like 'a%' and idmAddedByUser " +
    "!= 'Admin') or idmId = '003676270') or idmVerFileName like 'a%' " +
     "CONTAINS 'construction tree' anyorder '50' AND ('central unit' anyorder '10' " +
    "OR 'stomp' anyorder '0' ) " +
    "</querystatement>" +
    "<options maxrecords=\"100\"/>" +
    "</request>";
String resultString = oSearch.executeXML(searchStatement);
```

Execute Search XML Schema

This XML schema supports input to a query and includes the following topics:

Namespace Definition

This schema uses the following namespace:

xmlns="http://filenet.com/namespaces/Panagon/apps/1.0"

Conventions

This document uses the following conventions:

Table 8: XML Schema Documentation Conventions

Occurrence Indicators	Description
none	Required and non-repeatable.
?	Optional and non-repeatable.
*	Optional and repeatable.
+	Required and repeatable.
,	Sequence.
|	Choice.
:	Identifies a default after a list of enumerated values.
(...)	Group.

Element List

This schema includes the following elements:

Element Descriptions

Table 9: XML Schema Element Descriptions

objectstore	
description	Specifies an Object Store.
attributes	id, name
uses	*none*
used by	objectstores
content	*none*

Table 9: XML Schema Element Descriptions (Continued)

objectstores	
description	Collection of objectstore elements. **Note:** This release of the CS Java Connector requires that you specify only one objectstore element.
attributes	mergeoption **Note:** “mergeoption” is ignored in this release of the CS Java Connector.
uses	objectstore+
used by	request
content	sequence
options	
description	Specifies query options.
attributes	cursorlocation, maxrecords **Note:** “cursorlocation” is ignored in this release of the CS Java Connector.
uses	*none*
used by	request
content	*none*
querystatement	
description	Specifies a query statement.
attributes	dataencoding **Note:** “dataencoding” is ignored in this release of the CS Java Connector.
uses	*none*
used by	request
content	*none*

Table 9: XML Schema Element Descriptions (Continued)

request	
description	Root element of the XML file.
attributes	*none*
uses	objectstores, querystatement, options
used by	*none*
content	sequence

Restriction List

This schema has no restrictions.

Restriction Descriptions

This schema has no restrictions.

Attribute List

This schema includes the following attributes:

- "id" on page 126
- "maxrecords" on page 126

Note: Other attributes listed in this section are ignored in this version of the CS Java Connector.

Attribute Descriptions

Table 10: XML Schema Attribute Descriptions

cursorlocation is ignored in this release of the CS Java Connector.	
description	Specifies whether the cursor is located on the server or client.
used by	options
type	NMTOKEN
value	server | client : server
use	default

Table 10: XML Schema Attribute Descriptions (Continued)

dataencoding is ignored in this release of the CS Java Connector.	
description	Specifies query statement.
used by	querystatement
type	string
value	
use	optional
id	
description	Object Store ID.
used by	objectstore
type	string
value	The object store ID must be a concatenation of the library name, the carat character, and host machine name. For example, for a Content Services library named Aslan on an server named Narnia, the value would be **Aslan^Narnia**.
use	required
mergeoption is ignored in this release of the CS Java Connector.	
description	Specifies the merging of result sets when two or more object stores are searched. **Note:** This release of the CS Java Connector searches only one object store at a time. This attribute is ignored.
used by	objectstores
type	string
value	union
use	default
maxrecords	
description	Specifies maximum number of records to return.
used by	options
type	string
value	positiveInteger : 5000
use	default **Note:** Do not enter a value of zero (0); otherwise an error will occur.

Table 10: XML Schema Attribute Descriptions (Continued)

name is ignored in this release of the CS Java Connector.	
description	Object Store name.
used by	objectstore
type	string
value	
use	optional

Schema Source

Listing 73: Execute search XML schema source

```
<?xml version="1.0" encoding="UTF-8"?>
<!--W3C Schema generated by XML Spy v3.5 NT (http://www.xmlspy.com)-->
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
elementFormDefault="qualified">
    <xsd:complexType name="objectstoreType">
        <xsd:attribute name="id" type="xsd:string" use="required"/>
    </xsd:complexType>
    <xsd:complexType name="objectstoresType">
        <xsd:sequence>
            <xsd:element name="objectstore" type="objectstoreType"
maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="optionsType">
        <xsd:attribute name="maxrecords" type="xsd:string" use="default" value="0"/>
    </xsd:complexType>
    <xsd:complexType name="querystatementType">
        <xsd:simpleContent>
            <xsd:restriction base="xsd:string"/>
        </xsd:simpleContent>
    </xsd:complexType>
    <xsd:element name="request">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="objectstores" type="objectstoresType"/>
                <xsd:element name="querystatement" type="querystatementType"/>
                <xsd:element name="options" type="optionsType" minOccurs="0"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

XML Result Set Example

Listing 74 shows an XML-formatted search query and the returned result set, formatted according to the Microsoft ADO XML schema. Note that the returned properties are enclosed in the `rs:data XML tag`.

- dsd mkk

Listing 74: Example XML query and result set

```
Search Query:
<request>
    <objectstores>
      <objectstore id="libsys^fncsjc01"/>
    </objectstores>
    <querystatement>
      SELECT idmId, idmDocOnlineLimit, idmDocType, idmVerCsiStatus,
      idmDateAdded, idmDocProtected FROM Document
      WHERE idmAddedByUser = 'Admin'
      CONTAINS 'central processing unit' inorder '0'
    </querystatement>
    <options maxrecords="100"/>
</request>
Result Set:
<xml xmlns:s='uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882' xmlns:dt='uuid:C2F41010-
65B3-11d1-A29F-00AA00C14882' xmlns:rs='urn:schemas-microsoft-com:rowset'
xmlns:z='#RowsetSchema'>
     <s:Schema id='RowsetSchema'>
    <s:ElementType name='row' content='eltOnly' rs:CommandTimeout='30'>
        <s:AttributeType name='idmId'>
       <s:datatype dt:type='string' dt:maxLength='9'/>
        </s:AttributeType>
        <s:AttributeType name='idmDocOnlineLimit'>
       <s:datatype dt:type='int' dt:maxLength='9'/>
        </s:AttributeType>
        <s:AttributeType name='idmDocType'>
       <s:datatype dt:type='string' dt:maxLength='32'/>
        </s:AttributeType>
        <s:AttributeType name='idmVerCsiStatus'>
       <s:datatype dt:type='int' dt:maxLength='2'/>
        </s:AttributeType>
        <s:AttributeType name='idmDateAdded'>
       <s:datatype dt:type='dateTime' dt:maxLength='32'/>
        </s:AttributeType>
        <s:AttributeType name='idmDocProtected'>
       <s:datatype dt:type='int' dt:maxLength='2'/>
        </s:AttributeType>
        <s:extends type='rs:rowbase'/>
    </s:ElementType>
     </s:Schema>
     <rs:data>
    <z:row idmId='003670042' idmDocOnlineLimit='5' idmDocType='General'
idmVerCsiStatus='1' idmDateAdded='2004-01-15T12:55:25,910' idmDocProtected='1'/>
    <z:row idmId='003670043' idmDocOnlineLimit='5' idmDocType='General'
idmVerCsiStatus='1' idmDateAdded='2004-01-15T12:55:39,643' idmDocProtected='1'/>
    <z:row idmId='003670044' idmDocOnlineLimit='5' idmDocType='General'
idmVerCsiStatus='1' idmDateAdded='2004-01-15T12:55:50,547' idmDocProtected='1'/>
    <z:row idmId='003670045' idmDocOnlineLimit='5' idmDocType='General'
idmVerCsiStatus='1' idmDateAdded='2004-01-15T12:56:07,250' idmDocProtected='1'/>
     </rs:data>
</xml>
```

Search Code Samples

This section provides code samples that demonstrate the use of the search object. For these samples to work in your development environment, you must modify hard-coded references to servers, object stores, documents, and other resources stored on a Content Services server. You must also create a Session object to log into an object store.

The following code samples are included in this section:

- Listing 75 reads in an XML-formatted query from a file (query.xml) and executes property based search on the object store.
- Listing 77 executes content based search on the object store.
- Listing 78 executes combined search on the object store.

Listing 75: performPropertyBasedSearch code sample

```
// Reads in XML-formatted query from a file (see Listing 76 ) and executes search.

public void performPropertyBasedSearch()
{
    String ObjectStoreId = "pin2ms^Pinta";
    ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
      sess);

   // Create search object
   Search sch  = com.filenet.Panagon.ObjectFactory.getSearch(sess);

   File f;
   FileReader in = null;
   String searchXML = "";
   try
   {
      f = new File(".", "query.xml");
      in = new FileReader(f);

      // Read the input from the file
      char [] buf = new char[4096];
      int len;
      while ((len = in.read(buf)) != -1)
      {
         String s = new String(buf, 0, len);
         searchXML += s;
      }
   }
   catch (java.io.IOException e)
   {
        e.printStackTrace();
   }
   finally
   {
        try
        {
           if (in != null)
             in.close();
        }
            catch(java.io.IOException e)
            {};
        }
   // Execute search
   String outputXML = sch.executeXML(searchXML);

   // Print results to console
   System.out.println(outputXML);
}
```

Listing 76: performPropertyBasedSearch XML query file

```
<!-- Read by program in Listing 75 -->
<?xml version="1.0" ?>
<request>
   <objectstores>
      <objectstore id="pin2ms^Pinta"/>
   </objectstores>
   <querystatement>
     SELECT idmDocOnlineLimit, idmId, idmDocType,
     idmDocOwner, idmDateAdded, idmDocProtected from Document WHERE
     idmAddedByUser = 'Smith' INFOLDER '988143960' ORDER BY idmId
   </querystatement>
   <options maxrecords="100"/>
</request>
```

Listing 77: performContentBasedSearch code sample

```
public void performContentBasedSearch()
{
    String ObjectStoreId = "csdb^fn-test";
    ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
         sess);

   // Create search object
   Search oSearch  = com.filenet.Panagon.ObjectFactory.getSearch(sess);


    String searchStatement = "<request>"+
   "<objectstores>"+
   "<objectstore id=\"csdb^fn-test\"/>"+
   "</objectstores>"+
    "<querystatement>SELECT idmId, idmDateAdded, idmAddedByUser FROM Document "+
    "CONTAINS ('construction tree' anyorder '50' OR ( 'central unit' anyorder '10' "+
   "AND 'stomp' anyorder '0'  ) "+
   "OR 'house' inorder '0' "+
   "</querystatement><options maxrecords=\"100\"/>"+
   "</request>";

    // Execute search
    String resultString = oSearch.executeXML(searchStatement);

    // Print results to console
   System.out.println(outputXML);
}
```

Listing 78: performCombinedSearch code sample

```
public void performCombinedSearch()
{
    String ObjectStoreId = "csdb^fn-test";
    ObjectStore os  = com.filenet.Panagon.ObjectFactory.getObjectStore(ObjectStoreId,
         sess);

   // Create search object
   Search oSearch  = com.filenet.Panagon.ObjectFactory.getSearch(sess);

    String searchStatement = "<request>" +
    "<objectstores>" +
    "<objectstore id=\"csdb^fn-test\"/>" +
    "</objectstores>" +
    "<querystatement>" +
    "SELECT idmId, idmDocType, idmDocOwner, idmDocProtected, " +
    "idmDateAdded FROM Document " +
    "WHERE (idmId = '003676744' and (idmVerFileName like 'a%' and idmAddedByUser"+
    "!= 'Admin') or idmId = '003676270') or idmVerFileName like 'a%' " +
    "CONTAINS 'construction tree' anyorder '50' AND ('central unit' anyorder '10'" +
    "OR 'stomp' anyorder '0' ) " +
    "</querystatement>" +
    "<options maxrecords=\"100\"/>" +
    "</request>";

    // Execute search
    String resultString = oSearch.executeXML(searchStatement);

    // Print results to console
   System.out.println(outputXML);
}
```

Index

Q

R

S

T

U

V

W

X