Getting Started with the Java API for Records Management (JARM)

November 2014

Getting Started with the Java API for Records Management

1 What is the Java API for Records Management?

The Java API for Records Management ("JARM") is a replacement for the original RM API and is intended for use in new custom Enterprise Records development solutions. It is currently used by the Enterprise Records IBM Content Navigator Plug-in component, the Enterprise Records REST Service API and by several Enterprise Records tools.

NOTE

Starting with IBM Enterprise Records v5.2.0.1, JARM now provides support for IBM Content Manager V8 type content repositories. See Section 6 "Support for IBM Content Manager V8" for more information.

Whereas the legacy RM API is based upon the older, now deprecated P8-3.x Java API, JARM is built on top of the newer P8 4.x/5.x Java API for Content Engine ("JACE"). Due to this relationship, a working knowledge of JACE is very helpful when learning and using JARM.

Some high level features of JARM include:

- Takes advantage of the transactional batching mechanism of JACE to provide more robust operations and improved performance over the original RM API.
- Does not require the extensive to/from XML overhead that exists in the legacy APIs.
- Better exception handling and reporting – does not hide underlying error information as is often the case in the legacy RM API.
- Can be used within a P8 Content Engine Event Handler.
- Makes use of Java generics and enum language syntax to provide improved type-safety enforcement.
- The general style of JARM follows that of JACE since many custom application scenarios will typically involve the use of both APIs.
- Unlike the former relationship between the original RM API and the P8-3.x Java API, JARM does not explicitly expose its underlying JACE implementation types. There are convenience utility methods within JARM to allow conversion between some higher level JACE and JARM types.
- Meets IBM Logging and Tracing requirements.

What is the relationship between JARM and the legacy RMAPI? The basic answer is that there is none – both APIs reside in differing Java package namespaces. So for example, an RMAPI ***com.filenet.rm.api.RecordCategory*** is not the same Java type as the JARM ***com.ibm.jarm.api.RecordCategory*** and the two differing types **cannot** be exchanged for each other. What is common to both APIs is that they both represent the same data structures found on an IBM FileNet P8 Content Manager environment. The main implication is that the P8 Content Manager object String identity values are common to both APIs. So one can convert between RMAPI and JARM object types through use of such object identity values:

- The GUID Id string which is supported by all P8 Content Manager objects.
- In some cases, the full path string of a container object -- "/Records Management/File Plan/TopRecordCategory1".
- In some cases, the object's symbolic name – object store "AccountingFPOS".

1.1 Installation

JARM is installed along with other Enterprise Records public APIs by choosing the "API" option of the Enterprise Records Installation process. JARM will be installed in the location "<install_root_dir>/IBM/EnterpriseRecords/API/JARM". Both the Jarm.jar file and other dependent JAR files are included within this directory.

1.2 Dependencies

JARM depends upon JACE for connection to and authentication with IBM FileNet Content Manager repositories. It can make use of either the WSI or EJB transport mechanism provided by JACE.

JARM requires a minimum Java SE 6 run time version.

The following table provides the list of the dependent JAR files required by JARM:

.jar File	Location	Description
Jarm.jar JarmResources.jar	…/EnterpriseRecords/API/JARM	(Required) Core JARM interfaces, implementation and globalized string resources.
ierLogTrace.jar	…/EnterpriseRecords/API/JARM	(Required) Enterprise Records logging and tracing support.
log4j-1.2.13.jar (or equivalent)	…/EnterpriseRecords/API/JARM	(Required) Log4j API used for logging and trace. (Note that the log4j.jar from JACE deployment can be used as well).
Jace.jar	…/EnterpriseRecords/ CommonFiles/CE_API/lib	(Required) IBM FileNet P8 Java API for Content Engine. (See additional information concerning JACE below).
pe.jar peResources.jar	…/EnterpriseRecords/ CommonFiles	(Optional) IBM FileNet P8 Process Engine Java API. Only needed if Initiate Disposition feature of JARM is used.
log4j.xml	…/EnterpriseRecords/API/JARM	(Optional) Sample JARM log4j configuration file to control JARM logging and tracing.

Depending upon the IBM FileNet P8 environment involved, additional JACE-related dependencies are also required. These JACE-related dependencies must match the version of CE/PE that you are currently using. Please see the IBM FileNet P8 Version 5.2 Knowledge Center (http://www.ibm.com/support/knowledgecenter/#!/SSNW2F_5.2.0/com.ibm.p8toc.doc/filenetcontentmanager_5.2.0.htm) for more information concerning the specific dependencies required by JACE. The following Information Center topic is especially helpful:

Developing IBM FileNet P8 applications > Content Engine Development > Content Engine Java and .NET Developer's Guide > Getting Started

For example, to support the JACE CEWS transport mode, the following JACE-related dependencies typically suffice:

.jar File	Location	Description
Jace.jar	…/EnterpriseRecords/ CommonFiles/CE_API/lib	(Required) IBM FileNet P8 Java API for Content Engine.
log4j.jar (or equivalent)	…/EnterpriseRecords/ CommonFiles/CE_API/lib	(Required) Log4J framework. Note that the log4j-1.2.13.jar version typically can be used.
stax-api.jar	…/EnterpriseRecords/ CommonFiles/CE_API/lib	Provides JACE CEWS transport support.
xlxpScanner.jar	…/EnterpriseRecords/ CommonFiles/CE_API/lib	Provides JACE CEWS transport support.
xlxpScannerUitls.jar	…/EnterpriseRecords/ CommonFiles/CE_API/lib	Provides JACE CEWS transport support.

1.3 Documentation

The full set of JARM Javadocs is located at the IBM Knowledge Center (http://www-01.ibm.com/support/knowledgecenter/SSNVVQ_5.2.0/com.ibm.p8.ier.dev.java.doc/jarm_api/javadocs/overview-summary.html.

Additional, development-related information is also included at this site.

2 Working with the New Java API for Records Management

The following sub-sections provide an introduction to the types exposed by JARM. The subsequent "How To..." sections provide code samples that demonstrate the use of these JARM types to perform common records manager tasks. When first introducing a JARM type its full package name is used. For reasons of brevity, the package prefix is then left off in subsequent references to the same type.

This document assumes that the reader is knowledgeable about general concepts of both FileNet Content Manager and Enterprise Records and is familiar with the FileNet Java API for Content Engine (JACE). Also, this document is intended to be used along with the JARM Javadocs and the reader is advised to consults the Javadocs for detailed descriptions of the various types and methods described herein.

2.1 Some General Concepts

Most JARM types representing file plan entities implement the ***com.ibm.jarm.api.core.BaseEntity*** interface. This ***BaseEntity*** interface provides access to various characteristics of an object that are common amongst the many JARM types.

One particularly important characteristic is that of an object's entity type. The entity type is a static, read-only, single valued integer property (symbolic name is "RMEntityType") defined for most every Enterprise Records class description. In JARM, various entity type constant values are defined by the ***com.ibm.jarm.api.constants.EntityType*** enumeration. The specific entity type for any given class instance can be acquired using its ***BaseEntity.getEntityType*** method.

The ***BaseEntity*** interface also provides other commonly helpful getter methods:

RMClassDescription getClassDescription	Returns the metadata class description of the object.
String getClassName	Returns the symbolic name of the class the object is based upon.
String getObjectIdentity	Returns the unique identifying string for the object (typically a GUID).
RMProperties getProperties	Returns the subset of properties currently available for the object.
RMPermissions getPermmissions	Returns the current security permissions for the object.
Repository getRepository	Returns the repository on which the object resides.

2.2 RMFactory

The ***com.ibm.jarm.api.core.RMFactory*** class consists of many static inner classes each of which provides type-specific access to existing repository objects and, in many cases, provides for the creation of new instances of these object types.

When using a specific inner class of ***RMFactory*** to access an existing repository object, there are typically both ***fetchInstance*** and ***getInstance*** methods available. The ***fetchInstance*** method involves performing a round-trip call to the appropriate repository to retrieve the current state of an object. Use the ***fetchInstance*** method when you need current state information about the object and/or wish to perform subsequent operations on the object.

On the other hand, the ***getInstance*** method does not perform any immediate round-trip call to the repository but, instead, simply returns an in-memory reference to the identified repository object. This reference can then be used, for example, when setting the value of an object-type property belonging to another object. In this manner an unnecessary repository round trip is avoided and application performance is improved.

For many (but not all) JARM types, creation of a new instance of that type is also performed using the RMFactory as part of the following creation recipe:

1. Use the appropriate ***RMFactory.<JARMType>.createInstance*** method to generate a new in-memory instance.

2. Use appropriate setters of the instance and/or define property values based upon the instance's ***RMProperties*** collection.
3. Optionally define custom security for the new instance using its ***RMPermission*** collection.
4. Finally call the instance's ***save*** method to persist the new instance to a repository.

The JARM record container types do not make use of the ***RMFactory*** mechanism for their new instance creation. Instead, an ***add*** method is called upon their target parent container to both create and persist a new instance. This is primarily due to the validation requirements for these types. The JARM ***Record*** is another type that does not use the ***RMFactory.createInstance*** mechanism but, instead, makes use of dedicated ***declare*** methods on various record container types.

2.3 DomainConnection and Authentication

The ***com.ibm.jarm.api.core.DomainConnection*** interface is used to establish a connection to a specific URL-addressable repository domain. It is also used in conjunction with the ***com.ibm.jarm.api.util.RMUserContext*** class to authenticate with the applicable domain.

If, as is very common, both JACE and JARM are used in the same application solution, then a best practice is to establish a valid "Java Authentication and Authorization Service (JAAS)" ***javax.security.auth.Subject*** instance for use by JACE. This can be accomplished using the JACE ***com.filenet.api.util.UserContext*** class or via a JEE Application Server's container-managed-authentication mechanism. This same JAAS ***Subject*** instance will then automatically be available for use by JARM also (in other words, a separate JARM authentication using its ***RMUserContext*** class will not be necessary). Note that a valid JAAS ***Subject*** instance must be established on each thread of execution that makes use of JACE and/or JARM.

A secondary feature of the ***RMUserContext*** class is to establish a ***java.util.Locale*** value for the current thread. This locale value is used by JARM to return the proper globalized exception-related message strings. As with the JAAS ***Subject***, if a locale has already been establish using the underlying JACE ***UserContext*** class then that value can be automatically used by JARM also. As a last resort, the current thread's default locale is used.

2.4 RMDomain and Repository

The ***com.ibm.jarm.api.core.RMDomain*** interface represents the entity that controls authentication and provides access to available repositories (both file plan and content). An ***RMDomain*** instance can be acquired using the ***RMFactory.RMDomain.fetchInstance*** method.

Within JARM, the term repository represents what is known as an object store in FileNet P8 nomenclature. ***com.ibm.jarm.api.core.Repository*** is the base interface for both of:

▪ ***com.ibm.jarm.api.core.FilePlanRepository***	The repository type that contains all Enterprise Records file plan hierarchies, records, schedules, etc.
▪ ***com.ibm.jarm.api.core.ContentRepository***	The repository type that contains original document content that can be declared as records.

A specific repository instance can also be acquired using either the ***RMFactory.Repository***, ***RMFactory.FilePlanRepository*** or ***RMFactory.ContentRepository*** inner class.

Each file plan repository contains a small collection of custom objects of the class "System Configuration" each of which represents some configuration setting that is global to the entire file plan repository such as the version number and type of data model that is installed. The JARM ***FilePlanRepository.getSystemConfiguration*** method provides access to these system configuration items in the form a key/value pair collection (***java.util.Map<String,SystemConfiguration>***). Each ***com.ibm.jarm.api.core.SystemConfiguration*** item exposes both a "Property Name" by which it is identified and a "Property Value" which represent the item's current configuration setting value. The "Property Name" value is also used as the ***String*** key in the returned ***Map***. The JARM ***SystemConfiguration*** interface includes ***String*** constant definitions for the possible "Property Name" identifiers.

2.5 Metadata

The following JARM interfaces provide access to existing metadata available on a repository:

- ***com.ibm.jarm.api.meta.RMClassDescription*** Describes the meta-characteristics of a class defined on the repository. The collection of all available class descriptions can be acquired using the ***Repository.fetchClassDescriptions*** method . A specific class description can also be retrieved using the ***RMFactory.RMClassDescription*** inner class.
- ***com.ibm.jarm.api.constants.RMClassName*** String constants representing the symbolic name of many commonly used Enterprise Records classes.
- ***com.ibm.jarm.api.meta.RMPropertyDescription*** Describes the meta-characteristics of a specific property that is defined for a given class description. This includes both system- and custom-defined property types. Sub-types of this interface (such as ***com.ibm.jarm.api.meta.RMPropertyDescriptionString***) provide additional, data-type specific information about a property.
- ***com.ibm.jarm.api.constants.RMPropertyName*** String constants representing the symbolic name of many commonly used Enterprise Records properties.
- ***com.ibm.jarm.api.meta.RMChoiceList*** Provides information about a repository-defined choice list. Individual members of a choice list are represented by instances of ***com.ibm.jarm.api.meta.RMChoiceItem***. A choice list can be associated with a specific property description in order to limit the set of allowed values for that property on a specific instance of the corresponding class. Choice lists are defined on a per-repository basis and can be retrieved from the applicable ***Repository*** instance. A specific choice list can also be acquired using the ***RMFactory.RMChoiceList*** inner class.
- ***com.ibm.jarm.api.meta.RMMarkingSet*** Provides information about a domain-level marking set. Individual members of a marking set are represented by instances of ***com.ibm.jarm.api.meta.RMMarkingItem***. A marking set is similar to a choice list with the added feature that each of its marking item values can dynamically affect the security assigned to an object whose class description includes a property description associated with the marking set. Marking sets are defined at the domain level and can be retrieved using the ***RMDomain*** interface. A specific marking set can also be retrieved using the ***RMFactory.RMMarkingSet*** inner class.

Note that JARM does NOT provide any mechanism for the creation of new nor modification to existing repository metadata such as property templates or class descriptions.

2.6 Properties and Property Filter

Each object instance of a repository class has a collection of properties associated with it. The particular collection of properties for any individual object is defined by the repository class upon which it is based (***RMClassDescription***) and the set of property descriptions (***RMPropertyDescription***) defined for that class. When a given repository object is fetched by a JARM client, the actual collection of returned property values for that object is represented by the ***com.ibm.jarm.api.property.RMProperties*** interface. This collection interface contains members of the ***com.ibm.jarm.api.property.RMProperty*** interface, each of which represents the value of an individual property. The JARM ***RMProperties*** collection for a given client-side object instance acts as a client-side cache of the corresponding repository object's property values. It is important to note that JARM does not provide any automatic fetch of a missing property value from a given JARM instance's ***RMProperties*** collection, instead a "property not found" exception will result. If you need to specifically fetch additional property values from the repository into a specific JARM instance's ***RMProperties*** collection, then use one of the ***BaseEntity.refresh*** methods to do so.

Some class descriptions define a large number of property descriptions. For performance reasons, when retrieving objects of such a class it is a best practice to limit the number of property values returned at any given time to those that are required for the task at hand. Two interfaces, ***com.ibm.jarm.ap.property.RMPropertyFilter*** and ***com.ibm.jarm.api.property.RMFilterElement***, are used to control the number of properties and the detail about each that are returned. These two interfaces are simply wrappers of the underlying JACE classes, ***com.filenet.api.property.PropertyFilter*** and ***com.filenet.api.property.FilterElement***, respectively. The following documentation reference provides detailed information about the use of the JACE ***PropertyFilter*** mechanism that is also applicable to JARM's ***RMPropertyFilter***:

> Please see the IBM FileNet P8 Version 5.2 Knowledge Center (http://www.ibm.com/support/knowledgecenter/#!/SSNW2F_5.2.0/com.ibm.p8toc.doc/filenetcontentmanager_5.2.0.htm) under the topic *Developing IBM FileNet P8 applications > Content Engine Development > Content Engine Java and .NET Developer's Guide > Properties > Concepts > Property Filter Concepts*

When working with a specific Enterprise Record object type, JARM often requires that a minimum working set of properties currently exist in that object's ***RMProperties*** collection. During object retrievals, JARM will automatically embellish any user-defined ***RMPropertyFilter*** to ensure that the proper minimum working set of properties is returned. As a convenience, the static constant ***RMPropertyFilter.MinumumPropertySet*** value can be used as the value of any ***RMPropertyFilter*** method input parameter to indicate that this minimum working set of properties be returned.

As is the case for the underlying JACE ***PropertyFilter*** mechanism, a ***null*** value can be supplied for any ***RMPropertyFilter*** method input parameter to indicate that ALL scalar-type properties be returned and that object references be returned for ALL object-type properties. Use of a ***null*** parameter value in this manner does help avoid possible "missing property in cache" exceptions but at the potential cost in performance for those object types with very large property collections; so be careful when using this feature.

The in-memory collection of properties for an object is acquired using the object's ***BaseEntity.getProperties*** method. Access to an individual property value collection member, identified by the property's symbolic name, is performed using the applicable data-type-specific ***getXXXValue*** methods (for example, ***getDateTimeValue***).

Based upon property symbolic name, an ***RMProperties*** collection can be updated using either one of its generic ***add*** methods or the appropriate data-type-specific ***putXXXValue*** method (for example, ***putIntegerListValue***). The property value of an existing member of the ***RMProperties*** collection (based upon symbolic name) will be updated with the new value otherwise a new entry will be added to the collection. In order to permanently persist any such object property value changes to the object's host repository, the object's ***save*** method must be invoked.

In some cases when creating a new instance of some particular JARM type, it is necessary to supply a pre-populated ***RMProperties*** collection. The ***RMFactory.RMProperties.createInstance*** method can be used to generate a new, empty ***RMProperties*** collection for use in such a situation.

2.7 Object Collections

Many JARM methods return collections of specific types. These collections can be divided roughly into two variations:

1. Collections that contain a relatively small number of members. For example, the collection of holds that are currently applied to a specific record.
2. Collections that can potentially contain a very large number of members. For example, all records residing under some container in a file plan.

Collections of the limited-in-size first variation, are represented using one of the standard, type-safe Java collection mechanisms such as ***java.util.List<T>***, ***java.util.Set<T>*** or ***java.util.Map<T>***.

Those of the second variation are represented by the JARM collection interface ***com.ibm.jarm.api.collection.PageableSet<T>***. This ***PageableSet<T>*** interface allows for type-safe, paged access to a collection's content using the interface ***com.ibm.jarm.api.collection.RMPageIterator<T>***. Alternatively, a standard ***java.util.Iterator<T>*** access mechanism can also be used with the caveat that for very large collections, a timeout and/or loss-of-connection errors can result. Note also that ***PageableSet<T>*** implements the ***java.lang.Iterable<T>*** interface which allows use of the newer Java ***for*** loop syntax:

```
PageableSet<RecordCategory> resultSet = ...;
for (RecordCategory recCat : resultSet)
{
  // Use recCat ...
}
```

There is also the specialized version of ***RMPageIterator<T>*** known as ***com.ibm.jarm.api.collection.CBRPageIterator<T>*** which is used to represent results of Content Based Retrieval (CBR) type search operations.

2.8 Object Security

The security assigned to each repository object is represented by its collection of "Access Control Entry (ACE)" items which is referred to as an "Access Control List (ACL)". In JARM, each individual ACE is represented by the interface ***com.ibm.jarm.api.security.RMPermission*** and the ACL is represented as a ***java.util.List<RMPermission>***. Each ***RMPermission*** instance describes the principal involved (name and type), the principal's access rights, the source of the particular ACE and its inheritability to related objects.

An object's permissions can be acquired using the object's ***BaseEntity.getPermissions*** method and can be updated using its ***BaseEntity.setPermissions(RMPermissions acl)*** method. Note that the set method always works with the entire ACL of the object and requires a subsequent ***save*** method call on the object in order to persist the modified permissions collection.

In some cases when creating a new instance of some particular JARM type, it may be necessary to supply a pre-populated ***RMPermissions*** collection. The ***RMFactory.RMPermissions.createInstance*** method can be used to generate a new, empty ***RMPermissions*** collection for use in such a situation.

2.9 Search

Standard search operations are performed using the ***com.ibm.jarm.api.query.RMSearch*** class. This class allows a FileNet Content Manager SQL statement to be used to query one or more repositories. It supports several forms of query operation:

- fetchObjects — Used to query for a homogeneous result set of JARM object types. The returned value is an instance of ***PageableSet<T>***. Since each returned item is of the same repository class (or sub-class), the P8 SQL statement's SELECT clause can only include properties applicable to that class (or subclass).
- fetchRows — Used to query for a result set of multiple ***RMProperties*** collections. The actual returned value is an instance of ***PageableSet<com.ibm.jarm.api.query.ResultRow>*** where each ***ResultRow*** member provides access to a single ***RMProperties*** collection. Because this RMProperties collection is not necessarily tied to a specific repository class, it can contain members associated with different classes of objects that might result from the execution of a query involving one or more JOIN operations.
- contentBasedRetrieval — Used to perform a CBR-type search as defined by the input parameter of type ***com.ibm.jarm.api.query.RMContentSearchDefinition***. This type of query returns its results as a ***PageableSet<CBRResult>***.

2.10 JACE / JARM Conversion Utility

Since a custom application often involves the use of both JACE and JARM, the JARM utility class ***com.ibm.jarm.api.util.P8CE_Convert*** provides static methods that convert between analogous types in the two APIs:

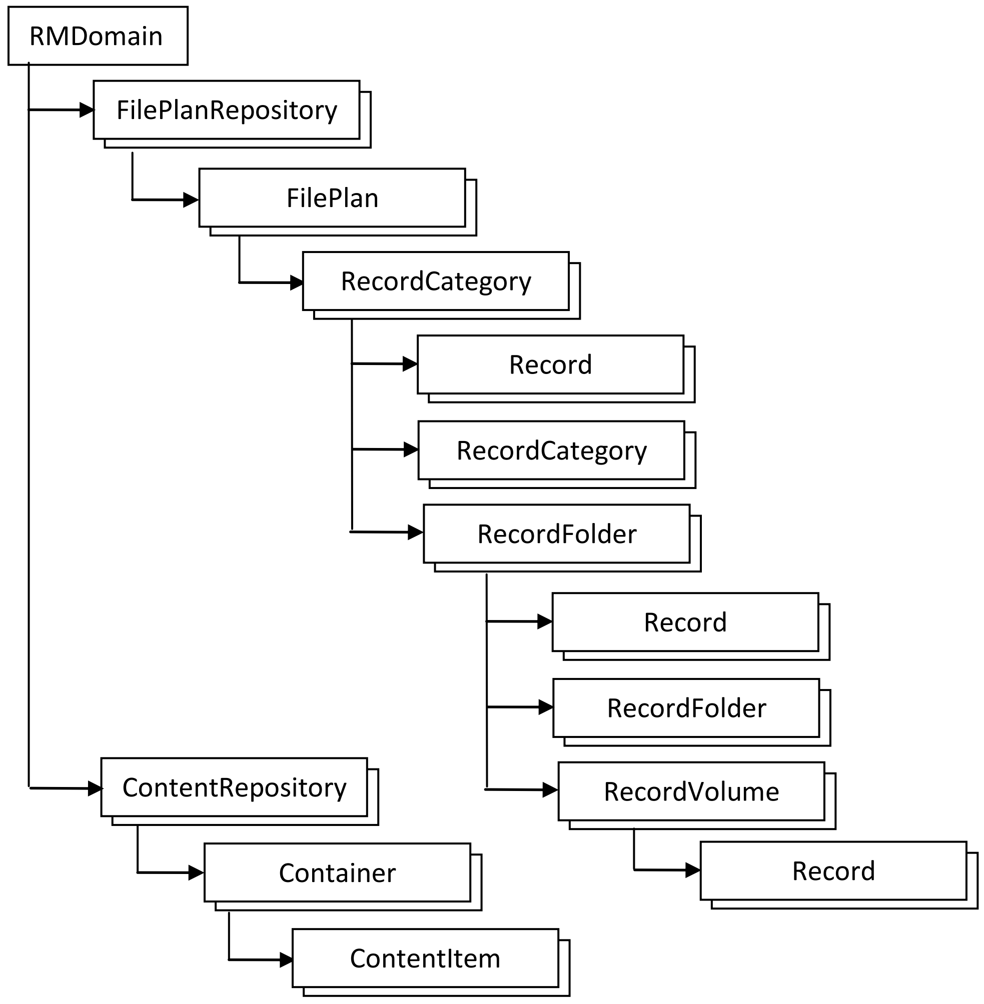
JACE	JARM
com.filenet.api.core.Connection	***com.ibm.jarm.api.core.DomainConnection***
com.filenet.api.core.Domain	***com.ibm.jarm.api.core.RMDomain***
com.filenet.api.core.ObjectStore	***com.ibm.jarm.api.core.Repository***
com.filenet.api.core.Folder	***com.ibm.jarm.api.core.Container***
com.filenet.api.core.Document	***com.ibm.jarm.api.core.ContentItem***

When converting from a JACE ***ObjectStore*** or ***Folder*** type to JARM, an attempt is made to generate the appropriate JARM subclass based upon the actual characteristics of the entity. For example, if a JACE Folder instance actually represents an Enterprise Records record category, then the conversion from JACE to JARM will result in a ***com.ibm.jarm.api.core.RecordCategory*** instance.

For conversion between types not supported by this P8CE_Convert utility class, the typical solution is to use the entity's common Id property value and the appropriate API factory class (***com.filenet.api.core.Factory*** for JACE and ***com.ibm.jarm.core.RMFactory*** for JARM) to re-fetch the entity in the other API's namespace.

2.11 *File Plan Container Hierarchy*

The following figure provides the high-level relationships between the various JARM container types:



The ***RMDomain*** interface provides access to all of the currently available ***FilePlanRepository*** and ***ContentRepository*** instances. The ***FilePlanRepository*** provides access to all of its available ***com.ibm.jarm.api.core.FilePlan*** instances.

For the ***FilePlan*** and its sub-container types, control over which types of child entities can be directly contained is maintained by the previously mentioned concept of ***EntityType*** available via the ***BaseEntity*** interface. Each file plan container type supports the method ***getAllowedContaineeTypes*** which returns an array of ***EntityType*** enumeration values, each of which defines a type of child entity that the container is allowed to contain at this point in time.

Where applicable, a given container type implements one or more of the following interfaces that provide access to its containees:

- ***com.ibm.jarm.api.core.RecordCategoryContainer*** Provides ability to retrieve and add new directly contained child record categories.
- ***com.ibm.jarm.api.core.RecordFolderContainer*** Provides ability to retrieve and add new directly contained child record folders.

- ***com.ibm.jarm.api.core.RecordVolumeContainer*** Provides ability to retrieve and add new directly contained child record volumes.
- ***com.ibm.jarm.api.core.RecordContainer*** Provides ability to retrieve and declare new directly contained records

Each container type supports the ***getParent*** method to allow navigating upwards through a file plan hierarchy. The Record interface provides the ***getContainedBy*** method used to determine where the record is currently filed.

All container types extend the base ***com.ibm.jarm.api.core.Container*** interface.

A specific container can also be retrieved using the appropriate ***RMFactory*** inner class: ***RMFactory.Container***, ***RMFactory.FilePlan***, ***RMFactory.RecordCategory***, ***RMFactory.RecordFolder*** or ***RMFactory.RecordVolume***. In these situations the container can be identified by either its unique identifying string (e.g. GUID) or by its full path name (including the "/Records Management/" prefix).

2.12 Records and Associated Content

In JARM, the ***com.ibm.jarm.api.core.Record*** interface is used represent both electronic and physical record types.

A physical record typically represents content that is external to any repository. Physical record classes have two properties that reference an object type of ***com.ibm.jarm.api.core.Location***, which in turn, describes the external physical location of the record's associated content (for example, "filing cabinet 42 in room 456 of building 123").

Electronic records are associated with one or more ***com.ibm.jarm.api.core.ContentItem*** instances, each of which represents a document or document version found on a ***ContentRepository***. Once a ***ContentItem*** has been "declared as a record" (meaning a ***Record*** instance has been created for it on a ***FilePlanRepository***), two important characteristics of the ***ContentItem*** are affected:

1. The original security assigned to the ***ContentItem*** is typically replaced by or combined with the effective security belonging the ***Record***. The effective security will be determined by the type of security proxy defined between the record and content classes.
2. The ***ContentItem*** can no longer be directly deleted – it can only be deleted as part of the process that deletes its associated ***Record***.

The ***Record.getAssociatedContentItems*** method can be used to retrieve all such ***ContentItem*** instances that are under control of a specific record. Likewise, the ***ContentItem.getAssociatedRecord*** method can be used to access a declared content item's controlling record; this method will return ***null*** if the content item is not currently declared as a record.

A currently declared content item can be "undeclared" by use of the ***undeclare*** method belonging to its associated ***Record*** object. This method will disassociated the record from the content item and will permanently delete the record object from its file plan repository. Any of the following situations, however, will prevent an undeclare operation:

- The record is currently involved in a disposition workflow process.
- The record is currently on hold (either directly or via a parent container).
- The record is associated with federated content.

Both a specific record and content item can be directly retrieved by using their appropriate factory class, ***RMFactory.Record*** or ***RMFactory.ContentItem***, respectively.

The actual set of binary contents represented by a ***ContentItem*** can be determined using the ***ContentItem.getContentElements*** method which returns a collection of ***com.ibm.jarm.api.core.RMContentElement***. ***RMContentElement*** provides access to any available content binary data.

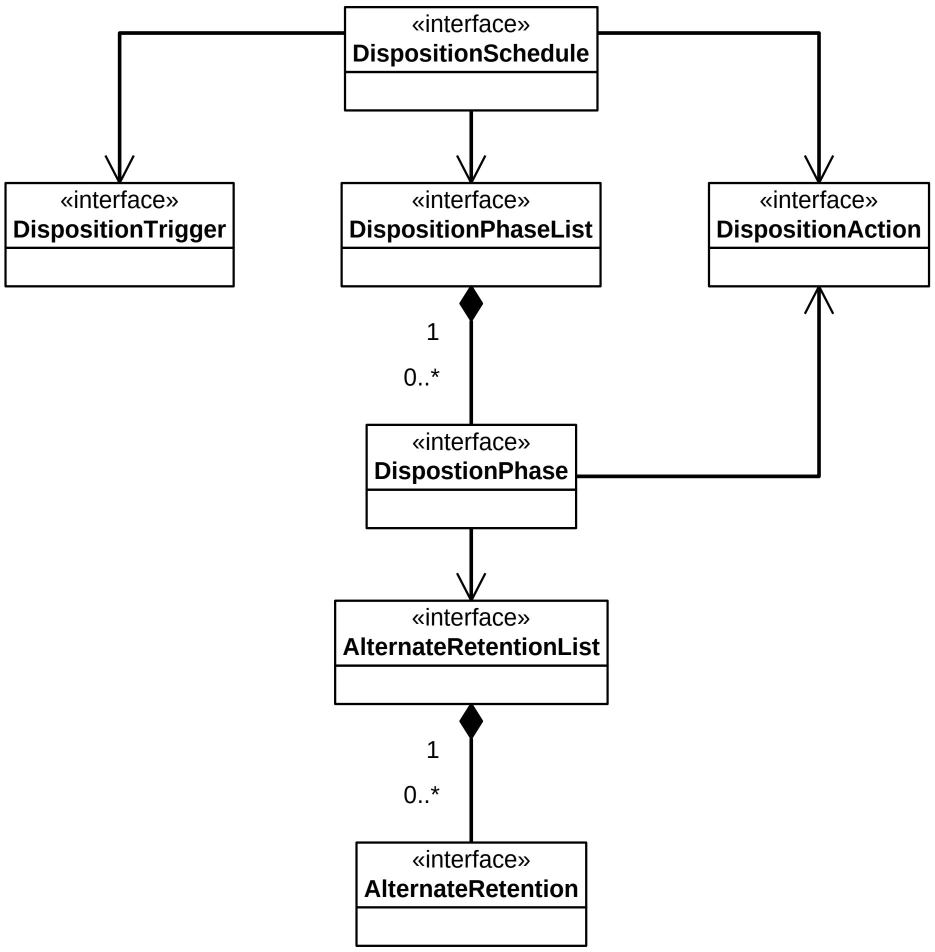
2.13 Naming Patterns

Several interfaces and a utility class exist within JARM to provide support to client applications that wish to provide automatic naming of new container and record instances:

▪ ***com.ibm.jarm.api.core.NamingPattern***	Assigned to a FilePlan in order to provide per-container-hierarchy level naming pattern rules for various file plan container types.
▪ ***com.ibm.jarm.api.core.NamingPatternLevel***	Provides specific naming pattern instructions for a particular level of a file plan container hierarchy.
▪ ***com.ibm.jarm.api.util.NamingUtils***	Provides static utility methods for generating and validating various container and record names based upon the appropriate ***NamingPattern*** and ***NamingPatternLevel*** instances involved.

2.14 Disposition Artifacts

The following figure provides relationships between the various disposition-related JARM types (all of which reside in the ***com.ibm.jarm.api.core*** package):



The ***FilePlanRepository*** interface provides access methods to retrieve the current set of available ***DispositionSchedule***, ***DispositionTrigger*** and ***DispositionAction*** instances. Specific instances of these three types can also be acquired using their respective factory inner classes, ***RMFactory.DispositionSchedule***, ***RMFactory.DispositionTrigger*** and ***RMFactory.DispositionAction***. These same factory inner classes each support a ***createInstance*** method that is used to create a new instance of each type (followed by a mandatory ***save*** call to persist to the appropriate file plan repository).

Entities that support assignment of a disposition schedule implement the ***DispositionAllocatable*** interface. When a disposition schedule is assigned to an eligible container, control over how the assignment affects its descendent containers is defined by the enumeration ***com.ibm.jarm.api.constants.SchedulePropagation***.

For a given disposition schedule, its corresponding ***DispositionPhaseList*** collection allows management of the zero or more ***DispositionPhase*** instances defined for it. Likewise, for each ***DispositionPhase***, its ***AlternateRetentionList*** collection allows management of zero or more ***AlternateRetention*** instances.

For legacy purposes, the ***com.ibm.jarm.api.core.RecordType*** interface provides the ability to assign a schedule directly to an individual record, but its use is **highly discouraged** due to the resulting detrimental effect on disposition processing performance.

2.15 Defensibly Disposable Containers

Starting with the Enterprise Records 5.1.2 release, the ability to define what is sometimes referred to as a "Basic Schedule" on a record container is now supported. Within JARM, such a container is referred to as a "defensibly disposable container" and the following interfaces provide support for such:

▪ ***com.ibm.jarm.api.core.DefensiblyDisposableContainerParent***	Provides indication of whether an existing file plan container is capable of containing a defensibly disposable child container and the mechanism for the creation of such a child container.
▪ ***com.ibm.jarm.api.core.DefensiblyDisposable***	Provides indication of the eligibility of an existing record container for conversion into a defensibly disposable container and the conversion mechanism itself. Also provides get/set access to the primary characteristics that define a defensibly disposable container.

Only record category type containers are potentially eligible to act as defensibly disposable containers.

Note that the ***FilePlanRepository.supportsDefensibleDisposal*** method should initially be used to verify that a given file plan repository does indeed provide data model support for defensibly disposable containers.

2.16 Externally Managed Entities

Some record container and disposition-related objects may be under control of an external application (for example the IBM PSS Atlas product line) and, as such, should normally not be modified nor deleted by any other Enterprise Records-related application. The following concepts can be used to help detect such externally managed objects:

▪ ***FilePlanRepository.supportsExternalManagement***	Reports whether or not data model support for external management has been installed on a given file plan repository.
▪ ***RMPropertyName.ExternallyManagedBy***	The symbolic name of a string property that contains a non-null, non-empty value indicating that the corresponding object is under external management. (The ***com.ibm.jarm.api.constants.ExternalManagementAgent***

.PSS_Atlas string constant represents the applicable value for IBM PSS Atlas).

2.17 Holds

A ***com.ibm.jarm.api.core.Hold*** type represents a legal hold directive that can be "placed" upon a record or a record container entity for the purpose of temporarily preventing any final disposition of the entity or its contents. The collection of all available ***Hold*** instances can be acquired using the ***FilePlanRepository*** interface.

A specific Hold instance can be retrieved using the ***RMFactory.Hold*** inner class. The ***RMFactory.Hold*** inner class also provides a ***createInstance*** method to define a new ***Hold*** instance. Once all necessary properties of a new ***Hold*** instance have been defined, it's ***save*** method is then used to persist the new instance onto its host file plan repository.

The JARM record container types (and Record itself) support the ***com.ibm.jarm.api.Holdable*** interface that provides mechanisms for the placement and subsequent removal of a hold. The ***Holdable*** interface also provides utility methods for determining whether any parent or child of a given entity has any current holds placed upon it.

The ***Hold*** interface also provides methods for determining which containers and/or records the specific hold instance is currently placed upon.

2.18 Custom Object types

In addition to any customer-defined Custom Object types, Enterprise Records implements many of its business logic types as sub-classes of Custom Object. In JARM (unlike the legacy RMAPI), many of these Custom Object sub-class types are elevated to first class entities (all found in ***com.ibm.jarm.api.core*** package):

- AlternateRetention
- DispositionAction*
- DispositionPhase
- DispositionSchedule*
- DispositionTrigger*
- Hold*
- Location*
- NamingPattern*, NamingPatternLevel, NamingPatternSequence*
- RecordType*
- SystemConfiguration

(*) Indicates that this particular type is also supported by a corresponding RMFactory inner class for both retrieval and creation purposes.

There is also the base JARM interface, ***com.ibm.jarm.api.core.RMCustomObject***, which each of these more specific types extends from. ***RMCustomObject*** can also be used to represent any other custom object sub-class item.

2.19 Link types

The various link types that subclass the "RMLink" class are all represented by the JARM interface ***com.ibm.jarm.api.core.RMLink***. The ***RMFactory.RMLink*** static inner class can be used to retrieve a known existing link and to create a new instance of such.

2.20 Deletion and Destruction of Objects

Depending upon the type of object involved and whether or not it is under control of a disposition schedule, deletion and destruction may or may not both apply and may have slightly different meanings.

For non-dispositionable types (i.e., typically those types other than record containers and records), a delete method is available that usually just causes the object in question to be permanently deleted from its host repository. In some cases, due to data model constraints such as the object in question is still being referenced by another object, deletion may be prevented at the repository level.

For dispositionable types, their ultimate demise usually comes about due to a call to their ***destroy*** method as a result of the disposition process. This destroy mechanism typically involves disposition-related validation processing to check for possible changes in either an associated schedule or the object itself that may disqualify it from the destroy process.

Alternatively, the ***delete*** method for either a record container or record can be used to dispose of the corresponding object. This delete method takes an input parameter of the enumerated type ***com.ibm.jarm.api.constants.DeleteMode*** that determines whether or not a hard versus logical deletion process occurs. A hard deletion involves permanently removing the object from the file plan repository. The purpose of the logical deletion is to support the Enterprise Record concept of "retain metadata". A logical deletion involves not actually deleting the container or record repository object, but instead, simply sets its ***RMPropertyName.IsDeleted*** property to ***true***. Standard Enterprise Record applications normally use this boolean property to filter out any such logically deleted objects to prevent their display during normal Enterprise Records operations.

Obviously, lack of applicable security rights and/or presence of an active Hold on an object will prevent either type of deletion from occurring.

Specific to an electronic record, both the hard and logical deletion process types result in the hard deletion of any of the record's associated content items from their content repository.

2.21 Bulk Operations

The class ***com.ibm.jarm.api.core.BulkOperation*** provides convenient static methods supporting the common application use case whereby multiple items are selected in a user interface and a common operation is to be performed upon each selection. In general, each of the ***BulkOperation*** methods returns a ***java.util.List<BulkItemResult>*** collection. Each returned ***com.ibm.jarm.api.core.BulkItemResult*** reports on the success or failure of the operation for each originally specified item. Depending upon the specific operation, each involved entity is specified either by just its GUID identifier or via an instance of ***BulkOperation.EntityDescription*** (which simply combines a GUID identifier with an entity type).

The available bulk operations include:

- Activate / inactivate multiple containers
- Close / reopen multiple containers
- Delete multiple entities
- Move / copy / file / undeclare multiple records
- Place / remove holds on multiple entities
- Initiate disposition on multiple entities

2.22 Audit Support

JARM exposes two interfaces that provide support for accessing existing FileNet P8 audit events belonging to Enterprise Record objects:

▪ ***com.ibm.jarm.api.core.AuditableEntity***	An interface extended by those types that are eligible for auditing. The ***getAuditedEvents*** method returns the collection of all "object change event" types, including the Enterprise Record's "RM Audit" subclass, that exist for a specific repository object.
▪ ***com.ibm.jarm.api.core.AuditEvent***	Represents a single audit event instance.

2.23 Exception Handling

JARM exception handling is based upon the single exception class ***com.ibm.jarm.api.exception.RMRuntimeException*** which derives from the unchecked ***java.lang.RuntimeException***. ***RMRuntimeException*** provides an enumerated ***com.ibm.jarm.api.exception.RMErrorCode*** to help distinguish between various error types.

Globalized exception message support is provided using the com.ibm.jarm.api.exception.MessageInfo interface which exposes the following types of globalized strings:

▪ Formatted Message	Provides a globalized message string formatted with any runtime replacement parameters.
▪ Explanation	Provides additional detail, if any, concerning the exception occurrence.
▪ Action	Provides a suggest action to take, if any, to help alleviate the error situation.

The ***RMRuntimeException*** also provides access to the ***com.ibm.jarm.api.exception.RMErrorStack*** and ***com.ibm.jarm.api.exception.RMErrorRecord*** interfaces that are used to report on any internal chained exceptions.

2.24 Users and Groups

Available users and groups are exposed using the JARM ***RMDomain*** interface methods ***findUsers*** and ***findGroups***, each of which allows for various filtering criteria to be used as part of the retrieval process. The resulting entities are represented using the JARM interfaces ***com.ibm.jarm.api.security.RMUser*** and ***com.ibm.jarm.api.security.RMGroup***, both of which are based upon the common base interface ***com.ibm.jarm.api.security.RMPrincipal***.

The currently authenticated user can be determined using the ***RMDomain.fetchCurrentUser*** method.

2.25 Legacy XML Export

The various record container and record types provide support for XML export of their state based upon the XML schema defined for the legacy P8 3.x Java API. The method used to perform the export process is ***exportUsingP8_XML***.

2.26 Workflow Definition

JARM represents a workflow definition using the ***com.ibm.jarm.api.core.RMWorkflowDefinition*** interface. A collection of available Enterprise Records workflow definitions can be obtained using the ***FilePlanRepository.getWorkflowDefinitions*** method. A single workflow definition can be retrieved using the ***RMFactory.WorkflowDefinition*** inner class.

Note that JARM does not provide support for the creation nor modification of any workflow definition instance.

3 How to…

The following subsections provide JARM example code snippets for common Enterprise Records application programming tasks. In order to keep these code example snippets short and directed towards the particular task at hand, necessary ***#import*** statements and exception handling are not typically shown. Also, other than for the first examples, it is assumed that a domain connection has been made and authentication has been performed.

3.1 Connect to and authenticate with a domain

The first example demonstrates how to authenticate using only JARM.

```
// Best practice is to eventually un-establish any JAAS Subject
// once communication with the domain is no longer needed.
boolean establishedJAASSubject = false;
try
{
  // Example WSI transport URL for a FileNet P8 domain.
  String domainURL = "http://<server>:<port>/wsi/FNCEWS40MTOM";
  String username  = ...;
  String password  = ...;

  // ---- Establishing a JARM DomainConnection ----
  // Only one type of domain is currently supported by JARM.
  DomainType domainType = DomainType.P8_CE;
  // Ancillary connection information that may be needed in
  // future JARM support for alternative domain types will
  // make use of a Map<String, Object> structure. However,
  // this Map is not currently required for a P8_CE domain type.
  java.util.Map<String, Object> connectionInfo = null;
  // Use the RMFactory to create the new DomainConnection.
  DomainConnection jarmConnection =
    RMFactory.DomainConnection.createInstance(domainType, domainURL, connectionInfo);

  // ---- Authenticate using the JARM RMUserContext ----
  // Use helper method to create a JAAS Subject.
  //  The following JAAS stanza identifier can be left null in
  //  order to allow the underlying JACE mechanism to automatically
  //  determine the correct stanza to use based upon the domain URL.
  String jaasStanza = RMUserContext.P8_STANZA_WSI;
  javax.security.auth.Subject jaasSubject =
    RMUserContext.createSubject(jarmConnection, username, password, jaasStanza);
  // Get the user context for the current thread.
  RMUserContext jarmContext = RMUserContext.get();
  // Establish the JAAS Subject on this thread.
  jarmContext.setSubject(jaasSubject);
  establishedJAASSubject = true;

  // ---- (Optionally) Define the user context locale ----
  jarmContext.setLocale(java.util.Locale.US);

  // ---- Perform custom application work ----
  // ...
}
finally
{
  // Un-establish any previously set JAAS Subject.
```

```
    if ( establishedJAASSubject )
    {
      RMUserContext.get().setSubject(null);
    }
  }
```

The following alternative authentication example based upon JACE is probably the more useful of the two since it is likely that both JACE and JARM will be used in any custom application. Establishment of a JAAS Subject using JACE makes it automatically available to JARM also.

```
  // Best practice is to always un-establish the JAAS
  //  Subject from the current thread once communication
  //  with the domain in no longer needed.
  boolean establishedSubject = false;
  try
  {
    // Example WSI transport URL for a FileNet P8 domain.
    String domainURL = "http://<server>:<port>/wsi/FNCEWS40MTOM";
    String username  = ...;
    String password  = ...;

    // ---- Connect and Authenticate using JACE ----
    // Acquire a JACE domain connection
    Connection jaceConnection = Factory.Connection.getConnection(domainURL);
    // Authenticate using JACE UserContext class.
    //  The following JAAS stanza name can optionally be left
    //  null to allow JACE to dynamically determine the correct
    //  value based upon the URL format.
    String jaasStanza = "FileNetP8WSI";
    javax.security.auth.Subject jaasSubject =
      UserContext.createSubject(jaceConnection, username, password, jaasStanza);
    // Acquire JACE UserContext for the current thread.
    UserContext jaceUC = UserContext.get();
    jaceUC.pushSubject(jaasSubject);
    establishedSubject = true;

    // ---- (Optionally) Define user context locale ----
    jaceUC.setLocale(java.util.Locale.US);

    // ---- Perform custom application work ----
    // ...
  }
  finally
  {
    if ( establishedSubject )
    { // Un-establish the Subject from the current thread.
      UserContext.get().popSubject();
    }
  }
```

The following example demonstrates that once authentication has succeeded, a JARM DomainConnection can then be used to acquire a RMDomain instance:

```
    DomainConnection jarmConnection = ...;

    // ---- Acquire the JARM RMDomain ----
    String domainIdent = null;  // For P8, null indicates 'local' domain.
    RMPropertyFilter filter = RMPropertyFilter.MinimumPropertySet;
    RMDomain jarmDomain =
      RMFactory.RMDomain.fetchInstance(jarmConnection, domainIdent, filter);
    // Use this domain object to verify the current JAAS
    //  Subject credentials.
    RMUser jarmCurrentUser = jarmDomain.fetchCurrentUser();
    System.out.printf("Domain name: %s, current user: %s%n",
           jarmDomain.getName(), jarmCurrentUser.getDisplayName());
```

3.2 Discover available repositories and retrieve a specific repository

This example demonstrates how to discover available file plan and content repositories for a given ***RMDomain*** and how to retrieve a specific repository based upon its symbolic name (its identifying GUID string can also be used).

```
  RMDomain jarmDomain = ...

  // ---- Discover all available file plan repositories ----
  List<FilePlanRepository> fpRepositories =
    jarmDomain.fetchFilePlanRepositories(RMPropertyFilter.MinimumPropertySet);
  for (FilePlanRepository fpRepository : fpRepositories)
  {
    System.out.printf("FPRepository SymbolicName: %s, Id: %s%n",
      fpRepository.getSymbolicName(), fpRepository.getObjectIdentity());
  }

  // ---- Discover all available content repositories ----
  boolean excludeCombined = true;
  List<ContentRepository> contentRepositories =
    jarmDomain.fetchContentRepositories(excludeCombined,
                 RMPropertyFilter.MinimumPropertySet);
  for (ContentRepository contentRepository : contentRepositories)
  {
    System.out.printf("ContentRepository SymbolicName: %s, Id: %s%n",
      contentRepository.getSymbolicName(), contentRepository.getObjectIdentity());
  }

  // ---- Retrieve a specific file plan repository by its symbolic name ----
  String fpReposSymbolicName   = "BaseFPOS";
  RMPropertyFilter filter = null; // Being lazy - request all properties.
  FilePlanRepository fpRepos =
    RMFactory.FilePlanRepository.fetchInstance(jarmDomain, fpReposSymbolicName, filter);
  System.out.printf("Single FPRepository DisplayName: %s, Id: %s%n",
    fpRepos.getDisplayName(), fpRepos.getObjectIdentity());
```

3.3 *Discover available metadata*

The initial example in this section demonstrates how to retrieve class descriptions for a given repository and how to work with a single class description's collection of property descriptions.

```
FilePlanRepository fpRepos = ...;

// ---- Retrieve a specific set of class descriptions ----
//  Specify desired class descriptions by their respective
//  symbolic names (or GUID string). If all are desired then
//  use a null or empty String[] parameter value.
String[] desiredClassIdents =
  new String[] { RMClassName.PhysicalRecord, RMClassName.DispositionTrigger };
List<RMClassDescription> specificClassDescs =
  fpRepos.fetchClassDescriptions(desiredClassIdents);
for (RMClassDescription jarmCD : specificClassDescs)
{
  System.out.printf("Class Description DisplayName: %s, Description: %s%n",
                    jarmCD.getDisplayName(), jarmCD.getDescriptiveText());
}
System.out.println();

// ---- Retrieve a single class description using the factory ----
RMClassDescription jarmCD =
  RMFactory.RMClassDescription.fetchInstance(fpRepos, RMClassName.RecordVolume,
                                          RMPropertyFilter.MinimumPropertySet);

// ---- Examine a class description's collection of property descriptions ----
List<RMPropertyDescription> jarmPropDescs = jarmCD.getPropertyDescriptions();
System.out.printf("Class: %s has %d property descriptions.%n",
       jarmCD.getDisplayName(), jarmPropDescs.size());
// Examine one of these property descriptions...
RMPropertyDescription jarmPD = jarmPropDescs.get( jarmPropDescs.size() - 25);
System.out.printf("Some characteristics of property description: %s%n",
       jarmPD.getDisplayName());
System.out.printf("  SymbolicName: %s, DataType: %s, Cardinality: %s%n%n",
       jarmPD.getSymbolicName(), jarmPD.getDataType(), jarmPD.getCardinality());
```

Next, we'll work with the available choice lists defined on a repository. Note that when using the RMFactory to retrieve a specific choice list, the choice list must be identified using its GUID.

```
Repository repository = ...;

// ---- Retrieve the collection of all available choice lists ----
List<RMChoiceList> allChoiceLists = repository.fetchChoiceLists();
for (RMChoiceList jarmChoiceList : allChoiceLists)
{
  System.out.printf("ChoiceList: %s contains %d choice items.%n",
         jarmChoiceList.getDisplayName(),
         jarmChoiceList.getChoiceListValues().size());
}
System.out.println();

// ---- Retrieve a single choice list ----
// NOTE: Choice List must be identified using its GUID.
```

```
String clIdent = "{56C6D10C-05FB-4CBD-9872-55B3F5DAFC2E}";
RMChoiceList jarmCL =
  RMFactory.RMChoiceList.fetchInstance(repository, clIdent);

// ---- Examine the first choice list item ----
List<RMChoiceItem> jarmCLItems = jarmCL.getChoiceListValues();
RMChoiceItem jarmCLItem = jarmCLItems.get(0);
Object choiceItemValue  = null;
ChoiceItemType choiceItemType = jarmCLItem.getChoiceItemType();
switch ( choiceItemType )
{
  case Integer:
    choiceItemValue = jarmCLItem.getChoiceIntegerValue();
    break;
  case String:
    choiceItemValue = jarmCLItem.getChoiceStringValue();
    break;
  case IntegerMidNode:
  case StringMidNode:
    choiceItemValue = "<" + choiceItemType.toString() + ">";
    break;
}
System.out.printf("For ChoiceList %s, 1st ChoiceItem label: %s, value: %s%n",
       jarmCL.getDisplayName(), jarmCLItem.getDisplayName(), choiceItemValue);
```

Finally, let’s examine any marking sets that are defined at the domain level and, thus, common to all repositories. Note that when using the RMFactory to retrieve a specific marking set, the marking set must be identified using its GUID.

```
RMDomain jarmDomain = ...;

// ---- Retrieve the collection of all available marking sets ----
List<RMMarkingSet> allMarkingSets = jarmDomain.fetchMarkingSets();
for (RMMarkingSet jarmMarkingSet : allMarkingSets)
{
  System.out.printf("MarkingSet: %s contains %d markings.%n",
         jarmMarkingSet.getDisplayName(),
         jarmMarkingSet.getMarkings().size());
}
System.out.println();

// ---- Retrieve a single marking set ----
// NOTE: Marking Set must be identified using its GUID.
String msIdent = "{4902E42B-58D6-4E97-8710-DC210DCF9D56}";
RMMarkingSet jarmMS = RMFactory.RMMarkingSet.fetchInstance(jarmDomain, msIdent);

// ---- Examine the available markings ----
System.out.printf("Marking Set %s contains the following markings:%n",
                  jarmMS.getDisplayName());
List<RMMarkingItem> jarmMarkings = jarmMS.getMarkings();
for (RMMarkingItem jarmMarking : jarmMarkings)
{
  System.out.printf("  %s%n", jarmMarking.getMarkingValue());
}
```

3.4 *Work with a file plan container hierarchy*

The collection of available file plan top level containers belonging to a specific file plan repository can be obtained as follows:

```
FilePlanRepository fpRepository = ...;

// ---- Retrieve the collection of top level file plan containers ----
// This example reports on the name and 'Retain Metadata' setting
//  for each available file plan.
List<FilePlan> filePlans =
  fpRepository.getFilePlans(RMPropertyFilter.MinimumPropertySet);
for (FilePlan filePlan : filePlans)
{
  System.out.printf("FilePlan: %s, Retain Metadata setting: %s%n",
         filePlan.getFolderName(), filePlan.getRetainMetadata().toString());
}
System.out.println();

//---- Retrieve a specific file plan container based upon its full path ----
// (Note that it can also be identified by its GUID).
String path = "/Records Management/File Plan";
FilePlan filePlan =
  RMFactory.FilePlan.fetchInstance(fpRepository, path,
                                  RMPropertyFilter.MinimumPropertySet);
System.out.printf("GUID for a file plan '%s' is %s.%n",
       filePlan.getPathName(), filePlan.getObjectIdentity());
```

3.4.1 Navigate a file plan container hierarchy

The example code provided in this section demonstrates several JARM concepts:

- How to navigate through a file plan hierarchy.
- How to make use of both the paging and non-paging capabilities of the JARM PageableSet<T> collection type.
- Determining what child entity types a given container is currently allowed to directly hold.
- Determining if a given container currently has any directly contained records.

```
FilePlan filePlan = ...

// ---- Discover the top level record categories of a specific file plan ----
System.out.printf("Top level record categories for file plan '%s':%n",
                  filePlan.getName());
RMPropertyFilter filter = RMPropertyFilter.MinimumPropertySet;
Integer pageSize = Integer.valueOf(5); // Can be null to use repository default.
PageableSet<RecordCategory> rcResultSet =
  filePlan.fetchRecordCategories(filter, pageSize);
// For demonstration purposes, use the paging mechanism of this PageableSet.
RMPageIterator<RecordCategory> rcPI = rcResultSet.pageIterator();
RecordCategory firstTopRC = null;
while ( rcPI.nextPage() )
{
  List<RecordCategory> rcPage = rcPI.getCurrentPage();
  for (RecordCategory recCat : rcPage)
  {
    if ( firstTopRC == null )
      firstTopRC = recCat;   // Remember 1st record category.

    System.out.printf("  %s%n", recCat.getRecordCategoryName());
  }
}
System.out.println();

// ---- Navigate down through the file plan hierarchy ----
// Arbitrarily choose to move down through each first child container
//  of the current parent container node. We use an iterative
//  method that is passed the current parent node, starting
//  with the first top record category saved from above.
navigateDown(firstTopRC);


...


// Iterative method that reports upon the current
//  container and then moves on down into its first
//  child container (if any).
private void navigateDown(Container currentParent)
{
  // Report on any directly contained records.
  PageableSet<Record> recResultSet =
    ((RecordContainer)currentParent).getRecords(null);
  System.out.printf("Current parent node '%s' of entity type = %s :%n",
         currentParent.getPathName(), currentParent.getEntityType());
  String containsRecordsPhrase = recResultSet.isEmpty() ? "does not" : "does";
  System.out.printf("  %s contain records,%n",
         containsRecordsPhrase);
  System.out.printf("  is capable of containing entity types of: %s%n%n",
         Arrays.toString(currentParent.getAllowedContaineeTypes()));
  PageableSet<Container> subContainers = currentParent.getImmediateSubContainers(null);
  // Use non-paging mode to access first child container (if any).
  Iterator<Container> iter = subContainers.iterator();
  if ( iter.hasNext() )
  {
    Container firstChildContainer = iter.next();
    navigateDown(firstChildContainer);
  }
}
```

3.4.2 Create new file plan container objects

The following example demonstrates how to create a new file plan container node. It also demonstrates the use of two different mechanisms for defining property values as part of the file plan creation process.

```
  FilePlanRepository fpRepository = ...;

  // ---- Create a new file plan container node ----
  // First, define some property values for the new file plan instance.
  RMProperties jarmProps = RMFactory.RMProperties.createInstance(DomainType.P8_CE);
  // Using the 'add' method allows explicit expression
  //  of a property's characteristics.
```

```
jarmProps.add(RMPropertyName.FilePlanName,  // Symbolic name
              DataType.String,              // Data type
              RMCardinality.Single,         // Single- vs. multi-valued
              "MyFilePlan");                // Value
// The more commonly used property value setting mechanism:
jarmProps.putStringValue(RMPropertyName.RMEntityDescription,  // Symbolic name
                         "File plan for learning JARM");      // Value.
jarmProps.putIntegerValue(RMPropertyName.RetainMetadata,
                          RetainMetadata.NeverRetain.getIntValue());
String classIdent = RMClassName.FilePlan;
List<RMPermission> jarmPerms = null;      // Allow default permissions to be used.
// The following 'add' method creates AND saves the new file plan to the repository.
FilePlan newFilePlan = fpRepository.addFilePlan(classIdent, jarmProps, jarmPerms);
System.out.printf("Created new file plan container, Id: %s, path: %s",
       newFilePlan.getObjectIdentity(),
       newFilePlan.getPathName());
```

Now we move on to creating a new record category container. Note that the parent of the new entity could potentially be either a file plan container or, in some circumstances, another record category, both of which support the ***com.ibm.jarm.api.core.RecordCategoryContainer*** interface. This example also demonstrates how to define custom permissions.

```
RecordCategoryContainer parent = ...;

// ---- Create a new record category container ----
// First, define some property values for the instance.
RMProperties jarmProps = RMFactory.RMProperties.createInstance(DomainType.P8_CE);
jarmProps.putStringValue(RMPropertyName.RecordCategoryName, "MyRecCat");
jarmProps.putStringValue(RMPropertyName.RecordCategoryIdentifier, "RC-123");
jarmProps.putStringValue(RMPropertyName.Reviewer, "jSmith");
// Define a custom permission collection for the new record category.
List<RMPermission> jarmPerms = new ArrayList<RMPermission>();
RMPermission jarmPerm = RMFactory.RMPermission.createInstance(DomainType.P8_CE);
jarmPerm.setGranteeName("RM Privileged Users");
jarmPerm.setAccessMask(RMAccessRight.Read, RMAccessRight.Write);
jarmPerm.setAccessType(RMAccessType.Allow);
jarmPerm.setInheritableDepth(0);
jarmPerms.add(jarmPerm);
// The following 'add' method creates AND saves the new record category.
String classIdent = RMClassName.RecordCategory;
RecordCategory newRC = parent.addRecordCategory(classIdent, jarmProps, jarmPerms);
System.out.printf("Created new record category %s.%n", newRC.getRecordCategoryName());
```

RecordFolderContainer is the interface that allows creation of a new ***RecordFolder*** under the appropriate parent container type (either a record category or, in some cases, another record folder). The following code snippet makes use of this interface and also demonstrates an example of how the use of an ***RMFactory*** '***getInstance***' method (instead of '***fetchInstance***') can avoid an unnecessary round trip call to the repository.

```
RecordCategory recCat = ...;
RecordFolderParent parent = recCat;
FilePlanRepository fpRepository =
  (FilePlanRepository)(recCat.getRepository());

// ---- Create a new record folder container ----
// First, define some property values for the new instance.
RMProperties jarmProps = RMFactory.RMProperties.createInstance(DomainType.P8_CE);
jarmProps.putStringValue(RMPropertyName.RecordFolderName, "MyRecFolder");
jarmProps.putStringValue(RMPropertyName.RecordFolderIdentifier, "RF-456");
jarmProps.putStringValue(RMPropertyName.Reviewer, "jSmith");
// 'HomeLocation' is an object-type property that only accepts
//  an instance of the custom object class 'Location'. To provide a value
//  for this object type property we only need a reference to such --
//  we do NOT have to perform an actual round trip to the repository
//  to fetch the actual 'Location' object in question.
String locationIdent = "{461A68F9-5F7B-4CC5-B9B3-4F35FD78CAC8}";
Location myLocation = RMFactory.Location.getInstance(fpRepository, locationIdent);
jarmProps.putObjectValue(RMPropertyName.HomeLocation, myLocation);
// The following 'add' method creates and saves the new record folder.
//  If applicable to the type of record folder, an initial child record volume
//  is automatically created under the new record folder.
String classIdent = RMClassName.HybridRecordFolder;
RecordFolder newRF = parent.addRecordFolder(classIdent, jarmProps, null);
System.out.printf("Created new record folder %s.%n", newRF.getRecordFolderName());
RecordVolume firstVol = newRF.getActiveRecordVolume();
System.out.printf(" Automatically created its first child volume: %s.%n",
                  firstVol.getVolumeName());
```

The final example of this section demonstrates creating an additional, new record volume for a given ***com.ibm.jarm.api.core.RecordVolumeContainer*** parent. The newly created record volume becomes the parent's new "active volume" and the previous active volume is automatically closed. There are three alternative ways to name the new record volume:

1. Define a property value using the symbolic name ***RMPropertyName.VolumeName***.
2. Provide a non-null value for the ***volumeName*** parameter of the ***RecordVolumeContainer.addRecordVolume*** method.
3. Do neither of the above and allow the new record volume to be auto-named (the most common mechanism used).

```
  RecordVolumeContainer parent = ...;

  // ---- Create a new record volume container ----
  // First, define some property values for the new instance.
  RMProperties jarmProps = RMFactory.RMProperties.createInstance(DomainType.P8_CE);
  jarmProps.putStringValue(RMPropertyName.Reviewer, "jSmith");
  // This example simply allows the new record volume to be auto-named.
  String volumeName = null;
  String classIdent = RMClassName.RecordVolume;
  List<RMPermission> jarmPerms = null; // Allow defaults to be applied.
  // The following 'add' method creates AND saves the new record volume AND
  //  closes the previously 'active volume'.
  RecordVolume newRecVol =
    parent.addRecordVolume(classIdent, volumeName, jarmProps, jarmPerms);
  System.out.printf("Created new record volume %s%n",
        newRecVol.getVolumeName());
```

3.4.3 Perform operations on file plan container objects

3.4.3.1 Close and reopen a container

Containers supporting the RecordContainer interface can be closed and reopened as follows:

```
RecordContainer container = ...;

System.out.printf("Originally, container's Closed status is: %s%n",
       container.isClosed());
// Close this container by providing an audit reason.
container.close("Closing for demo purposes.");
System.out.printf("After Close operation, Container's status is now: %s%n",
       container.isClosed());
// Finally, reopen the container.
container.reopen();
System.out.printf("After Reopen operation, Container's status is now: %s%n",
       container.isClosed());
```

3.4.3.2 Inactive and reactivate a container

Containers supporting the ***RecordContainer*** interface can be inactivated and reactivated as follows:

```
RecordContainer container = ...;

System.out.printf("Originally, container's Inactive status is: %s%n",
       container.isInactive());
// Inactivate this container by providing an audit reason.
container.setInactive("Inactivating for demo purposes.");
System.out.printf("After setInactive operation, Container's status is now: %s%n",
       container.isInactive());
// Finally, reopen the container.
container.setActive();
System.out.printf("After setActive operation, Container's status is now: %s%n",
       container.isInactive());
```

3.4.3.3 Relocate a record category or record folder

The ***RecordCategory*** and ***RecordFolder*** interfaces both support the ***move*** method whereby the entity can be unfiled from its current parent container and refiled under a different parent container (a record volume cannot be moved). The following example demonstrates this for a record category:

```
RecordCategory recCat = ...;
RecordCategoryContainer newRCParent = ...;

System.out.printf("Before the move, record category's current path is: '%s'%n",
       recCat.getPathName());
// Relocate the record category to the new parent by providing an audit reason.
recCat.move(newRCParent, "Relocating for demo purposes.");
System.out.printf("After the move, record category's new path is: '%s'%n",
                  recCat.getPathName());
```

3.4.4 Work with Defensibly Disposable Containers

Before attempting to work with defensibly disposable containers, it is a best practice to ensure that the current file plan repository supports this feature:

```
FilePlanRepository fpRepository = ...;
// Verify that Defensibly Disposable Containers are supported
//  on this file plan repository.
boolean isDDSupported = fpRepository.supportsDefensibleDisposal();
```

A defensibly disposable container is created using the ***com.ibm.jarm.api.core.DefensiblyDisposableContainerParent*** interface which is implemented by both record category and record folder types. (However, as of Enterprise Records-5.1.2, only the record category type is currently capable of containing or performing as a defensibly disposable container). This interface provides both an indication of the candidate parent container's eligibility to host a defensibly disposable child container and the ability to actually create such a child within it. As shown in the following example, when defining a new defensibly disposable container, in addition to the normal collection of property values needed to create a new record category, two additional defensibly disposable container characteristics must also be supplied:

Trigger property name	The symbolic name identifying the record-level date-time property used to trigger disposition processing for each child record contained by a defensibly disposable container.
Retention period	Retention period to apply to each contained record in the form of three integer values representing years, months and days.

```
DefensiblyDisposableContainerParent potentialDDParent = ...;

if ( potentialDDParent.canContainDefensibleDisposalContainer() )
{ // This parent container IS capable of hosting a defensibly
  //  disposable child container.

  // ---- Create a Defensibly Disposable Container ----
  // First define property values that are needed for a new record category.
  RMProperties jarmProps = RMFactory.RMProperties.createInstance(DomainType.P8_CE);
  jarmProps.putStringValue(RMPropertyName.RecordCategoryName, "MyDDContainer");
  jarmProps.putStringValue(RMPropertyName.RecordCategoryIdentifier, "RC-123");
  jarmProps.putStringValue(RMPropertyName.Reviewer, "jSmith");
  // Let default permissions be assigned to the new child container.
  List<RMPermission> jarmPerms = null;
  // Let unique identifier of the new container be automatically generated.
  String newId = null;
  // Define the defensibly disposable characteristics.
  String triggerPropertyName = RMPropertyName.SentOn;
  int retentionInYears  = 5;
  int retentionInMonths = 0;
  int retentionInDays   = 0;
  RecordContainer newDDContainer =
    potentialDDParent.addDefensiblyDisposableContainer(triggerPropertyName,
                                                       retentionInYears,
                                                       retentionInMonths,
                                                       retentionInDays,
                                                       jarmProps,
                                                       jarmPerms,
                                                       newId);
}
```

The JARM ***com.ibm.jarm.api.core.DefensiblyDisposable*** interface provides the following services relating to defensibly disposable containers:

- Its ***isADefensiblyDisposableContainer*** method indicates whether a given container, is in fact, already a defensibly disposable type.
- Getters/setters for both the trigger property name and retention period values of an existing defensibly disposable container.
- Support for the conversion of an existing record category into a defensibly disposable container.

There are actually two methods that support the conversion process:

isEligibleForConversion	This is a convenience method primarily to allow a preliminary check of the eligibility of a given container. The return value of this method is a bit unusual in that it can be an instance of ***RMRuntimeException***. If the return value is null, then the container IS eligible, otherwise the returned exception describes why the container is ineligible.
convertToDefensiblyDisposable	This method actually performs the conversion process attempt. It will throw an ***RMRuntimeException*** if the container is ineligible.

Depending upon the number of existing records contained by the container that is the target of a potential defensibly disposal conversion process, the internal validation process that JARM normally performs can potentially take a long time to complete since each record must be examined. For this reason, the ***convertToDefensiblyDisposable*** method does include the ***skipValidation*** input parameter to allow bypassing of this validation process. However, using this bypass mechanism SHOULD ONLY be used if a previous call to ***isElegibleForConversion*** succeeded.

3.5 Work with records

3.5.1 Declare a new record (and undeclare)

Both physical and electronic record declaration is performed using one of the several declare methods found on the ***com.ibm.jarm.api.core.RecordContainer*** interface (implemented by the various file plan container types). All of these declare methods have the following items in common:

- The container implementing the ***RecordContainer*** interface automatically becomes the initial container into which the new record is filed and also the record's security folder from which it inherits security permissions.
- The identity of the applicable record class must be provided.
- An ***RMProperties*** collection for the new record must be provided.
- A collection of ***RMPermission*** can optionally be provided if desired. The record's resulting permissions will be a combination of this collection and those permissions contributed by the record's security folder.
- An optional collection of additional ***RecordContainer*** instances into which the record should also be filed. Note that the ability to "multi-file" a record may be disabled for a particular file plan object store (primarily for improved disposition processing performance). If a record folder type container is specified AND the particular record folder type requires the use of child record volumes, then the newly declared record instance is actually filed into the current active child volume of the record folder; this is true both for the initial container and any optional additional containers involved.

For electronic record declaration, additional information is required as to which documents the record is being declared for (a physical record has no electronic content). A single electronic record can be declared for one or more separate documents or multiple versions of a single document. In either case, all associated content for a given electronic record must reside on the same content repository. For convenience purposes, these content items can be provided in either of two forms:

- As a ***List<ContentItem>*** collection, each member of which represents either a single document or document version.
- A combination of a single ***ContentRepository*** instance and a ***List<String>*** collection containing GUIDs of the appropriate documents/document versions.

The following record declaration example snippet is that of a physical record that is being multi-filed and has some additional permissions defined for it.

```
FilePlanRepository fpRepository  = ...;
RecordContainer primaryContainer = ...;
RecordContainer secondaryContainer = ...;

String classIdent = RMClassName.PhysicalRecord;
// Define properties for the new record.
RMProperties jarmProps = RMFactory.RMProperties.createInstance(DomainType.P8_CE);
jarmProps.putStringValue(RMPropertyName.DocumentTitle, "MyPhysicalRecord");
Location actualLocation =
  RMFactory.Location.getInstance(fpRepository,
                                 "{461A68F9-5F7B-4CC5-B9B3-4F35FD78CAC8}");
jarmProps.putObjectValue(RMPropertyName.HomeLocation, actualLocation);
// Define addition RMPermission instances.
List<RMPermission> jarmPerms = new ArrayList<RMPermission>();
RMPermission jarmPerm = RMFactory.RMPermission.createInstance(DomainType.P8_CE);
jarmPerm.setGranteeName("RM Privileged Users");
jarmPerm.setAccessMask(RMAccessRight.Read, RMAccessRight.Write);
jarmPerm.setAccessType(RMAccessType.Allow);
jarmPerm.setInheritableDepth(0);
jarmPerms.add(jarmPerm);
// Define the additional container.
List<RecordContainer> additionalContainers = new ArrayList<RecordContainer>(1);
additionalContainers.add(secondaryContainer);
// Ready to perform new physical record declaration.
Record newRecord =
  primaryContainer.declare(classIdent, jarmProps, jarmPerms, additionalContainers);
System.out.printf("Successfully declared new physical record: %s%n",
                  newRecord.getObjectIdentity());
```

The next declaration sample is for an electronic record that uses ***ContentItem*** instances to specify the documents to associate the new record with.

```
  RecordContainer primaryContainer = ...;
  ContentItem targetDocument = ...;

  String classIdent = RMClassName.ElectronicRecord;
  // Define properties for the new record.
  RMProperties jarmProps = RMFactory.RMProperties.createInstance(DomainType.P8_CE);
```

```
    jarmProps.putStringValue(RMPropertyName.DocumentTitle, "MyElectronicRecord_1");
    // No additional permission specified.
    List<RMPermission> jarmPerms = null;
    // No additional record filings needed.
    List<RecordContainer> additionalContainers = null;
    // Collection of content to declare for.
    List<ContentItem> targetDocuments = new ArrayList<ContentItem>(1);
    targetDocuments.add(targetDocument);
    // Ready to perform new electronic record declaration.
    Record newRecord =
      primaryContainer.declare(classIdent, jarmProps, jarmPerms,
                               additionalContainers,
                               targetDocuments);
    System.out.printf("Successfully declared new electronic record, %s," +
                      " for document %s.%n",
                      newRecord.getObjectIdentity(),
                      targetDocument.getObjectIdentity());
```

A similar electronic record declaration example but using document GUIDs instead of ***ContentItem***:

```
  RecordContainer primaryContainer = ...;
  ContentRepository contentRepository = ...;
  String docID1 = ...;
  String docID2 = ...;

  String classIdent = RMClassName.ElectronicRecord;
  // Define properties for the new record.
  RMProperties jarmProps = RMFactory.RMProperties.createInstance(DomainType.P8_CE);
  jarmProps.putStringValue(RMPropertyName.DocumentTitle, "MyElectronicRecord_2");
  // No additional permission specified.
  List<RMPermission> jarmPerms = null;
  // No additional record filings needed.
  List<RecordContainer> additionalContainers = null;
  // Collection of content to declare for.
  List<String> targetDocIDs = new ArrayList<String>(2);
  targetDocIDs.add(docID1);
  targetDocIDs.add(docID2);
  // Ready to perform new electronic record declaration.
  Record newRecord =
    primaryContainer.declare(classIdent, jarmProps, jarmPerms,
                             additionalContainers,
                             contentRepository,
                             targetDocIDs);
  System.out.printf("Successfully declared new electronic record, %s," +
                    " for documents:%n", newRecord.getObjectIdentity());
  PageableSet<ContentItem> associatedContent = newRecord.getAssociatedContentItems();
  for (ContentItem contentItem : associatedContent)
  {
    System.out.printf("   %s%n", contentItem.getObjectIdentity());
  }
```

The JARM ***Record*** interface does provide an ***undeclare*** method that is intended for the use case of undoing a previously misguided declaration (for example, a document that is not really a business record). This operation dis-associates any content items from the record and then hard deletes the record object. The original content item should then be eligible once again for declaration of a new record.

```
// ---- Undeclare an existing record ----
// For demonstration purposes, first acquire
//  any associated content items of the record.
PageableSet<ContentItem> contentItems =
  recordToUndeclare.getAssociatedContentItems();
System.out.printf("Before the undeclare operation:%n");
for (ContentItem contentItem : contentItems)
{
  System.out.printf("  Doc: %s -- is eligible for declaration? %s%n",
         contentItem.getObjectIdentity(),
         contentItem.isEligibleForDeclaration());
}

// Perform the undeclare operation.
recordToUndeclare.undeclare();

// To verify, refresh the content items and re-check
//  their declaration eligibilities.
System.out.printf("%nAfter the undeclare operation:%n");
for (ContentItem contentItem : contentItems)
{
  contentItem.refresh();
  System.out.printf("  Doc: %s -- is eligible for declaration? %s%n",
                    contentItem.getObjectIdentity(),
                    contentItem.isEligibleForDeclaration());
}
```

3.5.2 File and unfile a record

If enabled for a particular file plan repository, a record can be multi-filed into more than one container. Note that the record's current "security folder" container does not change when it is filed elsewhere. Here is an example of filing an existing record into an additional file plan container:

```
RecordContainer additionalContainer = ...;
Record recordToFile = ...;

// ---- Filing a record into an additional container ----
additionalContainer.fileRecord(recordToFile);
// Alternatively, the record's Identity could also be used as follows:
//String recordId = recordToFile.getObjectIdentity();
//additionalContainer.fileRecord(recordId);

List<Container> recordContainers = recordToFile.getContainedBy();
System.out.printf("Multi-filed record now located in %d containers.%n",
       recordContainers.size());
```

A multi-filed record can be unfiled from one of its containers as long as it remains filed in at least one container. If the container involved in an unfile operation happens to be the record's current security folder, then one of the records remaining containers will be automatically assigned as the record's new security folder.

```
RecordContainer container = ...;
Record recordToUnfile = ...;

container.unfileRecord(recordToUnfile);
// Alternatively, the record Identity could also be used as follows:
//String recordId = recordToUnfile.getObjectIdentity();
//container.unfileRecord(recordId);

List<Container> recordContainers = recordToUnfile.getContainedBy();
System.out.printf("Unfiled record is now located in %d containers.%n",
       recordContainers.size());
```

3.5.3 Move a record

A record can be moved from one container into another container within the same file plan as long as the target container is capable of containing such a record. If the source container happens to be the record's current security folder, then the target container will be assigned as the record's new security folder.

```
RecordContainer sourceContainer = ...;
RecordContainer destinationContainer = ...;
Record recordToMove = ...;

// ---- Moving a record ----
System.out.printf("Record's original security folder: %s%n",
       recordToMove.getSecurityFolder().getPathName());
// An audit reason is required.
String reasonForMove = "Moving record for demo purposes";
recordToMove.move(sourceContainer, destinationContainer, reasonForMove);

// Refresh to force a refetch of record state.
recordToMove.refresh();
System.out.printf("Record's security folder after the move: %s%n",
       recordToMove.getSecurityFolder().getPathName());
```

3.5.4 Copy a record

The following example demonstrates how to make a copy of an existing record. Normally all non-system, non-read-only source record properties are copied over to the new record instance. An optional ***RMProperties*** collection can be specified whose contents will override any matching properties from the original record. An instance of a Record Copy Link is also created that links the original record to its new copy.

For an electronic record, all associated content will also be duplicated on the same content repository. Note however, copying of an electronic record that is associated with P8 federated content is NOT supported.

```
RecordContainer targetContainer = ...;
Record originalRecord = ...;

// ---- Copy a record ----
// Define a collection of override property values.
RMProperties overrideProps =
  RMFactory.RMProperties.createInstance(DomainType.P8_CE);
overrideProps.putStringValue(RMPropertyName.DocumentTitle,
                             "MyRecordCopy");
overrideProps.putStringValue(RMPropertyName.RMEntityDescription,
                             "New copy description");
overrideProps.putDateTimeValue(RMPropertyName.SentOn,
                               new Date());
```

```
// Perform the actual copy operation placing the new record
//  into the specified target container.
Record recordCopy =
  originalRecord.copy(targetContainer, overrideProps);

System.out.printf("Description of original record: %s%n",
       originalRecord.getProperties()
                    .getStringValue(RMPropertyName.RMEntityDescription));
System.out.printf("Description of new, copied record: %s%n",
       recordCopy.getProperties()
                .getStringValue(RMPropertyName.RMEntityDescription));
```

3.5.5 Supersede an existing record

The concept of a "newer" record "superseding" an existing, target record typically occurs when the first record represents updated or improved content over that of the target. The JARM ***Record*** interface provides the supersede method that can be called on the "newer" record as shown below.

```
Record targetRecord = ...;
Record newerRecord  = ...;

// ---- Supersede one record by another ----
// The 'newer' record can only supersede one other
//  record, but the 'target' record could possibly
//  have been superseded by more than one 'newer'
//  record.
newerRecord.supersede(targetRecord);

// Refresh both and examine some supersede-related
//  characteristics of each.
targetRecord.refresh();
newerRecord.refresh();
System.out.printf("For target record:%n");
System.out.printf("  Is superseded? %s%n",
                  targetRecord.isSuperseded());
System.out.printf("  Last superseded on: %s%n",
  targetRecord.getProperties()
            .getDateTimeValue(RMPropertyName.SupersededDate));
List<Object> rawSupersededBy =
  targetRecord.getProperties()
            .getObjectListValue(RMPropertyName.SupersededBy);
System.out.printf("  Superseded by:%n");
for (Object rawObject : rawSupersededBy)
{
  Record supersedingRecord = (Record)rawObject;
  System.out.printf("    %s%n",
         supersedingRecord.getObjectIdentity());
}

System.out.printf("%nFor newer record:%n");
Object rawSupersedes =
  newerRecord.getProperties()
           .getObjectValue(RMPropertyName.Supersedes);
System.out.printf("  Supersedes: %s%n",
       ((Record)rawSupersedes).getObjectIdentity());
```

3.5.6 Link two records together

The following is an example of establishing a link between two records:

```
FilePlanRepository fpRepository = ...;
Record headRecord = ...;
Record tailRecord = ...;

// ---- Establish a link between records ----
// Use factory to create a new, in-memory instance of a link.
String linkClass = RMClassName.RecordSeeAlsoLink;
RMLink newLink =
  RMFactory.RMLink.createInstance(fpRepository, linkClass);
// Define the required head and tail property values and
//  any other property values that might be desired.
RMProperties linkProps = newLink.getProperties();
linkProps.putObjectValue(RMPropertyName.Head, headRecord);
linkProps.putObjectValue(RMPropertyName.Tail, tailRecord);
linkProps.putStringValue(RMPropertyName.LinkName, "My Link Name");
// Finally persist the new link instance to the repository.
newLink.save(RMRefreshMode.Refresh);
System.out.printf("Created new RMLink instance: %s%n",
       newLink.getObjectIdentity());
```

Next, is an example of a search that queries for all link instances that a given record is involved in:

```
FilePlanRepository fpRepository = ...;
Record record = ...;

String sqlPattern =
  "SELECT l.Id, l.LinkName FROM Relation l " +
  "WHERE l.Head = Object('%1$s') OR "        +
        "l.Tail = Object('%1$s') ";

String recordId = record.getObjectIdentity();
String sqlStmt =
  String.format(sqlPattern, recordId);
RMSearch jarmSearch = new RMSearch(fpRepository);
PageableSet<RMLink> resultSet = (PageableSet<RMLink>)
  jarmSearch.fetchObjects(
              sqlStmt,  // P8 SQL statement
              EntityType.RMLink,  // Type of object to return
              null,     // Default page size
              null,     // Bring back all properties
              Boolean.FALSE); // Not going to page.

for (RMLink link : resultSet)
{
  String headOrTail;
  if ( recordId.equalsIgnoreCase(link.getHead().getObjectIdentity()) )
    headOrTail = "head";
  else
    headOrTail = "tail";

  System.out.printf("For RMLink: %s, record is the link's %s.%n",
         link.getObjectIdentity(), headOrTail);
}
```

3.6 *Work with Holds*

3.6.1 Create and retrieve holds

To create a new hold on a file plan repository:

```
FilePlanRepository fpRepository = ...;

// ---- Create a new Hold ----
// Use the factory to create a new, in-memory instance.
Hold newHold = RMFactory.Hold.createInstance(fpRepository);
// Use Hold setters to define some property values.
//  (Same could be performed using Hold's RMProperties collection).
newHold.setHoldName("My Demo Hold");
newHold.setActiveState(true);
newHold.setHoldReason("Some law suit");
// Following HoldType value must come from HoldTypeList choice list.
newHold.setHoldType("Litigation");
// Now persist the new Hold onto the repository.
newHold.save(RMRefreshMode.Refresh);
System.out.printf("Created new Hold: %s%n",
       newHold.getObjectIdentity());
```

Retrieve all existing holds defined on a file plan repository:

```
FilePlanRepository fpRepository = ...;

// ---- Retrieve all existing Holds ----
List<Hold> allHolds =
  fpRepository.getHolds(RMPropertyFilter.MinimumPropertySet);
System.out.printf("Repository has %d number of holds.%n",
                  allHolds.size());
```

To retrieve a specific hold:

```
FilePlanRepository fpRepository = ...;

// ---- Retrieve an existing Hold by its GUID ----
String holdId = allHolds.get(0).getObjectIdentity();
RMPropertyFilter filter = RMPropertyFilter.MinimumPropertySet;
Hold hold = RMFactory.Hold.fetchInstance(fpRepos, holdId, filter);
System.out.printf("Retrieved hold: %s%n", hold.getHoldName());
```

3.6.2 Place and remove a hold

A hold can be placed upon either a record or record container. When placed upon a container, it applies to all of the sub-hierarchy of that container (both sub-containers and records). The following demonstrates how to place a hold on a specific ***Holdable*** object and then remove that same hold.

```
Holdable holdableObject = ...;
Hold hold = ...;
```

```
// ---- Place a hold on a Holdable object ----
holdableObject.placeHold(hold);
System.out.printf("Holdable object -- %s%n",
       holdableObject.isOnHold(false));

// ---- Remove an existing hold placement ----
holdableObject.removeHold(aHold);
System.out.printf("Holdable object -- %s%n",
       holdableObject.isOnHold(false));
```

3.6.3 Determine hold status of a Holdable object

The ***Holdable*** interface provides several mechanisms that report on the hold state of a given object:

```
Holdable holdable = ...;

// ---- Any hold directly placed on this object? ----
boolean checkHierParents = false;
System.out.printf("Any hold directly placed on this object? %s%n",
       holdable.isOnHold(checkHierParents));

// ---- Any hold place on this object or any hier. parent? ----
checkHierParents = true;
System.out.printf("Does any hold affect this object? %s%n",
       holdable.isOnHold(checkHierParents));

// ---- What holds are directly place on this object? ----
List<Hold> directHolds = holdable.getAssociatedHolds();

// ---- Which hierarchical parents are on hold? ----
List<RecordContainer> heldParents = holdable.getParentsOnHold();

// ---- Any hierarchical child on hold? ----
System.out.printf("Any child of this object on hold? %s%n",
       holdable.isAnyChildOnHold());
```

3.6.4 Determine which objects a hold is placed upon

The ***Hold*** interface provides two separate methods for determining which containers and records the hold is directly placed upon:

```
FilePlanRepository fpRepository = ...;
String holdIdent = ...;

// ---- Determine where Hold is placed ----
RMPropertyFilter filter = RMPropertyFilter.MinimumPropertySet;
Hold hold = RMFactory.Hold.fetchInstance(fpRepository, holdIdent, filter);

Integer pageSize = Integer.valueOf(100);
// Get collection of directly held containers
PageableSet<Container> heldContainers =
  hold.getAssociatedContainers(pageSize);
// Get collection of directly help records
PageableSet<Record> heldRecords =
  hold.getAssociatedRecords(pageSize);
```

3.7 Work with disposition schedules

3.7.1 Find existing schedules and related artifacts

Existing disposition schedules, actions and triggers can be discovered using the ***FilePlanRepository*** interface or the appropriate ***RMFactory*** inner class:

```
FilePlanRepository fpRepository = ...;

RMPropertyFilter filter = RMPropertyFilter.MinimumPropertySet;
// ---- Disposition Schedules ----
List<DispositionSchedule> allSchedules =
  fpRepository.getDispositionSchedules(filter);
// A specific instance:
DispositionSchedule aSchedule =
  RMFactory.DispositionSchedule.fetchInstance(fpRepository, scheduleId, filter);

// ---- Disposition Actions ----
List<DispositionAction> allActions =
  fpRepository.getDispositionActions(filter);
// A specific instance:
DispositionAction anAction =
  RMFactory.DispositionAction.fetchInstance(fpRepository, actionId, filter);

// ---- Disposition Triggers ----
List<DispositionTrigger> allTriggers =
  fpRepository.getDispositionTriggers(filter);
// A specific instance:
DispositionTrigger aTrigger =
  RMFactory.DispositionTrigger.fetchInstance(fpRepos, triggerId, filter);
```

3.7.2 Create a new action

Here is an example of creating a new disposition action for a review. Note the need to associate a proper workflow with the new action.

```
FilePlanRepository fpRepository = ...;

// ---- Create Disposition Action ----
// Use factory to create new in-memory instance.
DispositionActionType type = DispositionActionType.Review;
DispositionAction newAction =
  RMFactory.DispositionAction.createInstance(fpRepository, type);
newAction.setActionName("My Review Action");
// Define required workflow for the action
String wflId = "{8721B96F-7559-4C6E-B04C-FCFA1DEE5CB3}";
RMWorkflowDefinition wfl =
  RMFactory.WorkflowDefinition.getInstance(fpRepository, wflId);
newAction.setAssociatedWorkflow(wfl);
// Persist new action to the repository.
newAction.save(RMRefreshMode.Refresh);
```

3.7.3 Create a new trigger

This demonstrates how to create a new disposition trigger whose event type is that of a fixed future date:

```
FilePlanRepository fpRepository = ...;
```

```
// ---- Create Disposition Trigger ----
// Use factory to create new in-memory instance.
DispositionTriggerType type =
  DispositionTriggerType.PredefinedDateTrigger;
DispositionTrigger newTrigger =
  RMFactory.DispositionTrigger.createInstance(fpRepository, type);
// Define properties for the new trigger.
newTrigger.setTriggerName("My Date Trigger");
Calendar futureDate = Calendar.getInstance();
futureDate.set(Calendar.YEAR, 2016);
futureDate.set(Calendar.MONTH, 3);
futureDate.set(Calendar.DAY_OF_MONTH, 12);
newTrigger.setDateTime(futureDate.getTime());
// Persist the new trigger to the repository.
newTrigger.save(RMRefreshMode.Refresh);
```

3.7.4 Create a new schedule

Here an example of the definition of a new disposition schedule that does not have any phases defined as of yet:

```
FilePlanRepository fpRepository  = ...;
DispositionTrigger cutoffTrigger = ...;

// ---- Create Disposition Schedule ----
// Use factory to create new in-memory instance.
DispositionSchedule newSchedule =
  RMFactory.DispositionSchedule.createInstance(fpRepository);
// Define properties for the new schedule.
newSchedule.setScheduleName("My Demo Schedule");
newSchedule.setDispositionTigger(cutoffTrigger);
newSchedule.setScreeningRequired(Boolean.FALSE);
newSchedule.setCutoffBase("EventDate");
// Persist the new schedule to the repository.
newSchedule.save(RMRefreshMode.Refresh);
```

Continuing with this example, we define two phases for the schedule, the second of which makes use of alternate retention criteria:

```
DispositionAction p1Action = ...;
DispositionPhaseList phaseList = newSchedule.getDispositionPhases();

// Create an initial disposition phase and set its properties.
DispositionPhase p1 = newSchedule.createNewPhase("Phase1");
p1.setPhaseAction(p1Action);
p1.setScreeningRequired(false);
p1.setRetentionPeriod(0, 6, 0);
// Add this new phase the schedule's phase list.
phaseList.add(p1);

// Create a 2nd phase...
DispositionAction p2Action = ...;
DispositionPhase p2 = newSchedule.createNewPhase("Phase2");
p2.setPhaseAction(p2Action);
p2.setScreeningRequired(false);
```

```
p2.setRetentionPeriod(1, 0, 0);
// Define an alternate retention for this 2nd phase...
AlternateRetentionList altRetents = p2.getAlternateRetentions();
AlternateRetention altRetent = p2.createAlternateRetention();
altRetent.setRetentionBase(RMPropertyName.CutoffDate);
altRetent.setConditionXML("[xml - see an existing for format]");
altRetent.setRetentionPeriod(2, 3, 0);
altRetents.add(altRetent);
// Add this 2nd phase to the schedule's phase list.
phaseList.add(p2);

// Finally, perform a save of the schedule that will persist
//  all phase-related artifacts.
newSchedule.getProperties()
           .putStringValue(RMPropertyName.ReasonForChange, "To add phases");
newSchedule.save(RMRefreshMode.NoRefresh);
```

3.7.5 Modify a schedule

Modifying a schedule basically involves using the appropriate setters and/or manipulating the schedule ***RMProperties*** collection directly. The schedule's ***DispositionPhaseList*** is used both to get access to and to add/remove phases. Likewise, an individual phase's ***AlternateRetentionList*** is used both to get access to and to add/remove remove alternate retention instances on that phase. When all changes have been completed, the ***DispositionSchedule.save*** method should then be called to persist all of these schedule related entities.

3.7.6 Assign a schedule

A disposition schedule is typically assigned to either a record category or record folder type container both of which support the ***DispositionAllocatable*** interface. Here is an example:

```
DispositionAllocatable scheduleTarget = ...;
DispositionSchedule schedule = ...;

// ---- Assign a disposition schedule to a container ----
// In this example, no schedule propagation is desired.
SchedulePropagation assignProp = SchedulePropagation.NoPropagation;
scheduleTarget.assignDispositionSchedule(schedule, assignProp);
```

Likewise, when clearing a container of a previous schedule assignment, the propagation mode can also be specified:

```
// ---- Remove a schedule assignment ----
// Request that all existing inheritors also have the schedule removed.
SchedulePropagation removeProp = SchedulePropagation.ToAllInheritors;
scheduleTarget.clearDispositionAssignment(removeProp);
```

3.8 Perform search operations

3.8.1 Using Object-type return values

The following code demonstrates how to set up and perform a non-CBR type search operation whose result set consists of JARM Objects. In this particular example, the result set is accessed in a paged manner.

```
FilePlanRepository fpRepository = ...;
```

```
  // ---- Perform a search that returns objects ----
  RMSearch jarmSearch = new RMSearch(fpRepository);

  // P8 SQL statement. Note that SELECT clause includes one object type.
  String sqlStmt = "SELECT r.Id, r.DocumentTitle, r.SecurityFolder FROM RecordInfo r " +
                   "WHERE r.DateOfLastReview > 2013-03-14 ";
  // Expected type of object to return in result set.
  EntityType returnType = EntityType.Record;

  // Define a custom RMPropertyFilter to control amount of detail
  //  returned for the 'SecurityFolder'.
  RMPropertyFilter filter = new RMPropertyFilter();
    // Recursion depth to go down.
  Integer maxRecursion = Integer.valueOf(1);
    // No content involved - not applicable.
  Long maxContentSize  = null;
    // Use the repository default setting.
  Boolean levelDependents = null;
    // List of property symbolic names to include.
  String symbolicNames = "SecurityFolder Id FolderName RMEntityType";
  // Desired result set page size.
  Integer pageSize = Integer.valueOf(10);
  filter.addIncludeProperty(maxRecursion, maxContentSize, levelDependents,
                            symbolicNames, pageSize);

  // Perform the search operation...
  Boolean continuable = Boolean.TRUE; // use paging
  PageableSet<Record> objResultSet =
    (PageableSet<Record>)(jarmSearch.fetchObjects(sqlStmt,
                                                  returnType,
                                                  pageSize,
                                                  filter,
                                                  continuable));

  // Page through the result set...
  int pageNumber = 0;
  RMPageIterator<Record> pi = objResultSet.pageIterator();
  while ( pi.nextPage() )
  {
    pageNumber++;
    List<Record> currentPage = pi.getCurrentPage();
    System.out.printf("Search result page#%d, contains %d records.%n",
           pageNumber, currentPage.size());
    for (Record record : currentPage)
    {
      Container secFolder =
(Container)(record.getProperties().getObjectValue("SecurityFolder"));
      String secFolderName = secFolder.getFolderName();
      // ...
    }
  }
```

3.8.2 Using ResultRow return values

In the next search example, since the SELECT clause includes items from differing tables due to a JOIN operation, the result set must be in the form of a generalized ***ResultRow*** type. Each ***ResultRow*** basically has its own ***RMProperties*** collection whose members represent the requested SELECT clause columns. Note how this example uses an SQL "AS" alias for one of the columns in the SELECT clause and that the same alias name becomes the access mechanism used on the ***ResutlRow***'s ***RMProperties*** collection.

```
FilePlanRepository fpRepository = ...;

// ---- Perform a search that returns ResultRows ----
RMSearch jarmSearch = new RMSearch(fpRepository);

// P8 SQL statement that includes a JOIN clause.
//  Note use of column alias 'DateRecFiled'.
String sqlStmt =
  "SELECT r.DocumentTitle, rcr.DateCreated AS DateRecFiled " +
    "FROM RecordInfo r " +
    "INNER JOIN ReferentialContainmentRelationship rcr ON r.This = rcr.Head " +
  "WHERE rcr.Tail = Object('/Records Management/File Plan/JLWTopRecCat') ";

// No object values returned.
RMPropertyFilter filter = null;
// Not going to page.
Integer pageSize = null;

// Perform the search operation...
Boolean continuable = Boolean.FALSE;  // non-paging
PageableSet<ResultRow> rowResultSet =
  jarmSearch.fetchRows(sqlStmt, pageSize, filter, continuable);

// Use normal iterator to access results in a non-paged manner.
Iterator<ResultRow> iter = rowResultSet.iterator();
while ( iter.hasNext() )
{
  ResultRow row = iter.next();
  RMProperties rowProps = row.getProperties();
  String docTitle = rowProps.getStringValue("DocumentTitle");
  // Note use of alias name as the row result 'column' name.
  Date fileDate   = rowProps.getDateTimeValue("DateRecFiled");
  System.out.printf("Record '%s' was filed on %s%n",
         docTitle, fileDate);
}
```

3.8.3 Content-based retrieval

For Enterprise Records, the "content-based retrieval (CBR)" use case typically involves a combination of both content-based search of document content and SQL-type search of record metadata. The abstract class ***com.ibm.jarm.api.query.RMContentSearchDefinition*** provides the static ***createInstance*** method that returns a concrete instance that is then used to define the CBR search process. This definition is then used by the ***RMSearch*** class to execute the CBR process.

```
RMDomain jarmDomain = ...;
FilePlanRepository fpRepository = ...;

String containerPath = "'/Records Management/File Plan/JLWTopRecCat'";
String recordMetaSelect = "SELECT r.Id, r.DocumentTitle";
String recordMetaFrom   = "FROM ElectronicRecordInfo r";
String recordMetaWhere  = "WHERE r.This INSUBFOLDER " + containerPath +
                          " AND r.IsDeleted = false ";
String cbrKeywords      = "brothers AND band";

// ---- Content-Based Retrieval Search ----
// Define the desired search criteria.
RMContentSearchDefinition searchDef =
  RMContentSearchDefinition.createInstance(jarmDomain);
// Indicate that this search involves record metadata
//  and content search portions.
searchDef.setCBRConditionOnly(false);
searchDef.setOperBtwContentAndMetadataSearch(AndOrOper.and);
searchDef.setOrderBy(OrderBy.cbrscores);
// Define the portion that concerns record metadata.
searchDef.setSelectClause(recordMetaSelect);
searchDef.setFromClause(recordMetaFrom);
searchDef.setWhereClause(recordMetaWhere);
searchDef.setSqlAlias("r");
// Define the CBR portion
searchDef.setContentSearch(cbrKeywords);
searchDef.setSortOrder(SortOrder.desc);
// Set up the search object - note that the specified
//  file plan repository automatically 'knows' which
//  content repository(s) it is associated with.
RMSearch jarmSearch = new RMSearch(fpRepository);
Integer pageSize = Integer.valueOf(10);
Boolean continuable = Boolean.TRUE; // Use CBR paging
// Finally execute the search process.
PageableSet<CBRResult> resultSet =
  jarmSearch.contentBasedRetrieval(searchDef, pageSize, continuable);

// Make use of special CBR paged iterator
CBRPageIterator<CBRResult> cbrPI =
  (CBRPageIterator<CBRResult>)(resultSet.pageIterator());
while ( cbrPI.hasNextPage() )
{
  List<CBRResult> currentPage = cbrPI.getNextPage();
  for (CBRResult aResult : currentPage)
  {
    RMProperties rowProps = aResult.getResultRow().getProperties();
    String recTitle = rowProps.getStringValue("DocumentTitle");
    double cbrScore = aResult.getScore();
    System.out.printf("Record '%s', score: %f%n",
           recTitle, cbrScore);
  }
}
```

3.9 Perform Bulk Operations

This example demonstrates use of the BulkOperation class to perform the straight forward task of closing multiple record containers in a single call:

```
FilePlanRepository fpRepository = ...;
List<String> containerIdList = new ArrayList<String>();
for (RecordContainer container : containers)
```

```
{
  containerIdList.add( container.getObjectIdentity() );
}

String auditReason = "For demo purposes...";
List<BulkItemResult> BIRs =
  BulkOperation.closeContainers(fpRepository, containerIdList, auditReason);

for (BulkItemResult bir : BIRs)
{
  System.out.printf("Close of container %s ", bir.getEntityIdent());
  if ( bir.wasSuccessful() )
  {
    System.out.printf("was successful!%n");
  }
  else
  {
    RMRuntimeException ex = bir.getException();
    System.out.printf("failed due to: %s%n", ex.getLocalizedMessage());
  }
}
```

The next example demonstrates placing a hold on multiple-***Holdable*** entities which requires use of the ***BulkOperation.EntityDescription*** class:

```
FilePlanRepository fpRepository = ...;
Hold holdToPlace                = ...;
RecordCategory holdTarget1      = ...;
Record holdTarget2              = ...;

List<EntityDescription> targetList = new ArrayList<EntityDescription>(2);
targetList.add(
  new EntityDescription(holdTarget1.getEntityType(),
                        holdTarget1.getObjectIdentity()));
targetList.add(
               new EntityDescription(holdTarget2.getEntityType(),
                                     holdTarget2.getObjectIdentity()));
List<String> holdIdents = new ArrayList<String>(1);
holdIdents.add(holdToPlace.getObjectIdentity());

List<BulkItemResult> BIRs =
  BulkOperation.placeHolds(fpRepository, targetList, holdIdents);
for (BulkItemResult bir : BIRs)
{
  System.out.printf("Place of hold on entity %s ", bir.getEntityIdent());
  if ( bir.wasSuccessful() )
  {
    System.out.printf("was successful!%n");
  }
  else
  {
    RMRuntimeException ex = bir.getException();
    System.out.printf("failed due to: %s%n", ex.getLocalizedMessage());
  }
}
```

3.10 Working with RMRuntimeException

The following code snippet demonstrates how to work with a caught ***RMRuntimeException*** including discovery of any available chained exception information:

```
catch (RMRuntimeException rre)
{
  System.out.printf("Standard exception msg: %s%n", rre.getLocalizedMessage());
  // More detailed info and suggested action:
  MessageInfo msgInfo = rre.getMessageInfo();
  System.out.printf("MessageInfo contents:%n");
  System.out.printf("  Formatted text: %s%n", msgInfo.getFormattedMessage());
  System.out.printf("  Explanation text: %s%n", msgInfo.getExplanation());
  System.out.printf("  Action text: %s%n", msgInfo.getAction());
  // Specific error code:
  RMErrorCode ec = rre.getErrorCode();
  System.out.printf("Error Code: %s%n", ec);

  // Examine any attached exception info...
  RMErrorStack stack = rre.getErrorStack();
  if (stack != null)
  {
    System.out.printf("Error Stack: %s%n", stack.toString());
    // OR dive into the error stack...
    RMErrorRecord[] errRecs = stack.getErrorRecords();
    for (int i = 0; i < errRecs.length; i++)
    {
      System.out.printf("Error record #%d: %s%n", i, errRecs[i].toString());
      // OR dive into each individual error record...
      System.out.printf("Error record #d desc: %s%n", i, errRecs[i].getDescription());
      System.out.printf("  Source: %s%n", errRecs[i].getSource());
      Diagnostic[] diags = errRecs[i].getDiagnosticTypes();
      for (int d = 0; d < diags.length; d++)
      {
        System.out.printf("    Diagnostic[%d]  type: %s%n", d, diags[d].getType());
        System.out.printf("  , Diagnostic[%d] value: %s%n", d, diags[d].getValue());
      }
    }
  }
}
```

3.11 Access available Users and Groups

Although the following example concerns finding particular users that meet the specified criteria, the same mechanism is also available for group discovery.

```
RMDomain jarmDomain = domain;

// Define user search criteria...
String pattern = "Jan";
RMPrincipalSearchType type = RMPrincipalSearchType.PrefixMatch;
RMPrincipalSearchAttribute attr = RMPrincipalSearchAttribute.DisplayName;
RMPrincipalSearchSortType sort  = RMPrincipalSearchSortType.Ascending;
Integer pageSize = Integer.valueOf(5);

PageableSet<RMUser> resultSet =
  jarmDomain.findUsers(pattern, type, attr, sort, pageSize);
```

```
// Note that PageableSet supports java.lang.Iterable.
for (RMUser matchingUser : resultSet)
{
  System.out.printf("Found user: %s%n", matchingUser.getDisplayName());
}


// Currently authenticated user
RMUser currentUser = jarmDomain.fetchCurrentUser();
```

3.12 Conversion between JACE and JARM types

Here are some examples of the available conversion utilities between various ***com.filenet.api.core.**** and ***com.ibm.jarm.api.**** types provided by the ***com.ibm.jarm.util.P8CE_Convert*** class:

```
Domain jaceDomain           = ...;
Connection jaceConnection   = ...;
ObjectStore jaceObjectStore = ...;
Folder jaceFolder           = ...;
Document jaceDoc            = ...;

// Connection level
DomainConnection jarmConnection = P8CE_Convert.fromP8CE(jaceConnection);
Connection jaceConnection2 = P8CE_Convert.fromJARM(jarmConnection);

// Domain level
RMDomain jarmDomain = P8CE_Convert.fromP8CE(jaceDomain);
Domain jaceDomain2  = P8CE_Convert.fromJARM(jarmDomain);

// ObjectStore <--> Repository
Repository jarmRepository = P8CE_Convert.fromP8CE(jaceObjectStore);
//  Or, if you know it is a FPOS...
FilePlanRepository jarmFPRepos =
  (FilePlanRepository)(P8CE_Convert.fromP8CE(jaceObjectStore));
ObjectStore jaceObjStore = P8CE_Convert.fromJARM(jarmRepository);

// Folder <--> Container
Container jarmContainer = P8CE_Convert.fromP8CE(jaceFolder);
//  Or, if you know it is a record category for example...
RecordCategory jarmRecCat =
  (RecordCategory)(P8CE_Convert.fromP8CE(jaceFolder));
Folder jaceFolder2 = P8CE_Convert.fromJARM(jarmRecCat);

// Document <--> ContentItem
ContentItem jarmContentItem = P8CE_Convert.fromP8CE(jaceDoc);
Document jaceDoc2 = P8CE_Convert.fromJARM(jarmContentItem);
```

4 Logging and Tracing

Logging support for JARM is based upon the widely used Apache log4j framework. Globalization support for JARM log strings is packaged in the deployed JarmResources.jar file. JARM supports the following log4j hierarchical log levels:

▪ OFF	Logging is disabled.
▪ ERROR	Used to report JARM exception occurrences.
▪ WARN	Used to report unusual circumstances that do not necessarily prevent proper functioning.
▪ INFO	Used to report less common activity (primarily disposition-related).

The following are the hierarchical log4j logger names used by JARM and which can be configured for using standard log4j configuration files:

com.ibm.jarm.api.Error	All JARM RMRuntimeException log reporting.
com.ibm.jarm.ral.*	Other non-error-related JARM implementation log reporting.

JARM tracing capability is also based upon the Apache log4j framework but is not globalized. It supports three hierarchical levels of tracing output:

▪ minimum	Minimal amount of detail is provided (e.g., method name but no parameter information).
▪ medium	Medium amount of detail (e.g., parameter type and size but not content).
▪ maximum	Maximum amount of detail (e.g., individual collection member information).

There is also an independent "timer" level that provides duration time reporting for various internal JARM operations (primarily those involving repository communications).

JARM Tracing can be controlled individually for each of the following JARM internal sub-systems:

api	Tracing for the public JARM entry points.
ralCommon	Tracing for internal JARM business logic independent of repository type.
ralP8CE	Tracing for the internal FileNet P8CE repository implementation support.
ralP8CE_CBR	Tracing specifically related to FileNet P8CE content-based-retrieval searches.

4.1 Sample log4j.xml Configuration File

The standard Enterprise Records JARM installation includes a sample XML-type log4j configuration file that can be customized to control both JARM logging and/or tracing.

This file defines several example log4j Appenders that control the output destination(s) of logging and tracing:

FileNetNullAppender	Messages are formatted but not written anywhere.
FileNetConsolAppender	Sends messages to the default log4j console.
FileNetErrorAppender	Error-level messages are sent to the "p8_jarm_error.log" file.
FileNetLogAppender	Other log messages are sent to the "p8_jarm_log.log" file.
FileNetTraceAppender	Traces log messages are sent to the "p8_jarm_trace.log" file.

(Note that there are also "rolling" versions of the error, log and trace appenders).

Each of these appenders is defined in their respective "<append>...</appender>" XML elements of the log4j.xml file.

The previously mentioned JARM hierarchical loggers are controlled by their respective "<logger>...</logger>" XML elements. The "<level>" sub-element can be set to one of the supported log4j levels: error, warn or info. One of more of the sample "<appender-ref>" sub-elements is then uncommented to allow related logger output to be sent to the appropriate appender(s).

JARM trace generation is always performed using the log4j "debug" level. The previously mentioned JARM tracing sub-systems are controlled by means of individual "<logger>" XML elements using log4j logger names that follow this format:

jarmTrace.<subsystem>.timer	Controls timer trace level.
jarmTrace.<subsystem>.maximum.medium.minimum	Controls minimum trace level.
jarmTrace.<subsystem>.maximum.medium	Controls medium trace level.
jarmTrace.<subsystem>.maximum	Controls maximum trace level.

So, for each applicable trace subsystem, the ".timer" and/or one (and only one) of the ".maximum", ".maximum.medium" or ".maximum.medium.minimum" <logger> XML elements should be uncommented within the log4j.xml file.

5 Using JARM in a P8 Content Manager Event Handler

Unlike the original RMAPI, JARM is readily available for use in the development of custom, IER-related FileNet Content Manager Event Handlers. The following IBM Information Center topic provides detailed information concerning FileNet Content Manager Event Handler/Action/Subscriptions (http://www.ibm.com/support/knowledgecenter/#!/SSNW2F_5.2.0/com.ibm.p8toc.doc/filenetcontentmanager_5.2.0.htm):

> Developing IBM FileNet P8 applications > Content Engine Java and .NET Developer's Guide > About Server Extensions > Events and Subscriptions

Since a FileNet Content Manager Event Handler runs in the context of the FileNet Content Engine itself, many dependencies that normally would be required in a standalone JARM-based application, are automatically available (namely those involving JACE). Also, no authentication is required. Therefore the minimum set of dependency JARs that should be included in the event handler Code Module are:

- Jarm.jar
- JarmResources.jar
- ierLogTrace.jar

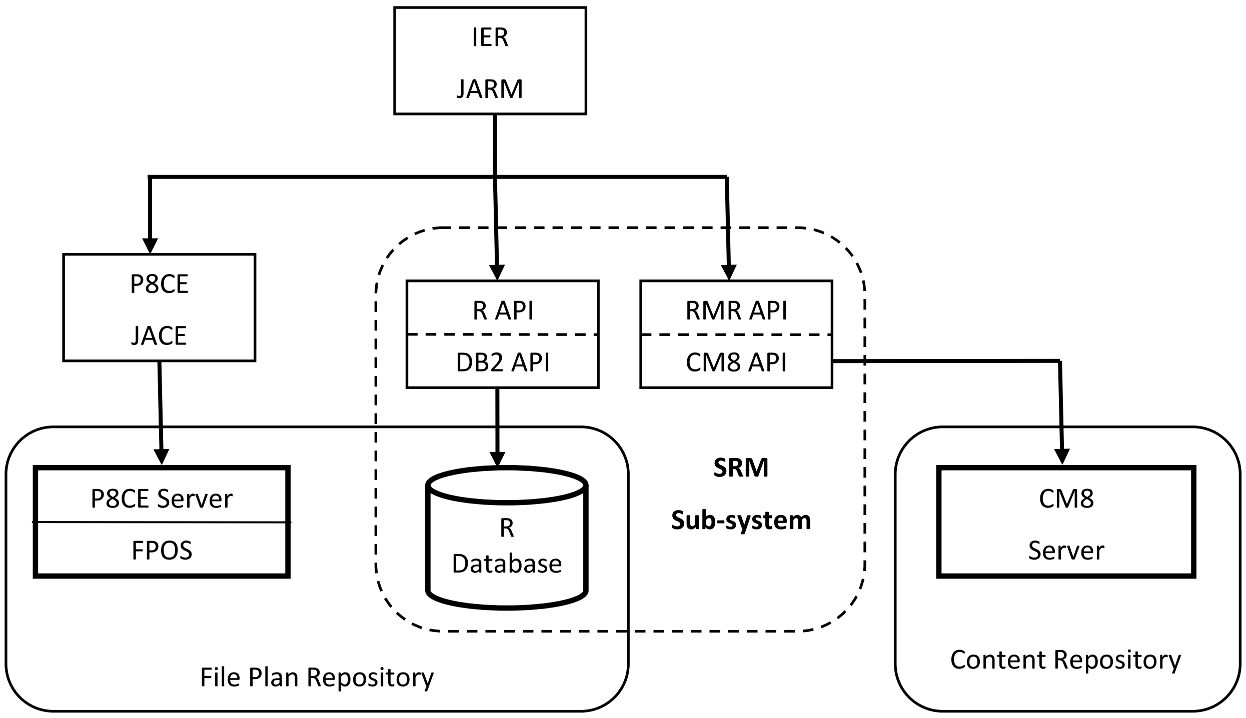
The following is a very simple example of an event handler that makes use of JARM. The example use case involves the need to update a particular dateTime property on the security folder belonging to a record whenever an update operation occurs on the record.

```
public class JarmExampleEventHandler implements EventActionHandler
{
  public void onEvent(ObjectChangeEvent event, Id subscriptionId)
                throws EngineRuntimeException
  {
    // Access JACE representation of the updated record object
    ObjectStore jaceObjStore = event.getObjectStore();
    Id sourceObjId = event.get_SourceObjectId();

    // Convert from JACE type to JARM type
    FilePlanRepository fpRepos =
      (FilePlanRepository)(P8CE_Convert.fromP8CE(jaceObjStore));

    // Fetch the record involved in the update process
    String recIdent = sourceObjId.toString();
    RMPropertyFilter filter = RMPropertyFilter.MinimumPropertySet;
    Record updatedRecord =
      RMFactory.Record.fetchInstance(fpRepos, recIdent, filter);

    // Determine its current security folder
    Container secFolder = updatedRecord.getSecurityFolder();

    // Perform an update to this folder and persist this change
    secFolder.getProperties()
             .putDateTimeValue("DateOfLastReview", new Date());
    secFolder.save(RMRefreshMode.NoRefresh);
  }
}
```

6 Support for IBM Content Manager V8

Starting with IBM Enterprise Records v5.2.0.1, JARM now includes support for working with records associated with content residing in an IBM Content Manager V8 repository. This section covers the differences in JARM related to this new feature.

The following diagram provides a high-level integration overview of the systems involved in this type of environment.



JARM Integration with IBM Content Manager V8

Some terminology for this diagram:

JARM	The IBM Enterprise Records "Java API for Records Management" and the primary topic of this document.
P8CE JACE	The IBM P8 "Java API for Content Engine" used to access the P8 domain and file plan object store.
P8CE Server/FPOS	Represents the IBM P8 Content Engine server that contains the file plan object store.
SRM Sub-system	Logical grouping of the pieces that provide record management and access to IBM Content Manager V8 content. "Scalable Records Management" (SRM) is an internal term for the new support for IBM Content Manager V8.
R API/DB2 API	API layers that provide record object persistence and management services.
R Database	A database used by R API for record object persistence.
RMR API/CM8 API	API layers that provide record-related services for IBM Content Manager V8 content.

6.1 Key Features of Enterprise Records for IBM Content Manager V8

- The file plan container hierarchy remains in the P8CE FPOS and continues to use the same security model at the container level.
- Records are not stored in the P8CE FPOS. They are persisted by means of the SRM sub-system in an external DB2 database. There is a record-to-file plan container relationship in place that is managed by JARM and SRM R APIs so the records are logically filed in FPOS file plan containers.
- Record declaration and disposition processing performance is improved over the existing P8CE/CFS-based solution.

6.2 Limitations of Enterprise Records for IBM Content Manager V8

- A single file plan repository can only be associated with a single SRM R database and a single IBM Content Manager V8 server.
- The file plan hierarchy only supports record category type containers. Record folder and volume containers are not supported.
- Multi-filing of records is not supported.
- Records can only be contained in leaf-type containers. This results in the following restrictions:
  - A container with one or more sub-containers cannot have any directly-contained records within it.
  - A container that does has one or more directly contained records cannot have any sub-containers.
- Individual record security is solely defined by the security defined for its file plan container.
- Hold placement is not supported at the record level. Holds are still supported at the file plan container level and do still apply to all child containers and records.
- A record class description does not support property value types of Object (neither single-valued nor multi-valued).
- Choice lists and marking sets are not supported as sources of record-level property values. They are still supported for file plan container property values.
- Naming patterns are not supported at the record-level.
- XML export is not supported at the record-level.
- Record copy is not supported.
- Record superseded is not supported.
- A link involving a record is not supported.
- Although file plan containers do support basic disposition schedules, they do not support advanced disposition schedules.

6.3 Installation

All of the additional components for JARM support for IBM Content Manager V8 are automatically installed as described in "Section 1.1 Installation".

6.4 Dependencies

The following table provides the list of dependent JAR (and other) files required by JARM when used to support IBM Content Management V8:

.jar File	Location	Description
Jarm.jar JarmResources.jar	…/EnterpriseRecords/API/JARM	(Required) Core JARM interfaces, implementation and globalized string resources.
ierLogTrace.jar	…/EnterpriseRecords/API/JARM	(Required) Enterprise Records logging and tracing support.
log4j-1.2.13.jar (or equivalent)	…/EnterpriseRecords/API/JARM	(Required) Log4j API used for logging and trace. (Note that the log4j.jar from JACE deployment can be used as well).
Jace.jar	…/EnterpriseRecords/ CommonFiles/CE_API/lib or …/EnterpriseRecords/API/JARM/ p8/CE_API/5.2	(Required) IBM FileNet P8 Java API for Content Engine. (See additional information concerning JACE below).
pe.jar peResources.jar	…/EnterpriseRecords/ CommonFiles or …/EnterpriseRecords/API/JARM/ p8/CE_API/5.2	(Optional) IBM FileNet P8 Process Engine Java API.
log4j.xml	…/EnterpriseRecords/API/JARM	(Optional) Sample JARM log4j configuration file to control JARM logging and tracing.
JSON4J.jar	…/EnterpriseRecords/API/JARM	(Required) Used for record auditing.
RecordClassUtil.jar servlet-api-2.5.jar	…/EnterpriseRecords/API/JARM	(Required) Supports record class metadata.
RMR.jar RMR_CM8.jar RMRUtil.jar	…/EnterpriseRecords/API/JARM	(Required) Provides record-related services for IBM Content Manager V8-based content.
SRM.jar SRM_sys.jar	…/EnterpriseRecords/API/JARM	(Required) Provides record object persistence and management services.
cmbicmsdk81.jar cmbutil81.jar cmbutilicm81.jar db2jcc_license_cisuz.jar db2jcc_license_cu.jar db2jcc4.jar log4j-1.2.15.jar	…/EnterpriseRecords/API/JARM/ cm8/lib	(Required) IBM Content Manager V8 Java API. (See additional information concerning IBM Content Manager V8 API below).
cmgmt & cmgmt/connectors directories	…/EnterpriseRecords/API/JARM/ cm8	(Required) CM8 Java API configuration files. (See additional information concerning IBM Content Manager V8 API below).

See Section 1.2 Dependencies for information concerning the P8CE JACE dependencies.

Use the following IBM Knowledge Center URL for information concerning acquiring the proper version of the IBM Content Manager V8 Java API and its proper configuration for your environment:

http://www-01.ibm.com/support/knowledgecenter/SSRS7Z_8.5.0/com.ibm.programmingcm.doc/dcmjv031.htm

6.5 Documentation

The JARM Javadocs referred to in "Section 1.3 Documentation" include all of the JARM features applicable to the support for IBM Content Manager V8.

6.6 General Concepts Specific to IBM Content Manager V8 Support

Here are some items to be aware of relating to JARM's IBM Content Manager V8 support:

- Unlike the UUID (or its string equivalent) that is used in P8CE to uniquely identify a record, an SRM record is identified by a unique integer value. This value can be determined using either of:
    - the ***Record.getObjectIdentity*** method (returns the string version of the numerical Id), or
    - from a record ***RMProperties*** collection defined by the symbolic name ***RMPropertyName.SRMRecordId*** constant.
- Each file plan repository container (for example, record category) residing on the FPOS has an equivalent entry in the SRM R database that is identified by a unique integer value. This integer value is stored on the FPOS container object as a single-valued Integer32 property whose symbolic name is defined by the ***RMPropertyName.RMSrmCID*** constant.

The following sub-sections describe changes to various JARM types for the SRM environment.

6.6.1 DomainConnection and Authentication

The existing ***RMFactory.DomainConnection.createInstance*** method is used to instantiate a new ***DomainConnection*** instance. In addition to the existing URL string needed to identify the P8CE domain, information must also be provided during the definition of the JARM ***DomainConnection*** for the following:

- The new ***DomainType.P8_SRM*** enum member must be used to indicate that a P8/SRM type of configuration is in use.
- The previously unused ***Map<String,Object> connectionInfo*** parameter is now expected to contain the following items:
    - An instance of ***com.ibm.jarm.api.core.SRM_R_ConnectionInfoEntry*** that defines the database connection criteria for the SRM R database. An instance of this type can be defined using either discrete database connection and credential criteria, or by providing a JNDI name to a pre-defined JEE data source. See the Javadocs for details.
    - An instance of ***com.ibm.jarm.api.core.SRM_RMR_ConnectionInfoEntry*** that defines the connection criteria for the IBM Content Manager V8 server. An instance of this type can be defined using either discrete IBM Content Manager V8 data store and credential criteria, or by means of an existing IBM Content Manager V8 API ***com.ibm.mm.sdk.server.DKDatastoreICM*** instance that represents an active, authenticated IBM Content Manager V8 connection. See the Javadocs for details.
    - An (optional) instance of ***com.ibm.jarm.api.SRM_Audit_ConnectionInfoEntry*** that defines connection and other criteria controlling record-level auditing. See the Javadocs for details.

See the "How to…" "Subsection 6.9.1 Connect to and authenticate with a P8_SRM domain" for a detailed code example describing the configuration of the JARM ***DomainConnection*** type.

As mentioned in "Section 2.3 DomainConnection and Authentication", a JAAS Subject must still be established using either the JACE ***UserContext*** and/or the JARM ***RMUserContext*** types. This action provides the necessary authentication for the P8CE domain portion of a P8/SRM JARM environment.

6.6.2 RMDomain and Repository

An ***RMDomain*** instance is still acquired using the existing ***RMFactory*** mechanism. ***RMDomain*** can still be used to determine the available ***FilePlanRepository*** and ***ContentRepository*** as defined by the corresponding ***DomainConnection*** instance.

Two new members have been defined for the existing ***RepositoryType*** enum:

- ***Content_RMR_CM8*** – indicates that a ***ContentRepository*** represents a IBM Content Manager V8 server accessed using the SRM RMR API layer.
- ***FilePlan_SRM*** – indicates that a ***FilePlanRepository*** associated with an SRM R database.

6.6.3 Record-level Metadata

Record class and property metadata are defined and persisted within the SRM sub-system. All other container and custom object metadata is defined and persisted in the P8CE FPOS. Record metadata is defined and managed using the IBM Content Navigator/Enterprise Records web application. The existing JARM ***RMClassDescription*** and ***RMPropertyDescription*** types can also be used to access SRM-type record metadata.

6.6.4 ContentItem

The differences between the SRM implementation of the JARM ContentItem type and that of the existing P8CE version include the following items:

- The following methods are not supported: ***delete, deleteAllVersions, exportAsXML, getContentElements***.
- The ***isEligibleForDeclaration*** method always returns true. If an attempt to declare a new record for a IBM Content Manager V8-based ***ContentItem*** that is actually ineligible, then an exception is thrown during the declare process.
- The new ***getCM8Pid*** method returns any available ***com.ibm.mm.sdk.common.DKPidICM*** instance for a IBM Content Manager V8-based ContentItem as an opaque ***java.lang.Object*** type.

The existing ***RMFactory.ContentItem*** factory supports defining a ***ContentItem*** instance for an existing IBM Content Manager V8 content entity by means of the new method:

getInstanceFromObject(ContentRepository repository, java.lang.Object opaqueObject)

where the ***opaqueObject*** parameter can be an instance of either a ***com.ibm.mm.sdk.common.DKDDO*** or ***com.ibm.mm.sdk.common.DKPidlICM***.

Note that neither of the existing ***RMFactory.ContentItem fetchInstance*** nor ***getInstance*** methods are supported for IBM Content Manager V8-based content.

6.6.5 Record Declaration

The JARM ***RecordContainer*** type single-record declaration methods provide support for IBM Content Manager V8-based content in the following manners:

- The ***declare(…, List<ContentItem> associatedContent)*** method is supported for IBM Content Manager V8-related ***ContentItem*** instances as decussed in the previous section.

- The ***declareObjects(…, List<java.lang.Object> opaqueContentObjs)*** method provides for direct declaration of content entities without defining them as ***ContentItem*** instances. The objects allowed in the ***opaqueContentObjs*** collection must be of the following types:
    - For ***DomainType.P8_CE***, these opaque objects must be instances of the JACE com.filenet.api.core.Document type.
    - For ***DomainType.P8_SRM***, these opaque objects must be instances of the IBM Content Manager V8 API ***com.ibm.mm.sdk.common.DKDDO*** or ***com.ibm.mm.sdk.common.DKPidICM*** types.

The existing JARM RecordContainer type also exposes the following new method that supports higher performance, bulk record declaration[1][2]:

BulkDeclareResults bulkDeclare(EnumSet<BulkDeclareOptions> declareOptions, String classIdent,
List<RecordDefinition> recordDefinitions,
RecordContainer… additionalContainers)

where:

declareOptions	a set of enumerated options that control the level of validation to perform.
classIdent	the symbolic name of the record class to base each record upon.
recordDefintions	a collection of ***RecordDefinition*** instances, each of which defines the metadata and associated content for a single new record.
additionalContainers	a collection of additional file plan containers into which each of the new records should be filed into. Currently not supported.
BulkDeclareResults	indicates the relative success or failure status of each of the attempted record declarations including any available exception information.

The new ***RecordDefinition*** type describes the characteristics of each of the new records to be declared, including:

- The RMProperties collection for the new record.
- The associated content entities that the record is to be declare for.

New instances of RecordDefinition are created using the new factory method:

RecordDefintion RMFactory.RecordDefintion.createInstance(DomainType domainType)

6.6.6 Record-level Security

Unlike its ***DomainType.P8_CE*** counterpart, a ***DomainType.P8_SRM*** based ***Record*** does not support customized ***RMPermission*** definitions. Instead the record exhibits the security assigned to the primary RecordContainer in which it logically resides.

[1] This new ***RecordContainer.bulkDeclare*** method is currently ONLY supported for ***DomainType.P8_SRM*** and not for ***DomainType.P8_CE***.

[2] Note that the existing IBM Enterprise Records Bulk Declare Services (BDS) API does not support declaration of IBM Content Manager V8. This new JARM ***RecordContainer.bulkDeclare*** method should be used instead.

6.6.7 Record Search

The JARM ***com.ibm.jarm.api.query.SRMSearch*** class provides search capability for ***DomainType.P8_SRM*** type records. This new search mechanism only support searches involving records and only provides for their retrieval as a collection of ***Record*** instances. The search specification is defined using the SRM Query Language syntax which is provided in "Appendix A SRM Query Language Syntax".

When a search involves file plan containers, the existing ***RMSearch*** class is still used along with the existing P8CE SQL syntax.

6.6.8 PageableSet Changes

The ***PageableSet<T>*** interface supports the new ***void close()*** method that should be called once use of the ***PageableSet*** and any associated ***iterator<T>*** or ***RMPageIterator<T>*** instance is no longer needed. This method frees any underlying resources associated with these iterators.

6.7 Record Auditing

Because records are now persisted outside of the P8CE-based file plan object store, the existing P8CE-based audit mechanism no longer applies to record entities. A dedicated table, "IER_AUDIT_EVENT", is defined in the R Database to store audit data for record-related actions. The supported actions include Declare, Delete, Relocate, Update and Undeclare. These are defined in JARM using the enum ***com.ibm.jarm.api.constants.AuditEventType***.You can specify a set of property symbolic names representing the record property values to capture in the audit event data for Update actions. For each action type, you can enable auditing for either or both success and failure actions.

Control over the level of auditing detail and the specific record-actions that are to be audited can be implemented using either of two mechanisms:

1. An ease-of-use mechanism is controlled by defining a SystemConfiguration instance stored on the P8CE file plan object store. This instance should have a ***PropertyName*** property value defined by the JARM constant ***SystemConfiguration.SRM_R_AUDIT_ENABLE*** with a corresponding ***PropertyValue*** value of "true" to enable record auditing. However, this mechanism only supports a fixed configuration:
    a. Only successful Declare, Delete, Relocate and Update actions are included.
    b. No property value capture is performed.
2. You can have programmatic control over auditing using the ***com.ibm.jarm.api.core.SRM_Audit_ConnectionInfoEntry*** class as part of the previously mentioned DomainConnection definition. This mechanism allows full control over the record auditing process. See the Javadocs for details.

Retrieval of available audit event data for a specific record is available using the ***Record.getSRMAuditedEvents*** method. This method returns a pageable collection of ***com.ibm.jarm.api.core.SRMAuditEvent*** instances. See the Javadocs for details of this type.

6.8 Changes to JARM Tracing

The following additional JARM trace-related log4j logger names are defined and included in the existing, out-of-the-box sample log4j.xml file. You can find this file in the ".../EnterpriseRecords/API/JARM" installation directory.

jarmTrace.ralSRM.declare.timer	- captures timing data for record declaration operations.
jarmTrace.ralSRM.delete.timer	- captures timing data for record delete operations.

jarmTrace.ralSRM.search.timer	- catpture timing data for record search operations.
jarmTrace.ralSRM.maximum.medium.minimum	- controls minimum level of trace for SRM sub-system.
jarmTrace.ralSRM.maximum.medium	- controls medium level of trace for SRM sub-system.
jarmTrace.ralSRM.maximum	- controls maximum level of trace for SRM sub-system.

6.9 How to…

The following subsections provide JARM example code snippets demonstrating use of the new JARM features included to support IBM Enterprise Records for IBM Content Manager V8.

6.9.1 Connect to and authenticate with a P8_SRM domain

The follow code snippet shows how the additional connection information for the SRM subsystem is incorporated when defining the JARM ***DomainConnection***.

```
// --------------------------------------------------------------
// ---- Define the connection information for the P8CE server
// --------------------------------------------------------------
String p8ceURL       = "http://<server>:<port>/wsi/FNCEWS40MTOM/";
String p8CEUsername = ...;
String p8CEPassword = ...;

String fposSymName   = "FPOS_symName";

// --------------------------------------------------------------
// ---- Define the connection information for the SRM R database
// --------------------------------------------------------------
// Alternative 1 - using JDBC connection data...
String srm_r_dbName      = "MyR_DBName";
String srm_r_dbHost      = "MyDB_host";
String srm_r_dbPort      = "50000";
String srm_r_dbUsername = ...;
String srm_r_dbPassword = ...;
SRM_R_ConnectionInfoEntry srm_r_CIEntry =
    new SRM_R_ConnectionInfoEntry(srm_r_dbName, srm_r_dbHost, srm_r_dbPort,
                                  srm_r_dbUsername, srm_r_dbPassword);
// Alternative 2 - using JEE JNDI data source path...
// String srm_r_jndiPath    = "SRM_R_JNDI_DS";
// SRM_R_ConnectionInfoEntry srm_r_CIEntry =
//    new SRM_R_ConnectionInfoEntry(srm_r_jndiPath);

// Store this SRM R CIEntry into a Map keyed by associated
//  file plan repository symbolic name.
Map<String,SRM_R_ConnectionInfoEntry> srm_r_CIEntryMap =
    new HashMap<String,SRM_R_ConnectionInfoEntry>();
srm_r_CIEntryMap.put(fposSymName, srm_r_CIEntry);


// --------------------------------------------------------------
// ---- Define the connection information for the SRM RMR/CM8 server
// --------------------------------------------------------------
// Alternative 1 = using CM8 Datastore connection data...
String srm_rmr_dsName       = "MyCM8_DSName";
String srm_rmr_dsUsername   = ...;
String srm_rmr_dsPassword   = ...;
SRM_RMR_ConnectionInfoEntry srm_rmr_CIEntry =
    new SRM_RMR_ConnectionInfoEntry(srm_rmr_dsName, srm_rmr_dsUsername,
                                    srm_rmr_dsPassword);
// Alternative 2 - using an existing CM8 API
//  com.ibm.mm.sdk.server.DKDatastoreICM instance...
// DKDatastoreICM cm8DSObj  = ...;
// SRM_RMR_ConnectionInfoEntry srm_rmr_CIEntry =
//    new SRM_RMR_ConnectionInfoEntry(cm8DSObj);


// Store this SRM RMR CIEntry into a Map keyed by the
//  CM8 datastore name.
Map<String,SRM_RMR_ConnectionInfoEntry> srm_rmr_CIEntryMap =
 new HashMap<String,SRM_RMR_ConnectionInfoEntry>();
srm_rmr_CIEntryMap.put(srm_rmr_dsName, srm_rmr_CIEntry);


//----------------------------------------------------------------
//---- Use all three connection information entries
//----  to create the JARM DomainConnection
//----------------------------------------------------------------
// Define the DomainConnection's "connectionInfo" map...
Map<String,Object> jarmConnInfo = new HashMap<String,Object>();
jarmConnInfo.put(DomainConnection.KEY_SRM_R_MAP,   srm_r_CIEntryMap);
jarmConnInfo.put(DomainConnection.KEY_SRM_RMR_MAP, srm_rmr_CIEntryMap);

DomainConnection jarmDomainConn =
    RMFactory.DomainConnection.createInstance(DomainType.P8_SRM,
                                              p8ceURL,
                                              jarmConnInfo);

//----------------------------------------------------------------
//---- Establish a JAAS Subject for P8CE authentication purposes...
//----------------------------------------------------------------
javax.security.auth.Subject jaasSubject =
    RMUserContext.createSubject(jarmDomainConn, p8CEUsername, p8CEPassword,
                                RMUserContext.P8_STANZA_WSI);
RMUserContext.get().setSubject(jaasSubject);


//----------------------------------------------------------------
//---- Acquire the JARM RMDomain instance...
//----------------------------------------------------------------
RMDomain p8SRMDomain =
    RMFactory.RMDomain.fetchInstance(jarmDomainConn, null,
                                     RMPropertyFilter.MinimumPropertySet);


//----------------------------------------------------------------
//---- Acquire the File Plan and Content Repositories...
//----------------------------------------------------------------
FilePlanRepository fpRepos =
    RMFactory.FilePlanRepository.fetchInstance(p8SRMDomain, fposSymName,
                                               RMPropertyFilter.MinimumPropertySet);
```

```
ContentRepository cm8ContentRepos =
    RMFactory.ContentRepository.fetchInstance(p8SRMDomain, srm_rmr_dsName,
                                              RMPropertyFilter.MinimumPropertySet);
```

6.9.2 Declare a single record for IBM Content Manager V8 content

The following example demonstrates how to declare a new record for an existing IBM Content Manager V8 content entity. Note that it is out of the scope of this document as to how a IBM Content Manager V8-based custom application makes use of the IBM Content Manager V8 Java API to acquire an instance of ***com.ibm.mm.sdk.DKDDO*** used in the example.

```
RMDomain p8SRMDomain                = ...
FilePlanRepository fpRepository     = ...
ContentRepository contentRepository = ...

// The CM8 Java API representation of the CM8 content to declare.
//  Alternatively, could use the CM8 API com.ibm.mm.sdk.common.DKPidICM type instead.
com.ibm.mm.sdk.common.DKDDO cm8DKDDO = ...
List<Object> opaqueContentObjs       = new ArrayList<Object>();
opaqueContentObjs.add(cm8DKDDO);

RecordContainer primaryContainer =
    RMFactory.RecordCategory.fetchInstance(fpRepository, RCPath, null);
List<RecordContainer> addlContainers = null;  // Not supported for P8_SRM.
String recordClassIdent              = RMClassName.ElectronicRecord;
List<RMPermission> addlRecordPerms   = null;  // Not supported for P8_SRM

// Define the properties collection for the new record...
//  Note the need to specify DomainType.P8_SRM AND the entity type to whom
//  this properties collection will apply to.
RMProperties jarmProps =
    RMFactory.RMProperties.createInstance(DomainType.P8_SRM, EntityType.Record);
jarmProps.putStringValue(RMPropertyName.DocumentTitle, "ERec For CM8 content");

// Ready to declare the new record...
Record newRecord = primaryContainer.declareObjects(recordClassIdent,
                                                   jarmProps,
                                                   addlRecordPerms,
                                                   addlContainers,
                                                   contentRepository,
                                                   opaqueContentObjs);

System.out.printf("Successfully declared new electronic record, %s, " +
                  " for CM8 document: %s%n",
                  newRecord.getObjectIdentity(),
                  cm8DKDDO.getPidObject().toString());
```

6.9.3 Bulk Declare Multiple IBM Content Manager V8 Contents as Individual Records

The following example demonstrates how to declare individual records for multiple IBM Content Manager V8 content entities in a single bulk operation.

```
RMDomain p8SRMDomain                  = ...
FilePlanRepository fpRepository       = ...
ContentRepository contentRepository = ...

// Following is an array of CM8 API representations of the content entities
//  for which individual records are to be declared within a single
//  bulk declare operation.
com.ibm.mm.sdk.common.DKDDO[] cm8ContentEntities = ...

RecordDefinition[] recordDefs = new RecordDefinition[cm8ContentEntities.length];
// For each such content entity, need to define a RecordDefinition instance...
for( int i = 0; i < cm8ContentEntities.length; i++ )
{
    RecordDefinition newRecDef =
          RMFactory.RecordDefinition.createInstance(DomainType.P8_SRM);

    // Define the property collection for the new record...
    RMProperties jarmProps =
        RMFactory.RMProperties.createInstance(DomainType.P8_SRM, EntityType.Record);
    jarmProps.putStringValue(RMPropertyName.DocumentTitle, "ERecForCM8Doc_" + i);
    newRecDef.setProperties(jarmProps);

    // Define the content entity for this new record
    newRecDef.setAssociatedContentObjects(contentRepository, cm8ContentEntities[i]);

    recordDefs[i] = newRecDef;
}
List<RecordDefinition> recordDefList = Arrays.asList(recordDefs);

// The file plan container for all of the new records.
RecordContainer primaryContainer =
     RMFactory.RecordCategory.fetchInstance(fpRepository, RCPath, null);

// Define options relating to record declaration validation.
EnumSet<BulkDeclareOptions> declareOptions = EnumSet.allOf(BulkDeclareOptions.class);

// Symbolic name of record class to base each new record on.
String recordClassIdent = RMClassName.ElectronicRecord;

// (Additional record containers are not supported for P8_SRM.)
RecordContainer[] addlContainers = null;

// Ready to declare the entire set of records...
BulkDeclareResults results = primaryContainer.bulkDeclare(declareOptions,
                                                          recordClassIdent,
                                                          recordDefList,
                                                          addlContainers);
```

```
// Check success/failure status of each attempted record...
Exception[] failures = results.getFailures();
for (int i = 0; i < failures.length; i++)
{
    if ( failures[i] == null )
    {
        String newRecIdent = recordDefs[i].getId();
        System.out.printf("Declaration of record #%d succeeded, recIdent: %s%n",
                          i, newRecIdent);
    }
    else
    {
        System.out.printf("Declaration of record #%d failed: %s%n", i,
                          failures[i].getLocalizedMessage());
    }
}
```

Appendix A SRM Query Language Syntax

This appendix provides the grammer for the Query Language used by the ***com.ibm.jarm.api.query.SRMSearch*** class.

```
GET [ TOP n ] RECORDS
      [ USING <synonym-list> ]
      PROJECT <projection-list>
             FILTER <predicates-list>
      [ ORDER <order-list> ]  ;

    <projection-list> :=  <type.*>  [ , <type.*>  ]  | <type>.<field> [ ,
<type>.<field>  ]
    <predicate-list> := <predicate> [  AND | OR  <predicate> ] ...
    <predicate> := <dbpredicate>
    <predicate> := <rmrpredicate>
    <predicate> := <containmnetpredicate>
    <dbpredicate> :=  <type>.<field> <dboperator> <literal>
    <containmnetpredicate> :=  IN | NOT IN  [DEEP] FOLDER ( <containment> [ ,
<containment> ] ) ..
    <containmnetpredicate> :=  IN | NOT IN  REPORT ( <containment> [ ,
<containment> ] ) ..
    <containmnetpredicate> :=  ON | NOT ON  HOLD [ ( <containment> [ ,
<containment> ] ) ]  ..
    <containment> := " <id> "

    <rmrpredicate> := CONTAINS ( "<text index search query string>"  )

    <order-list> := ORDER <type>.<field> [ASC|DSC] [ , <type>.<field> [ASC|DSC] ]

    <text index search query string> := <whatever is the CBR supported string/no
deep parse>

    <type>:=  A type in the Type System ( aka. RecordClass ) , This can be the
super type RECORD
    <field>:= A field in the Type System. A <field> must be fully qualified by
its <type>.
          A field can be a built in field of the super class RECORD. Please see
BuiltIns List.

    <dboperator> := ">" | "<" | "=" | "<=" | ">=" | LIKE| IN | IS NULL | IS NOT
NULL

    <synonym-list> := <macro> = <symbol> [ , <macro> = <symbol> ]

    <symbol> := <typename> | <fieldname>

    <macro> := <string>

    <string> := string

    <id> := integer
    A logical super type called "Record" is builtin and understood (super class
Record)

    A "*" implies all Fields of <type>

    Key words :         GET
                RECORDS
                USING   >>> Synonym List Specification
                PROJECT >>> Projection List Specification
                FILTER  >>> Predicate List Sepcification
                TOP
                ORDER   >>> Sort Order Specification
                RANK
                ASC
                DSC
                IN
                ON
                FOLDER
                REPORT
                HOLD
                RECORD >>> Super Class Record
                *
                    IS
                NOT
                NULL
                )      >>> Open Bracket
                (      >>> Close Bracket

    PRECEDENCE Rules Apply With Bracket as Highest Precedence. If Not brackets ,
Predicate Legs will be combined to
    3 classes : RMR , Containment and DB in occuring Order. Brackets are Also
Used For Segmentation.
```

Notices

Trademarks