Apache UIMA™

Apache UIMA™ Development Community

Version 3.2.0

UIMA 3 User's Guide

The document is a manual for users of Apache UIMA, specifically focussing on the new features introduced in version 3.

License and Disclaimer

Trademarks

Overview of UIMA Version 3

UIMA Version 3 adds significant new functionality for the Java SDK, while remaining backward compatible with Version 2. Much of this new function is enabled by a shift in the internal details of how Feature Structures are represented. In Version 3, these are represented internally as ordinary Java objects, and subject to garbage collection.

In contrast, version 2 stored Feature Structure data in special internal arrays of `ints` and other data types. Any Java object representation of Feature Structures in version 2 was merely forwarding references to these internal data representations.

If JCas is being used in an application, the JCas classes must be migrated, but this can often be done automatically. In Version 3, the JCas classes ending in "_Type" are no longer used, and the main JCas class definitions are much simplified.

If an application doesn't use JCas classes, then nothing need be done for migration. Otherwise, the JCas classes can be migrated in several ways:

generating during build

If the project is built by Maven, it's possible the JCas classes are built from the type descriptions, using UIMA's Maven JCasGen plugin. If so, you can just rebuild the project; the JCasGen plugin for V3 generates the new JCas classes.

running the migration utility

This is the recommended way if you can't regenerate the classes from the type descriptions.

This does the work of migrating and produces new versions of the JCas classes, which need to replace the existing ones. It allows complex existing JCas classes to migrated, perhaps with developer assistance as needed. Once done, the application has no migration startup cost.

The migration tool is capable of using existing source or compiled JCas classes as input, and can migrate classes contained within Jars or PEARs.

regenerating the JCas classes using the JCasGen tool

The JCasGen tool (available as a Eclipse or Maven plugin, or a stand-alone application) generates Version 3 JCas classes from the XML descriptors.

This is perfectly adequate for migrating non-customized JCas classes. When run from the UIMA Eclipse plugin for editing XML component descriptors, it will attempt to merge customizations with generated code. However, its approach is not as comprehensive as the migration tool, which parses the Java source code.

Migration of JCas classes is the first step needed to start using UIMA version 3. See the later chapter on migration for details on using the migration tool.

What's new in UIMA 3

The major improvements in version 3 include:

Support for arbitrary Java objects, transportable in the CAS

Support is added to allow users to define additional UIMA Types whose JCas implementation may include Java objects, with serialization and deserialization performed using normal CAS transportable data. A following chapter on Custom Java Objects describes this new facility.

New UIMA semi-built-in types, built using the custom Java object support

The new support that allows custom serialization of arbitrary Java objects so they can be transported in the CAS (above) is used to implement several new semi-built-in UIMA types.

FSArrayList

a Java ArrayList of Feature Structures. The JCas class implements the List API.

IntegerArrayList

a variable length int array. Supports OfInt iterators.

FSHashSet, FSLinkedHashSet

a Java HashSet or LinkedHashSet containing Feature Structures. This JCas class implements the Set API.

Select framework for accessing Feature Structures

A new *select framework* provides a concise way to work with Feature Structure data stored in the CAS or other collections. It is integrated with the Java 8 *stream* framework, while providing additional capabilities supported by UIMA, such as the ability to move both forwards and backwards while iterating, moving to specific positions, and doing various kinds of specialized Annotation selection such as working with Annotations spanned by another annotation.

By default, when sorted iterators are set up by the select framework, they ignore typePriorities; this addresses a need of many use cases, and makes operation when there are many annotations spanning the same begin and end more reliable. Each select can specify to use typePriority as part of the ordering when required.

This user's guide has a chapter devoted to this new framework.

Elimination of ConcurrentModificationException while iterating over UIMA indexes

The index and iteration mechanisms are improved; it is now allowed to modify the indexes while iterating over them (the iteration will be unaffected by the modification).

Note that the automatic index corruption avoidance introduced in more recent versions of UIMA could be automatically removing Feature Structures from indexes and adding them back, if the user was updating some Feature of a Feature Structure that was part of an index specification for inclusion or ordering purposes.

In version 2, you would accomplish this using a two pass scheme: Pass 1 would iterate and merely collect the Feature Structures to be updated into a Java collection of some kind. Pass 2 would use a plain Java iterator over that collection and modify the Feature Structures and/or the UIMA indexes. This is no longer needed in version 3; UIMA iterators use a copy-on-write technique to allow index updating, while doing whatever minimal copying is needed to continue iteration over the original index.

In both version 2 and 3, there are 3 iterator movement APIs which have a side effect of insuring the iterator is operating correctly over the current index contents. These are the `moveToFirst, moveToLast, and moveTo(some_feature_structure)` API calls. In version 3, using these will reinitialize the iterator (if needed) so that it is iterating over the current index contents; if the index has not been modified, no reinitialization is needed (or done).

CAS reset and index removeAll operations clear the index without preserving any existing iteration. If you try to continue an iteration over an index cleared by these operations, the results are undefined, and may throw exceptions.

Logging updated

The UIMA logger is a facade that can be hooked up at deploy time to one of several logging backends. It has been extended to implement all of the Logger API calls provided in the SLF4j `Logger` interface, and has been changed to use SLF4j as its back-end. SLF4j, in turn, requires a logging back-end which it determines by examining what's available in the classpath, at deploy time. This design allows UIMA to be more easily embedded in other systems which have their own logging frameworks.

Modern loggers support MDC/NDC and Markers; these are supported now via the slf4j facade. UIMA itself is extended to use these to provide contexts around logging.

See the following chapter on logging for details.

Automatic garbage collection of unreferenced Feature Structures

This allows creating of temporary Feature Structures, and automatically reclaiming space resources when they are no longer needed. In version 2, space was reclaimed only when a CAS was reset at the end of processing.

better performance

The internal design details have been extensively reworked to align with recent trends in computer hardware over the last 10-15 years. In particular, space and time tradeoffs are adjusted in favor of using more memory for better locality-of-reference, which improves performance. In addition, the many internal algorithms (such as managing Feature Structure indexes) have been improved.

Type system implementations are reused where possible, reducing the footprint in many scaled-out cases.

Backwards compatible

Version 3 is intended to be binary backwards compatible - the goal is that you should be able to run existing applications without recompiling them, except for the need to migrate or regenerate any User supplied JCas Classes. Utilities are provided to help do the necessary JCas migration mostly automatically.

Integration with Java 8

Version 3 requires Java 8 as the minimum level. Some of version 3's new facilities, such as the `select` framework for accessing Feature Structures from CASs or other collections, integrate with the new Java 8 language constructs, such as `Streams` and `Spliterators`.

Programming convenience

Many APIs have been made more consistent and better integrated; see the chapter on new and extended APIs. Examples: UIMA Indexes now implement Iterable, so you can use the Java "extended for" construct directly; UIMA Lists have new push and pushNode methods to create and link a new node onto the front of a list; there are new methods on the CAS and JCas to get a shared instance of common immutable objects, like 0-length arrays and empty lists.

Just to give a small taste of the kinds of things Java 8 integration provides, here's an example of using the new `select` framework, where the task is to compute a Set of all the found types

- in a UIMA index
- under some top-most type "MyType"
- occurring as Annotations within a particular bounding Annotation
- that are nonOverlapping

Here is the Java code using the new `select` framework together with Java 8 streaming functions:

```
Set<Type> foundTypes =
   myIndex.select(MyType.class)
   .coveredBy(myBoundingAnnotation)
   .nonOverlapping()
   .map(fs -> fs.getType())
   .collect(Collectors.toCollection(TreeSet::new));
```

Another example: to collect, by category, the average length of the annotations having that category. Here we assume that `MyType` is an `Annotation` and that it has a feature called `category` which returns a String denoting the category:

```
Map<String, Double> freqByCategory =
   myIndex.select(MyType.class)
   .collect(Collectors
     .groupingBy(MyType::getCategory,
                 Collectors.averagingDouble(f ->
                   (double)(f.getEnd() - f.getBegin()))));
```

Java 8 is required

The UIMA Java SDK Version 3 requires Java 8.

Backwards Compatibility

Because users have made substantial investment in developing applications using the UIMA framework, a goal of version 3 is to protect this investment, by enabling Annotators and applications developed under previous versions to be able to be used in subsequent versions of the framework.

To this end, version 3 is designed to be backwards compatible, except for needing:

- possibly a recompilation (due to some rearrangements of many classes and interfaces)
- a new set of User-defined JCas classes (if these were previously being used). The creation of these CAS classes can be done by regenerating them using JCasGen, or by using a migration tool that handles converting the existing JCas classes. A later chapter covers how to upgrade the JCas classes.

There are some additional exceptions, described in the following sections.

JCas and non-JCas APIs

The JCas class changes include no longer needing or using the `Xyz_Type` sister classes for each main JCas class. User code is unlikely to access these sister classes. The JCas API method to access this sister class now throws a UnsupportedOperation exception.

The non-JCas Java cover classes for the built-in UIMA types remain, for backwards compatibility. So, if you have code that casts a Feature Structure instance to AnnotationImpl (a now deprecated version 2 non-JCas Java cover class), that will continue to work.

Additional reserved names in the JCas generated classes

Names beginning with "_" (underscore) are being used by the new JCas implementation, so you should not name things with this convention. If you do, please insure your names are not colliding with the names being used by the generated JCas files.

Serialization forms

The backwards compatibility extends to the serialized forms, so that it should be possible to have a UIMA-AS services working with a client, where the client is a version 3 instance, but the server is still a version 2 (or vice versa).

Delta CAS Version 2 Binary deserialization not supported

The binary serialization forms, including Compressed Binary Form 4, build an internal model of the v2 CAS in order to be able to deserialize v2 generated versions. For **delta** CAS, this model cannot be accurately built, because version 3 excludes from the model all unreachable Feature Structures, so in most cases it won't match the version 2 layout.

Version 3 will throw an exception if delta CAS deserialization of a version 2 binary delta CAS is

attempted.

APIs for creating and modifying Feature Structures

There are 3 sets of APIs for creating and modifying Feature Structures; all are supported in V3.

- Using the JCas classes
- Using the normal CAS interface with Type and Feature objects
- Using the low level CAS interface with int codes for Types and Features

Version 3 retains all 3 sets, to enable backward compatibility.

The low level CAS interface was originally provided to enable a extra-high-performance (but without compile-time type safety checks) mode. In Version 3, this mode is actually somewhat slower than the others, and no longer has any advantages.

Using the low level CAS interface also sometimes blocks one of the new features of Version 3 - namely, automatic garbage collection of unreachable Feature Structures. This is because creating a Feature Structure using the low level API creates the Java object for that Feature Structure, but returns an "int" handle to it. In order to be able to find the Feature Structure, given that int handle, an entry is made in an internal map. This map holds a reference to this Feature Structure, which prevents it from being garbage collected (until of course, the CAS is reset).

The normal CAS APIs allow writing Annotators where the type system is unknown at compile time; these are fully supported.

Preserving V2 ids, with low level CAS Api accessibility

Some V2 applications make use of the Feature Structure address, using these as an integer identifier and using the low level CAS APIs to access the Feature Structure, given this integer. These applications also often use the stability of these ids across some serialization/deserializations.

Normally in V3, deserialization of CASs having these IDs occurs without preserving the IDs, and without setting up the low level CAS APIs to be able to access these using them. If an existing application depends on the low level access via the address, a special mode, called `V2IdRefs`, can be specified, which will support this. It comes at a cost however, which is that all new Feature Structures created (or deserialized) will be added to an internal table to enable the low level CAS `getFSForRef(int)` method to work. As a result, these Feature Structures are not eligible for garbage collection.

This mode is set on individual CASs via a new API; a default value may optionally be specified. Once set on a CAS, it remains until set to a different value; CAS Reset does not affect the setting, nor does checking it into / out of a CAS Pool.

When a new CAS is created, this mode is set according to two sources:

- a `-Duima.default_v2_id_references` system property, read once when the UIMA framework classes are loaded.

- A run-time value kept per thread, managed by an API on the LowLevelCAS interface. The setting is inherited by any child threads the thread creates, and overrides the system property if used.
- If neither of these are used, then the default is to not be in the sepcial v2-mode.

The APIs for this are part of the LowLevelCAS. The controlling APIs all return an instance of `AutoClosableNoException`, which can be used to reset the setting to its previous value. A recommended way of using these is with the Java `try with resources` construct:

```
try (AutoClosableNoException w = llcas.ll_enableV2IdRefs) {
  ... some operations
} // automatically restores previous value
```

LowLevelCas instance APIs for enabling/disabling this mode on a particular CAS:

```
// set the mode
AutoClosable ll_enableV2IdRefs()

// same, but with explicit set or reset of the mode
AutoClosableNoException ll_enableV2IdRefs(true/false)

// return true if the mode is enabled
boolean is_ll_enableV2IdRefs()
```

Static LowLevelCas APIs for setting the default value for this mode for new CASs on a particular thread:

```
// set the default
AutoClosableNoException LowLeveCas.ll_defaultV2IdRefs()

// same, but with explicit set or reset of the mode
AutoClosableNoException LowLeveCas.ll_defaultV2IdRefs(true/false)

// return true if the mode is enabled
boolean LowLeveCas.is_ll_defaultV2IdRefs()
```

This mode modifies multiple things in the operation of UIMA V3.

- Newly created Feature structures have IDs which match what UIMA V2 references (the "addresses") would be. For serialized forms (except Xmi), these IDs match the (imputed) v2 IDs of the serialized form.

  Newly created Feature Structures, including those created when deserializing, are added to an internal map which maps the ID to the Feature Structure instance. Feature Structures may be located by ID using the LowLevelCAS API `getFSForRef().`

  In order for this to work correctly, the mode must be set while the CAS is empty. If the mode is attempted to be set on a non-empty CAS, an IllegalStateException is thrown.

- This mode modifies serialization (except for XCas, XMI, and Compressed form 6, which in V2 are implemented to just serialize reachable Feature Structures) to include non-reachable FSs.
- Note: This does not affect the `select` framework results - unreachable Feature Structures are not included.

PEAR support

Pears are supported in Version 3. If they use JCas, their JCas classes need to be migrated.

When a PEAR contains a JCas class definition different from the surrounding non-PEAR context, each Feature Structure instance within that PEAR has a lazily-created "dual" representation using the PEAR's JCas class definition. The UIMA framework things storing references to Feature Structures are modified to store the non-PEAR version of the Feature Structure, but to return (when in a particular PEAR component in the pipeline) the dual version. The intent is that this be "invisible" to the PEAR's annotators. Both of these representations share the same underlying CAS data, so modifications to one are seen in the other.

If a user builds code that holds onto Feature Structure references, outside of annotators (e.g., as a shared External Resource), and sets and references these from both outside and inside one (or more) PEARs, they should adopt a strategy of storing the non-PEAR form. To get the non-PEAR form from a Feature Structure, use the method `myFeatureStructure._maybeGetBaseForPearFs()`.

Similarly, if code running in an Annotator within a PEAR wants to work with a Feature Structure extracted from non-UIMA managed data outside of annotators (e.g., such as a shared External Resource) where the form stored is the non-PEAR form, you can convert to the PEAR form using the method `myFeatureStructure.__maybeGetPearFs()`. This method checks to see if the processing context of the pipeline is currently within a PEAR, and if that PEAR has a different definition for that JCas class, and if so, it returns that version of the Feature Structure.

The new Java Object support does not support multiple, different JCas class definitions for the same UIMA Type, inside and outside of the PEAR context. If this is detected, a runtime exception is thrown.

The workaround for this is to manually merge any JCas class definitions for the same class.

toString()

The formatting of various UIMA artifacts, including Feature Structures, has changed somewhat, to be more informative. This may impact situations such as testing, where the exact string representations are being compared.

A special global Java property, -Duima.v2_pretty_print_format can be set to have the toString() operation for Feature Sructures print in the V2 style.

Logging configuration is somewhat different

The default logging configuration in v2 was to use Java Util Logging (the logger built into Java). For v3, the default is to use SLF4J which, in turn, picks a back-end logger, depending on what it finds in

the class path.

This change was done to permit easier integration of UIMA as a library running within other frameworks.

V3 UIMA logger includes the APIs like info(..), warn(..) etc., that are part of the SLF4j APIs. In addition, these are augmented with the Java 8 style lambda arguments that were introduced in log4j-2, for more concise and efficient log message computation.

The new UIMA Logger APIs (e.g. logger.info(...), logger.warn(...)) use the SLF4j and other modern logger substitutable notation of "{}", as opposed to the style adopted by the original Java logger, of "{nnn}". All modern loggers have switched to this.

The technique for (optionally) reporting the class and method (and sometimes, line number) was changed to conform to current logger conventions - whereby the loggers themselves obtain this information from the call stack. The V2 calls which pass in the sourceClass and sourceMethod information have this information ignored, but replaced with what the loggers obtain from the stack track. In some cases, where the callers in V2 were not actually passing in the correct class/method information, this will result in a different log record.

For more details, please see the logging chapter.

Type System sharing

Type System definitions are shared when they are equal. After type systems have been built up from type definitions, at "commit" time, a check is made to see if an identical type system already exists (same types and features). This is often the case when a UIMA application is scaling up by adding multiple pipelines, all using the same type system.

If an identical committed type system already exists, then the commit operation returns it, and the one just built is discarded. Normally, this is not an issue. However, some application code may save references to the type system object or to defined types and features. These references end up pointing to the discarded version, when the commit operation finds an already committed equal version.

Application code may code around this by re-acquiring references to the type system object, and to any type and feature objects, if the type system instance object returned from `commit` is not identical (==) to the one being committed. The type system commit APIs are changed to return the type system - either the one being committed, or an already existing equal committed type system. So when coding `myTypesystem.commit();` if you later refer to `myTypesystem`, change this to `myTypesystem = myTypesystem.commit();`, to keep the variable `myTypesystem` always referring to to the committed type system.

Some checks moved to native Java

In the interest of performance, some duplicate checks, such as whether an array index is within bounds, have been removed from UIMA when they are already being checked by the underlying Java runtime. This has affected some of the internal APIs, such as the JCas's `checkArrayBounds` which was removed because it was no longer being used.

Some class hierarchies have been modified

The various JCas Classes implementing the built-ins for arrays have some additional interfaces added, grouping them into `CommonPrimitiveArray` or `CommonArray`. These changes are internal, and should not affect users.

Enabling multiple versions of type systems to work with a single common JCas class

Some applications may use a JCas class definition, defining for type T features f1, f2, f3 (for example), in a mode where under a single class loader (for example, in one Java application), multiple CASs are loaded and processed, where each CAS might have other versions of the type system, defining for type T a subset of the features in the JCas.

In order to make this scenario possible, v3 takes an extra step, right before type system commit time, of loading the JCas classes corresponding to the types, and then augmenting the type definitions with additional features defined in the JCas but not in the type description. After this is done, the type system is committed, and offsets are assigned to the JCas class that are constant, even when a subsequent type system is loaded that defines more features (provided that no new features are introduced).

This feature represents a trade-off between highly efficient, locked-down offsets for features, and some limited flexibility to handle a somewhat common use case where additional features exist in the JCas. The JCas loading code always checks to insure compatibility between the offsets in the JCas classes, as first set up, and any subsequent type system being used with that JCas.

This accommodation doesn't handle many possible scenarios. Some of these include situations where a supertype might subsequently add extra feature slots, or the features end up after merging to have a different ordering.

For cases where this accommodation is insufficient, the workaround is to run separate UIMA applications, each under its own class loader, for the incompatible situations.

PEARs, because they are loaded lazily after the type system has been committed, do not support this kind of augmentation of types from the Pear-specific JCas class definition.

New and Extended APIs

UIMA FSIndex and FSIterators improvements

The FSIndex interface implements Collection, so you can now write `for (MyType item : myIndex)` to iterate over an index.

Because it implements Collection, the FSIndex interface includes a `stream()` method, so you can now write `myIndex.stream().any-stream-operations`, which will use the items in the index as the source of the stream.

The FSIterator interface now implements the Java ListIterator Interface, and supports the methods there except for add, nextIndex, previousIndex, and set; the remove() method's meaning is changed to remove the item from all of the UIMA indexes.

The iterators over indexes no longer throw concurrent modification exceptions if the index is modified while it is being iterated over. Instead, the iterators use a lazily-created copy-on-write approach that, when some portion of the index is updated, prior to the update, copies the original state of that portion, and continues to iterate over that. While this is helpful if you are explicitly modifying the indexes in a loop, it can be especially helpful when modifying Feature Structures as you iterate, because the UIMA support for detecting and avoiding possible index corruption if you modify some feature being used by some index as a key, is automatically (under the covers) temporarily removing the Feature Structure from indexes, doing the modification, and then adding it back.

Similarly to version 2, iterator methods `moveToFirst`, `moveToLast`, and `moveTo(a_positioning_Feature_Structure)` "reset" the iterator to be able to "see" the current state of the indexes. This corresponds to resetting the concurrent modification detection sensing in version 2, when these methods are used.

Note that the phrase *Concurrent Modification* is being used here in a single threading to the indexes. UIMA does not support multi-threaded write access to the CAS; it does support multi-threaded read access to a set of CAS Views, concurrent with one thread having write access (to different views).

The `remove()` API for iterators is now implemented for FSIterators. Its meaning is slightly different from the normal Java meaning - it doesn't remove the item from the collection being iterated over; rather it removes the Feature Structure returned by `get()` from all indexes in the view.

The FSIterator methods that normally check for iterator validity have versions which skip that check. This may be a performance optimization in cases where you can guarantee the iterator is valid, for example if you have a loop which is checking `hasNext()` and following it with a `next()`, which is only executed if the `hasNext()` was true. The non-checking versions are suffixed with Nvc (stands for No Validity Check).

The FSIndex API has a new method, `subType(type-spec)`, which returns an FSIndex for the same index, but specialized to elements which are a subtype of the original index. The type-spec can be either a JCas class, e.g. `MyToken.class`, or a UIMA type instance.

New Select API

A versatile new Select framework for accessing and acting on Feature Structures selected from the CAS or from Indexes or from other collection objects is documented in a separate chapter. This API is integrated with Java 8's Stream facility.

New custom Java objects in the CAS framework

There is a new framework that supports allowing you to add your own custom Java objects as objects transportable in the CAS. A following chapter describes this facility, and some new semi-built-in types that make use of it.

Built-in lists and arrays

The built-in FSArray JCas class is now parameterized with the type of its elements.

UIMA Array and List types implement Iterable, so you can use them like this: `for (MyType item : myArray) ⋯`.

UIMA Arrays and Lists support `contains(element)` and `isEmpty()`.

UIMA Array and List types support a `stream()` method returning a Stream or a type-specialized sub interface of Stream for primitives (IntStream, LongStream, DoubleStream) over the objects in the collection. Omitted are stream types where boxing would occur - Arrays of Byte, Short, Float, Boolean.

The `iterator()` methods for `IntegerList IntegerArrayList, IntegerArray, DoubleArray,` and `LongArray` return an `OfInt / OfDouble / OfLong` instances. These are subtypes of `Iterator` with an additional methods nextInt / nextLong / nextDouble which avoid the boxing of the normal iterator.

The new `select` framework supports stream operations; see the "select" chapter for details.

A new set of methods on UIMA built-in lists, `createNonEmptyNode() and emptyList()`, creates a non-empty node of the type, or retrieves a (shared) empty node of the type. These methods are not static, and create or get the instance in the same CAS as the object instance. These methods are callable on both the empty and non-empty node instances, or on their shared super interface, for example, on NonEmptyFloatList, EmptyFloatList, and FloatList (the common super interface).

A new set of static methods on UIMA built-in lists and arrays, `create(jcas, array_source)` take a Java array of items, and creates a corresponding UIMA built-in list or array populated with items from the array_source.

For UIMA Lists and Arrays, the CAS and JCas has emptyXXXList/Array methods, which return a shared instance of the immutable empty object. The Cas and JCas have generic emptyArray/List, taking an argument JCas class identifying the type, e.g. FloatArray.class, StringList.class, etc.

For lists, there are some new common APIs for all list kinds.

- `push(item)` pushes the item onto an existing list node, creates a new non-empty node, setting its

head to `item` and its tail to the existing list node. This allows easy construction of a list in backwards order.

- `pushNode()` creates and links in a new node in front of this node.
- `insertNode()` creates and links in a new node following this node.
- `createNonEmptyNode()` creates a node of the same type, in the same CAS, without linking it.
- `getCommonTail()` gets the tail of the node
- `setTail()` sets the tail of the node
- `walkList()` walks the list applying a consumer to each item
- `getLength()` walks the list to compute its length
- `emptyList` returns a shared instance of the empty list of the same type, in the same CAS

Built-in lists and arrays have common super classes / interfaces

Some methods common to multiple implements were moved to the super classes, some classes were made abstract (to prevent them from being instantiated, which would be an error). For arrays, a new method common to all arrays, `copyValuesFrom()` copies values from arrays of the same type.

Many UIMA objects implement Stream or Collection

In Java 8, classes which implement Collection can be converted to streams using the `xxx.sream()` method. To better integrate with Java 8, the following UIMA classes and interfaces now implement Stream or Collection:

- FSIndex (implements Collection)
- all of the built-in Arrays, e.g. FloatArray implement Stream, the Integer/long/double arrays implement the non-boxing version
- all of the built-in Lists implement Stream, the IntegerList implements the non boxing version

Reorganized APIs

Some APIs were reorganized. Some of the reorganizations include altering the super class and implements hierarchies, making some classes abstract, making use of Java 8's new `default` mechanisms to supply default implementations in interfaces, and moving methods to more common places. Users of the non-internal UIMA APIs should not be affected by these reorganizations.

As an example, version 2 had two different Java objects representing particular Feature Structures, such as "Annotation". One was used (`org.apache.uima.jcas.tcas.Annotation`) if the JCas was enabled; the other (`org.apache.uima.cas.impl.AnnotationImpl`)otherwise. In version 3, there's only one implementation; the other (AnnotationImpl) is converted to an interface. Annotation now "implements AnnotationImpl.

Use of JCas Class to specify a UIMA type

Several APIs require a UIMA type to be specified. For instance, the API to remove all Feature Structures of a particular type requires the type to be specified. Instead of a UIMA Type object, if there is a JCas cover class for that type, you can pass that as well, as (for example) `Annotation.class`.

JCasGen changes

JCasgen is modified to generate the v3 style of JCas cover classes. It no longer generates the the xxx_Type.java classes, as these are not used by UIMA Version 3.

JCas additional static fields

Static final string fields are declared for each JCas cover class and for each feature that is part of that UIMA type. The fields look like this example, taken from the Sofa class:

```
public final static String _TypeName = "org.apache.uima.jcas.cas.Sofa";
public final static String _FeatName_sofaNum    = "sofaNum";
public final static String _FeatName_sofaID     = "sofaID";
public final static String _FeatName_mimeType   = "mimeType";
public final static String _FeatName_sofaArray  = "sofaArray";
public final static String _FeatName_sofaString = "sofaString";
public final static String _FeatName_sofaURI    = "sofaURI";
```

Each string has a generated name corresponding to the name of the type or the feature, and a string value constant which of the type or feature name. These can be useful in Java Annotations.

Generics added

Version 3 adds generic typing to several structures, and makes use of this to enable users to unclutter their code by taking advantage of Java's type inferencing, in many cases.

Generic types are added to:

- *FSIndex* <T extends FeatureStructure> the type the index is over.
- *FSArray* <T extends FeatureStructure> the type the FSArray holds.
- *FSList* <T extends TOP> the type the FSList holds.
- *SelectFSs* <T extends FeatureStructure> the type the select is producing.

Other changes

The convenience methods in the JCas have been duplicated in the CAS, e.g. `getAllIndexFS`.

New methods `getIndexedFSs(myUimaType)` and `getIndexedFSs(MyJCas.class)` return unmodifiable, unordered Collections of all indexed Feature Structures of the specified type and its subtypes in this CAS's view. This collection can be used in a Java extended-for loop construction. `getIndexedFSs()` is

the same but is for all Feature Structures, regardless of type. These are methods on the CAS, JCas, FSIndexRepository interfaces, and return the Feature Structures of the specified type (including subtypes).

The TypeSystemMgr Interface has a variation of the `commit` method, which has a parameter that specifies the class loader to be used when loading JCas class. This should be used whenever there are user-specified JCas classes associated with the type system. If not specified, it defaults to the class loader used to load the UIMA framework.

The utility class `org.apache.uima.util.FileUtils` has a new method `writeToFile(path, string)`, which efficiently writes a string using UTF-8 encoding to `path`.

The StringArray class has a new `contains(a_string)` method.

The CAS `protectIndexes` method returns an instance of AutoClosableNoException which is a subtype where the close method doesn't throw an exception. This allows writing the try-with-resources form without a catch block for Exception.

Sometimes Annotators may log excessively, causing problems in production settings. Although this could be controlled using logging configuration, sometimes when UIMA is embedded into other applications, you may not have easy access to modify those.

For this case, the produceAnalysisEngine's "additionalParameters" map supports a new key, AnalysisEngine.PARAM_THROTTLE_EXCESSIVE_ANNOTATOR_LOGGING. This key specifies that throttling should be applied to messages produced by annotators using loggers obtained by Annotator code using the getLogger() API.

The value specified must be an Integer, and is the number of messages allowed before logging is suppressed. This number is applied to each logging level, separately. To suppress all logging, use 0.

The Type interface has new methods `subsumes(another_type)`, `isStringOrStringSubtype()`, and `isStringSubtype()`.

The FlowController_ImplBase supports a getLogger() API, which is shorthand for getContext().getLogger().

Many error messages were changed or added, causing changes to localization classes. For coding efficiency, some of the structure of the internal error reporting calls was changed to make use of Java's variable number of arguments syntax.

The UIMA Logger implementation has been extended with both the SLF4J logger APIs and the Log4j APIs which support Java 8's `Supplier` Functional Interfaces.

The TypeSystem and Type object implementations implement `Iterable` and will iterate over all the defined types, or, for a type, all the defined Features for that type.

SelectFS CAS data access

The *select* framework provides a concise way to work with Feature Structure data stored in the CAS. It is integrated with the Java 8 *stream* framework, and provides additional capabilities supported by the underlying UIMA framework, including the ability to move both forwards and backwards while iterating, moving to specific positions, and doing various kinds of specialized Annotation selection such as working with Annotations spanned by another annotation (think of a Paragraph annotation, and the Sentences or Tokens within that).

There are 3 main parts to this framework:

- The source
- what to select, ordering
- what to do

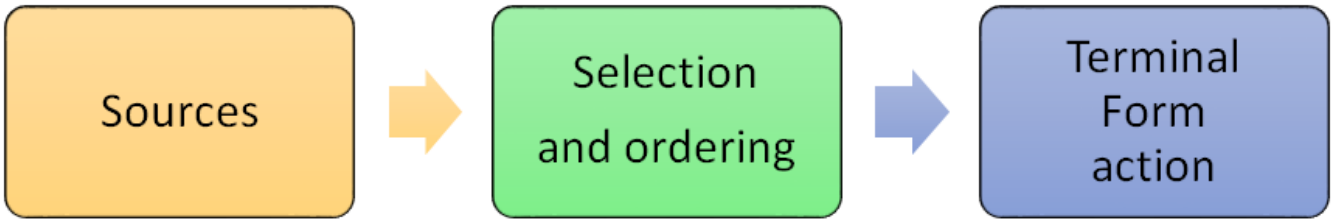


Figure 1. Select - the big picture

These are described in code using a builder pattern to specify the many options and parameters. Some of the very common parameters are also available as positional arguments in some contexts. Most of the variations are defaulted so that in the common use cases, they may be omitted.

Select's use of the builder pattern

The various options and specifications are specified using the builder pattern. Each specification has a name, which is a Java method name, sometimes having further parameters. These methods return an instance of SelectFSs; this instance is updated by each builder method.

A common approach is to chain these methods together. When this is done, each subsequent method updates the SelectFSs instance. This means that the last method in case there are multiple method calls specifying the same specification is the one that is used.

For example,

```
a_cas.select().typePriority(true).typePriority(false).typePriority(true)
```

would configure the select to be using typePriority (described later).

Some parameters are specified as positional parameters, for example, a UIMA Type, or a starting position or shift-offset.

Sources of Feature Structures

Feature Structures are kept in the CAS, and may be accessed using UIMA Indexes. Note that not all Feature Structures in the CAS are in the UIMA indexes; only those that the user had "added to the indexes" are. Feature Structures not in the indexes are not included when using the CAS as the source for the select framework.

Feature Structures may, additionally, be kept in `FSArrays`, `FSLists`, and many additional collection-style objects that implement `SelectViaCopyToArray` interface. This interface is implemented by the new semi-built-in types `FSArrayList`, `FSHashSet` and `FSLinkedHashSet`; user-defined JCas classes for user types may also choose to implement this. All of these sources may be used with `select`.

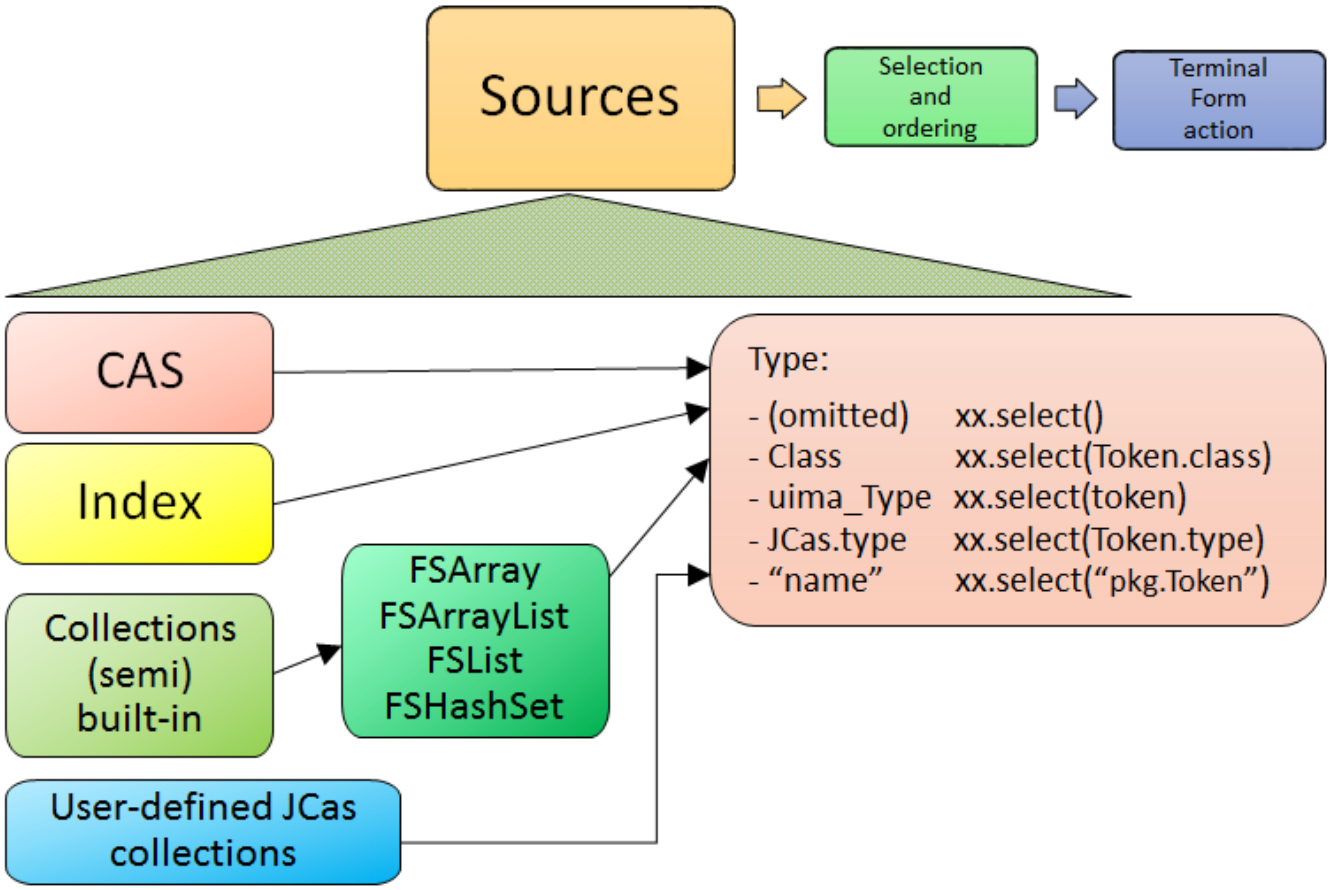


Figure 2. select method with type

For CAS sources, if Views are being used, there is a separate set of indexes per CAS view. When there are multiple views, only one view’s set of indexed Feature Structures is accessed - the view implied by the CAS being used. Note that there is a way to specify aggregating over all views; see `allViews` described later.

For CAS sources, users may specify all Feature Structures in a view, or restrict this in two ways:

- specifying an index: Users may define their own indexes, in additional to the built in ones, and then specify which index to use.
- specifying a type: Only Feature Structures of this type (or its subtypes) are included.

It is possible to specify both of these, using the form `myIndex.select(myType)`; in that case the type must be the type or a subtype of the index’s top most type.

If no index is specified, the default is

- to use all Feature Structures in a CAS View, or
- to use all Feature Structures in the view's AnnotationIndex, if the selection and ordering specifications require an AnnotationIndex.

Note that the non-CAS collection sources (e.g. the FSArray and FSList sources are considered ordered, but non-sorted, and therefore cannot be used for an operations which require a sorted order.

There are 4 kinds of sources of Feature Structures supported:

- a CAS view: all the FSs that were added to the indexes for this view.
- an Index over a CAS view. Note that the AnnotationIndex is often implied by other `select` specifications, so it is often not necessary to supply this.
- Feature Structures from a (semi) built-in UIMA Collection instance, such as instances of the types `FSArray, FSArrayList, FSHashSet,` etc.
- Feature Structures from a user-defined UIMA Collection instance.

UIMA Collection sources have somewhat limited configurability, because they are considered non-sorted, and therefore cannot be used for an operations which require a sorted order, such as the various bounding selections (e.g. `coveredBy`) or positioning operations (e.g. `startAt`).

Each of these sources has a new API method, `select(⋯)`, which initiates the select specification. The select method can take an optional parameter, specifying the UIMA type to return. If supplied, the type must must be the type or subtype of the index (if one is specified or implied); it serves to further restrict the types selected beyond whatever the index (if specified) has as its top-most type.

Use of Type in selection of sources

The optional type argument for `select(⋯)` specifies a UIMA type. This restricts the Feature Structures to just those of the specified type or any of its subtypes. If omitted, if an index is used as a source, its type specification is used; otherwise all types are included.

Type specifications may be specified in multiple ways. The best practice, if you have a JCas cover class defined for the type, is to use the form `MyJCasClass.class`. This has the advantage of setting the expected generic type of the select to that Java type.

The type may also be specified by using the actual UIMA type instance (useful if not using the JCas), using a fully qualified type name as a string, or using the JCas class static `type` field.

Sources and generic typing

The select method results in a generically typed object, which is used to have subsequent operations make use of the generic type, which may reduce the need for casting.

The generic type can come from arguments or from where a value is being assigned, if that target has a generic type. This latter source is only partially available in Java, as it does not propagate past

the first object in a chain of calls; this becomes a problem when using `select` with generically typed index variables.

There is also a static version of the `select` method which takes a generically typed index as an argument.

The best practice is to pass the JCas class representing the type you want, to the select statement. This enables the generic typing mechanism to be set to that type. In the example below, we use `Token` as the type, and `fsIterator()` just as an example of some terminal form action.

```
// Best practice, when possible
// the generic type for Token is passed as an argument to select
FSIterator<Token> token_it = cas.select(Token.class).fsIterator();
```

A compile-time generic type can be specified after the select, if the class argument form of `select` is not used. In these two examples, the generic type is being specified at compile time, explicitly:

```
// ... myCas.select(myType).<Token>fsIterator() ...
// ... myIndexOversomeType.select().<Token>further-operators-of-select-etc
```

Java 8's type inference doesn't take the generic type past the first object in a build chain, so you can use these techniques to overcome that. In these examples, tkn_idx is a generically typed variable:

```
FSIndex<Token> tkn_idx = ... ; // generically typed variable
```

We show a straight-forward syntax that doesn't work, followed by 3 alternatives that do work.

```
// this next fails because the Token generic type from the
// index variable being assigned doesn't get passed to the select().

FSIterator<Token> token_iterator = tkn_idx.select().fsIterator();
```

You can overcome this in three ways:

```
// pass in the type as an argument to select using the JCas cover type.

FSIterator<Token> token_iterator =
    tkn_idx.select(Token.class).fsIterator();

// Or use the static form of select (avoids repeating the type info)

FSIterator<Token> token_iterator =
    SelectFSs.select(tkn_idx).fsIterator();

// Or you can also explicitly set the generic type
// that select() should use, like this:

FSIterator<Token> token_iterator =
    tkn_idx.<Token>select().fsIterator();
```

Note: the static `select` method may be statically imported into code that uses it, to avoid repeatedly qualifying this with its class, `SelectFSs`.

Any specification of an index may be further restricted to just a subType (including that subtype's subtypes, if any) of that index's type. For example, an AnnotationIndex may be specialized to just `Token`s (and their subtypes):

```
FSIterator<Token> token_iterator =
    annotation_index.select(Token.class).fsIterator();
```

Selection and Ordering

There are four sets of sub-selection and ordering specifications, grouped by what they apply to:

- all sources
- Indexes or FSArrays or FSLists
- Ordered Indexes
- The Annotation Index

With some exceptions, configuration items to the left also apply to items on the right.

When the same configuration item is specified multiple times, the last one specified is the one that is used.

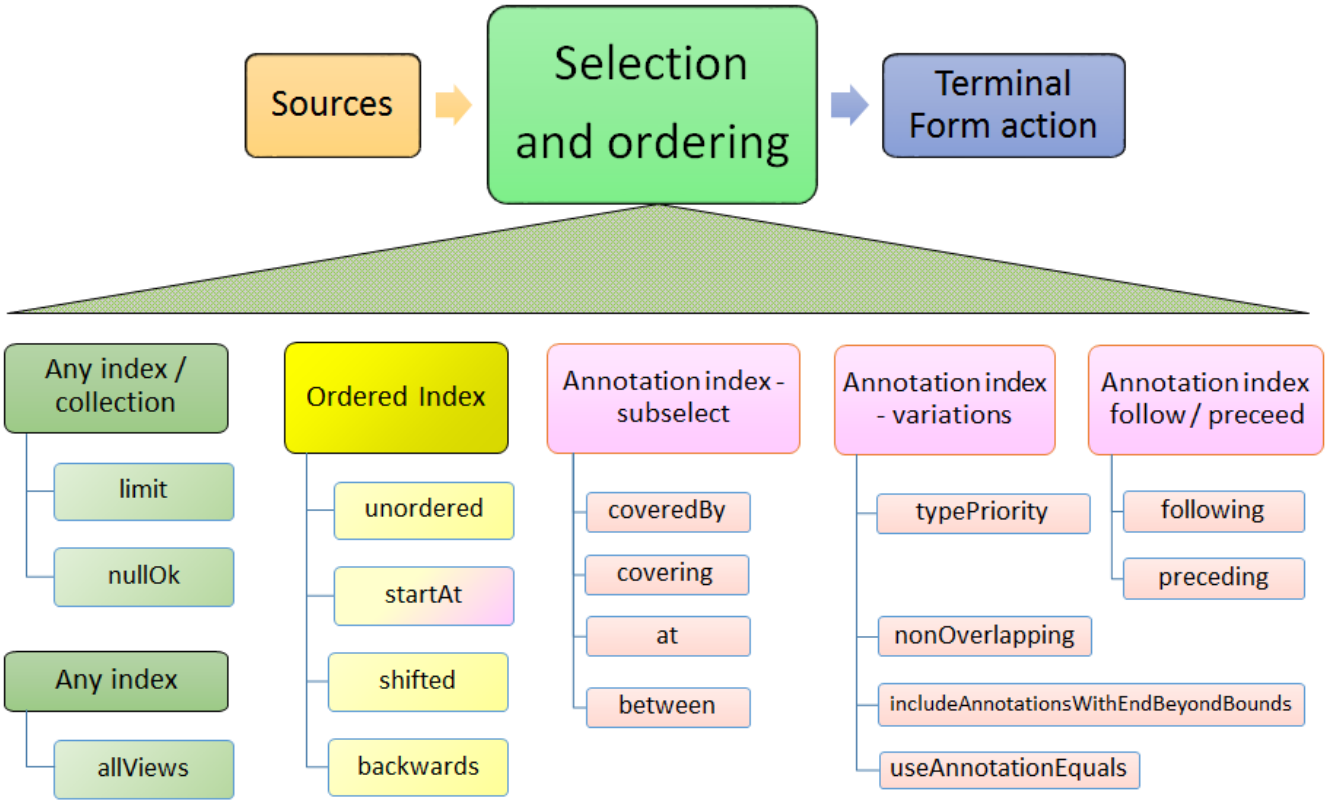


Figure 3. Selection and Ordering

Boolean properties

Many configuration items specify a boolean property. These are named so the default (if you don't specify them) is generally what is desired, and the specification of the method with null parameter switches the property to the other (non-default) value.

For example, normally, when working with bounded limits within Annotation Indexes, type priorities are ignored when computing the bound positions. Specifying typePriority() says to use type priorities.

Additionally, the boolean configuration methods have an optional form where they take a boolean value; true sets the property. So, for example typePriority(true) is equivalent to typePriority(), and typePriority(false) is equivalent to omitting this configuration.

Configuration for any source

limit

a limit to the number of Feature Structures that will be produced or iterated over.

nullOK

changes the behavior for the terminal_form actions `get(…) and single(…)`, which would otherwise throw an exception if a null result happened.

Configuration for any index

allViews

Normally, only Feature Structures belonging to the particular CAS view are included in the

selection. If you want, instead, to include Feature Structures from all views, you can specify `allViews()`.

When this is specified, it acts as an aggregation of the underlying selections, one per view in the CAS. The ordering among the views is arbitrary; the ordering within each view is the same as if this setting wasn't in force. Because of this implementation, the items in the selection may not be unique — Feature Structures in the underlying selections that are in multiple views will appear multiple times.

Configuration for sort-ordered indexes

When an index is sort-ordered, there are additional capabilities that can be configured, in particular positioning to particular Feature Structures, and running various iterations backwards.

orderNotNeeded

relaxes any iteration by allowing it to proceed in an unordered manner. Specifying this may improve performance in some cases. When this is specified, the current implementation skips the work of keeping multiple iterators for a type and all of its subtypes in the proper synchronization.

startAt

position the starting point of any iteration. `startAt(⋯)` can be used for general sorted indexes, and also has special formats only usable for Annotation Indexes.

```
// Forms for any sorted index
startAt(fs);          // fs specifies a feature structure
                      // indicating the starting position

startAt(fs, shifted); // same as above, but after positioning,
                      // shift to the right or left by the shift
                      // amount which can be positive or negative

// Forms for AnnotationIndex sources

startAt(begin);       // sets no TypePriorities, and starts at the
                      //   leftmost annotation whose begin is >= begin
startAt(begin, end);  // start at the position indicated by begin/end

startAt(begin, end, shifted) // same as above,
                             // but with a subsequent shift.
                             // which can be positive or negative
```

The use of `startAt` in conjunction with `following` or `prededing` or any of the bounded sub-selection operators is **not** supported.

backwards

specifies that the result is returned the opposite order than normal. It does not matter at which point in the selection this method is invoked. So e.g. `⋯backwards().limit(5)` returns the same

result as `⋯limit(5).backwards()`. Also, invoking this method twice does not "un-reverse" the results.

Following or Preceding

For an Annotation Index, you can specify all Feature Structures following or preceding a position. The position can be specified either as an Annotation or by specifying an annotation begin index. Both of these can have an additional shift offset amount as a 2nd parameter. Note that the positioning arguments differ from the `startAt` specification, which uses both begin and end values.

following

Position the iterator according to the argument, and then move the iterator forwards until the Annotation at that position has its begin value >= to the positioning annotation's end value.

If the position is specified as an int, move the iterator forwards until the Annotation at that position has its begin value >= the specified int.

preceding

Position the iterator according to the argument, and then move it backwards until the Annotation's (at that position) `end` value is ⇐ to the positioning Annotation's `begin` value.

If the position is specified as an int, treat this as the begin value.

Once positioned, the actual iteration starts at the beginning and ends at the last position.

The `preceding` iteration skips over annotations whose `end` values are > the positioning annotation's begin value, or the positioning int's value.

When using following/preceding in combination with `limit`, `backwards`, `shifted` and `non-overlappin`, the order in which these operations are internally applied is as follows.

1. unambiguous - first, ambiguous annotations are skipped. The shift amount does not affect which which of the ambiguous annotations are skipped.
2. shifted - after removing the ambiguous annotations, items in the result set can be skipped in the direction of the selection. A negative shift is implicitly capped to 0. E.g. consider you have `[10, 20] [20, 30]` and `select().preceding([30, 40])`, the result would be `[10, 20] [20, 30]` (in this order). Because the shift skips away from the reference point, the result of `select().preceding([30, 40]).shifted(1)` is `[10, 20]` and not `[20, 30]`.
3. limit - the limit is applied after shifting. Thus, the amount of shifting has no effect on the limit.
4. backwards - finally, the result set may be reversed.

Bounded sub-selection within an Annotation Index

When selecting Annotations, frequently you may want to select only those which have a relation to a bounding Annotation. A commonly done selection is to select all Annotations (of a particular type

including its subtypes) within the span of another bounding Annotation, for example, all `Tokens` within a `Sentence`.

There are four varieties of sub-selection within an annotation index. They all are based on a bounding Annotation (except the `between` which is based on two bounding Annotations).

The bounding Annotations are specified using either a Annotation (or a subtype), or by specifying the begin and end offsets that would be for the bounding Annotation.

Leaving aside `between` as a special case, the bounding Annotation's `begin` and `end` (and sometimes, its `type`) is used to specify where an iteration would start, where it would end, and possibly, which Annotations within those bounds would be filtered out. There are many variations possible; these are described in the next section.

The returned Annotations exclude the one(s) which are `equal` to the bounding FS. There are several variations of how this `equal` test is done, discussed in the next section.

coveredBy

iterates over Annotations within the bound

covering

iterates over Annotations that span the bound.

at

iterates over Annotations that have the same span (i.e., begin and end) as the bound.

between

uses two Annotations, and returns Annotations that are in between the two bounds, specified by Annotations. If the bounds are backwards, then they are automatically used in reverse order. The meaning of between is that an included Annotation's begin has to be >= the earlier bound's `end`, and the Annotation's end has to be ⇐ the later bound's `begin`.

When using following/preceding in combination with `limit`, `backwards`, `shifted` and `non-overlappin`, the order in which these operations are internally applied is as follows.

1. unambiguous - first, ambiguous annotations are skipped. The shift amount does not affect which which of the ambiguous annotations are skipped.
2. backwards - if requested, reversal is applied before shift and limit.
3. shifted - after removing the ambiguous annotations, items in the result set can be skipped. the end of the. A negative shift is implicitly capped to 0.
4. limit - the limit is applied after shifting. Thus, the amount of shifting has no effect on the limit.

Variations in Bounded sub-selection within an Annotation Index

There are five variations you can specify. Two affect how the starting bound position is set; the other three affect skipping of some Annotations while iterating. The defaults (summarized

following) are designed to fit the popular use cases.

typePriority

The default is to ignore type priorities when setting the starting position, and just use the begin / end position to locate the left-most equal spot. If you want to respect type priorities, specify this variant.

nonOverlapping

Normally, all Annotations satisfying the bounds are returned. If this is set, annotations whose `begin` position is not >= the previous annotation's (going forwards) `end` position are skipped. This is also called *unambiguous* iteration. If the iterator is run backwards, it is first run forwards to locate all the items that would be in the forward iteration following the rules; and then those are traversed backwards. This variant is ignored for `covering` selection.

includeAnnotationsWithEndBeyondBounds

The Subiterator *strict* configuration is equivalent to the opposite of this. This only applied to the `coveredBy` selection; if specified, then any Annotations whose `end` position is > the end position of the bounding Annotation are included; normally they are skipped.

skipSameBeginEndType

While doing bounded iteration, if the Annotation being returned is identical (has the same _id()) with the bounding Annotation, it is always skipped.

Other annotations, which might have the same begin, end, and type values, are not skipped, but instead, included, by default.

When this configuration is specified, any Annotation which has the same begin, end, and type is also skipped.

If you do not want any of the indexed annotations to be skipped, you can achieve this by

- insuring you haven't set `skipWhenSameBeginEndType()`
- making a bounding annotation with the begin / end / type you want for the bound
- Don't add this bounding annotation to the index

Because the bounding annotation will not be equal (have the same Feature Structure ID) as any annotations in the index (because you haven't indexed it), it will never match any annotations found in the index while iterating.

Defaults for bounded selects

The ordinary core UIMA Subiterator implementation defaults to using type order as part of the bounds determination. uimaFIT, in contrast, doesn't use type order, and sets bounds according to the begin and end positions.

This `select` implementation mostly follows the uimaFIT approach by default, but provides the

above configuration settings to flexibly alter this to the user's preferences. For reference, here are the default settings, with some comparisons to the defaults for `Subiterators`:

typePriority

default: false; type priorities are not used when moving to left-most among equal items. Subiterators created using the AnnotationIndex, in contrast, use type priorities.

nonOverlapping

default: false; no Annotations are skipped because they overlap. This corresponds to the "ambiguous" mode in Subiterators.

includeAnnotationsWithEndBeyondBounds

default: (only applies to `coveredBy` selections; The default is to skip Annotations whose end position lies outside of the bounds; this corresponds to Subiterator's "strict" option.

skipSameBeginEndType

default: only the single Annotation with the same _id() is skipped when using a bounded iteration. Use this setting to expand the set of skipped Annotations to include all those equal to the bound's begin, end and type.

Terminal Form actions

After the sources and selection and ordering options have been specified, one terminal form action may be specified. This can be an getting an iterator, array or list, or a single value with various extra checks, or a Java stream. Specifying any stream operation (except limit) converts the object to a stream; from that point on, any stream operation may be used.

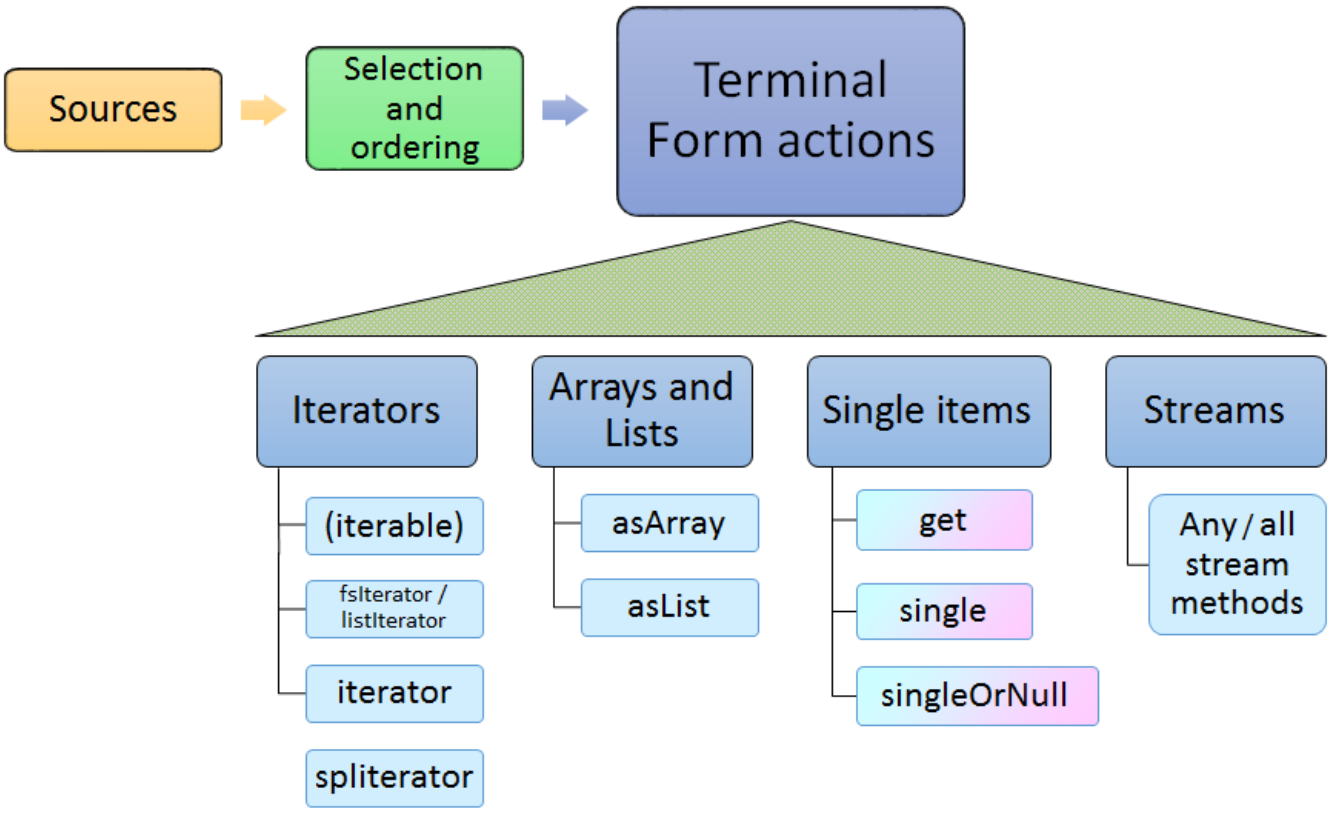


Figure 4. Select Terminal Form Actions

Iterators

(Iterable)

The `SelectFSs` object directly implements `Iterable`, so it may be used in the extended Java `for` loop.

fsIterator

returns a configured fsIterator or subIterator. This iterator implements `ListIterator` as well (which, in turn, implements Java `Iterator`). Modifications to the list using `add` or `set` are not supported.

iterator

This is just the plain Java iterator, for convenience.

spliterator

This returns a spliterator, which can be marginally more efficient to use than a normal iterator. It is configured to be sequential (not parallel), and has other characteristics set according to the sources and selection/ordering configuration.

Arrays and Lists

asArray

This takes 1 argument, the class of the returned array type, which must be the type or subtype of the select.

asList

Returns a Java list, configured from the sources and selection and ordering specifications.

Single Items

These methods return just a single item, according to the previously specified select configuration. Variations may throw exceptions on empty or more than one item situations.

These have no-argument forms as well as argument forms identical to `startAt` (see above). When arguments are specified, they adjust the item returned by positioning within the index according to the arguments.

Positioning arguments with a Annotation or begin and end require an Annotation Index. Positioning using a Feature Structure, by contrast, only require that the index being use be sorted.

get

If no argument is specified, then returns the first item. If there is no item, then an exception is thrown unless `nullOK` is set.

If any positioning arguments are specified, then this returns the item at that position unless there is no item at that position, in which case it throws an exception unless `nullOK` is set.

single

returns the item at the position, but throws exceptions if there are more than one item in the selection, or if there are no items in the selection.

singleOrNull

returns the item at the position, but throws an exception if there are more than one item in the selection.

isEmpty

returns true if the selection is empty.

Streams

any stream method

Select supports all the stream methods. The first occurrance of a stream method converts the select into a stream, using `spliterator`, and from then on, it behaves just like a stream object.

For example, here's a somewhat contrived example: you could do the following to collect the set of types appearing within some bounding annotation, when considered in nonOverlapping style:

```
// items of MyType or subtypes
Set<Type> foundTypes = myIndex.select(MyType.class)
   .coveredBy(myBoundingAnnotation)
   .nonOverlapping()
   .map(fs -> fs.getType())
   .collect(Collectors.toCollection(TreeSet::new));
```

Or, to collect by category a set of frequency values:

```
Map<Category, Integer> freqByCategory = myIndex.select(MyType.class)
     .collect(Collectors
     .groupingBy(MyType::getCategory,
                 Collectors.summingInt(MyType::getFreq)));
```

Annotation relation predicates

When working with annotations, it is often necessary to express how two annotations related to each other. This happens for example when using the Select framework to say "select all annotations of type T that follow a given annotation X". So there are a number of possible relationships which two annotations can have with each other such as "following", "preceding", "being colocated", "being covered by", "covering", etc. This chapter provides specification of these relationships which are also available as a set of predicate functions. The Select framework is also consistent with these definitions. In order to query the CAS for annotations that exist in a certain relationship to each other, it is possible to e.g. stream an entire annotation index or CAS and filtering the annotations using the provided predicate functions. However, using the corresponding selector functions of the Select framework is generally much faster than filtering using the predicates as Select knows more efficient way of finding the starting point for a particular query in the annotation index and also knows when a search can be aborted without having to scan an entire index and also without missing any matches. The predicates are implemented as static functions in the `org.apache.uima.cas.text.AnnotationPredicates` class.

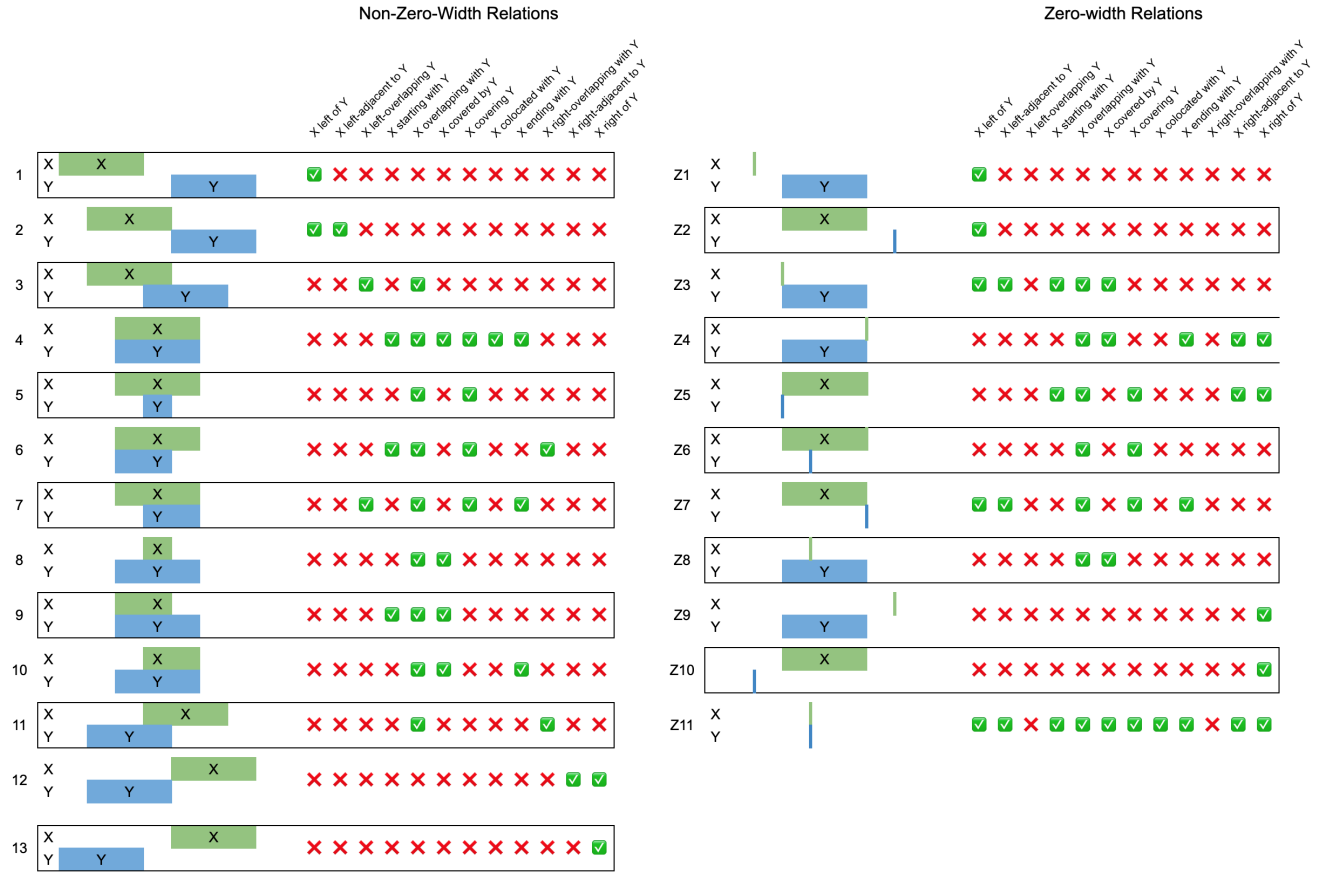


Figure 5. Annotation relation types

As shown below, all of the relations can be expressed in terms of the "covered by" relation.

Definitions:

Bold definitions are axiomatic while non-bold items are derived from axiomatic definitions.

X left-of Y (preceding)	X covered-by [MIN, Y.begin]	X right-of Y (following)	X covered-by [Y.end, MAX]
X left-adjacent-to Y	[X.end, X.end] colocated-with [Y.begin, Y.begin]	X right-adjacent-to Y	[X.begin, X.begin] colocated-with [Y.end, Y.end]
X left-overlapping Y	NOT([X.begin, X.begin] covered-by Y) && [X.end, X.end] covered-by Y && NOT([X.end, X.end] colocated-with [Y.begin, Y.begin])	X right-overlapping Y	[X.begin, X.begin] covered-by Y && NOT([X.end, X.end] covered-by Y) && NOT([X.begin, X.begin] colocated-with [Y.end, Y.end])
X starting-with Y	[X.begin, X.begin] colocated-with [Y.begin, Y.begin]	X ending-with Y	[X.end, X.end] colocated-with [Y.end, Y.end]
X overlapping-with Y	X left-overlapping Y || X right-overlapping Y || X covered-by Y || X.covering Y		
X covered-by Y	**Y.begin <= X.begin && X.end <= Y.end**		
X covering Y	Y covered-by X		
X colocated-with Y (at)	X covered-by Y && Y covered-by X	Note: It is assumed that begin <= end	

X between (Y1,Y2)	X covered-by [Y1.end, Y2.begin]	Note: It is assumed that Y1.end <= Y2.begin

Figure 6. Annotation relation types

CAS-transported custom Java objects

One of the goals of v3 is to support more of the Java collection framework within the CAS, to enable users to conveniently build more complex models that could be transported by the CAS. For example, a user might want to store a Java "Set" object, representing a set of Feature Structures. Or a user might want to use an adjustable array, like Java's ArrayList.

With the current version 2 implementation of JCas, users already may add arbitrary Java objects to their JCas class definitions as fields, but these do not get transported with the CAS (for instance, during serialization). Furthermore, in version 2, the actual JCas instance you get when accessing a Feature Structure in some edge cases may be a fresh instance, losing any previously computed value held as a Java field. In contrast, each Feature Structure in a CAS is represented as the same unique Java Object (because that's the only way a Feature Structure is stored).

Version 3 has a new a capability that enables converting arbitrary Java objects that might be part of a JCas class definition, into "ordinary" CAS values that can be transported with the CAS. This is done using a set of conventions which the framework follows, and which developers writing these classes make use of; they include two kinds of marker Java interfaces, and 2 methods that are called when serializing and deserializing.

The marker interfaces identify those JCas classes which need these extra methods called. The extra methods are methods implemented by the creator of these JCas classes, which marshal/unmarshal CAS feature data to/from the Java Object this class is supporting.

Storing the Java Object data as the value of a normal CAS Feature means that they get "transported" in a portable way with the CAS - they can be saved to external storage and read back in later, or sent to remote services, etc.

Tutorial example

Here's a tutorial example on how to design and implement your own special Java object. For this example, we'll imagine we need to implement a map from FeatureStructures to FeatureStructures.

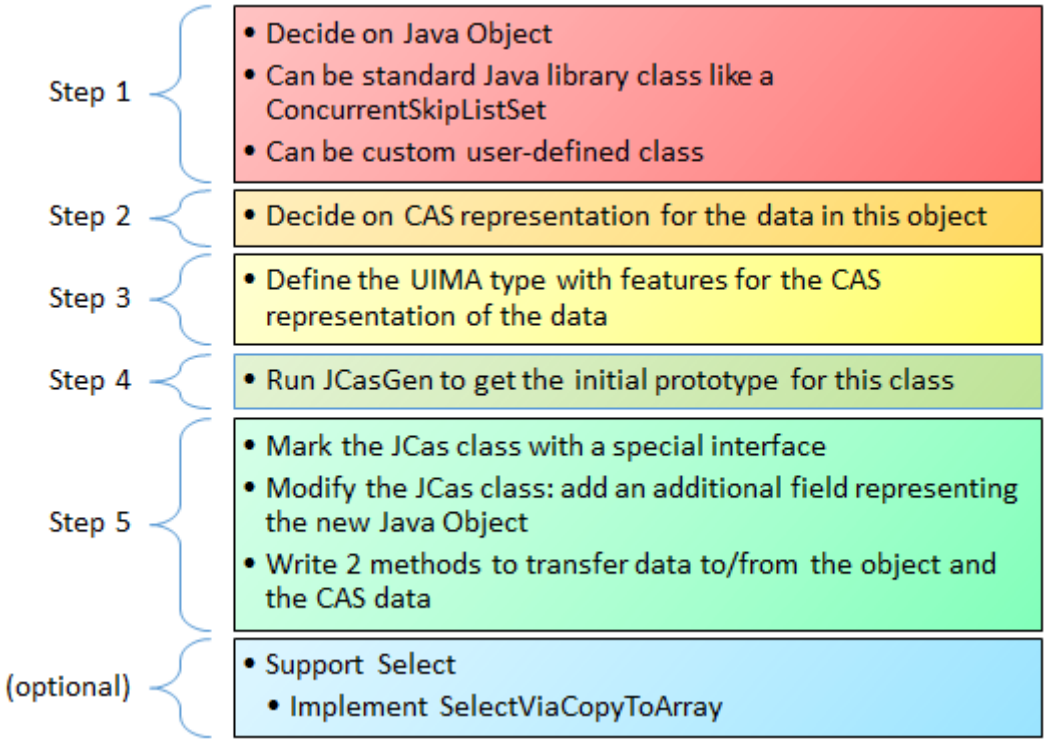


Figure 7. Creating a custom Java CAS-stored Object

Step 1 is deciding on the Java Object implementation to use. We can define a special class, but in this case, we'll just use the ordinary Java HashMap<TOP, TOP> for this.

Step 2 is deciding on the CAS Feature Structure representation of this. For this example, let's design this to represent the serialized form of the hashmap as 2 FSArrays, one for the keys, and one for the values. We could also use just one array and intermingle the keys and values. It's up to the designer of this new JCas class to decide how to do this.

Step 3 is defining the UIMA Type for this. Let's call it FS2FSmap. It will have 2 Features: an FSArray for the keys, and another FSArray for the values. Let's name those features "keys" and "values". Notice that there's no mention of the Java object in the UIMA Type definition.

Step 4 is to run JCasGen on this class to get an initial version of the class. Of course, it will be missing the Java HashMap, but we'll add that in the next step.

Step 5: modify 3 aspects of the generated JCas class.:

+ 1. Mark the class with one of two interfaces::

+ * `UimaSerializable` * `UimaSerializableFSs` These identify this JCas class a needing the calls to marshal/unmarshal the data to/from the Java Object and the normal CAS data features. Use the second form if the data includes any Feature Structure references. In our example, the data does include Feature Structure references, so we add `implements UimaSerializableFSs` to our JCas class.

+ 2. Add the Java Object as a field to the class:: We'll define a new field:

+

```
final private Map<TOP, TOP> fs2fsMap = new HashMap<>();
```

+ 3. Implement two methods to marshal/unmarshal the Java Object data to the CAS Data Features:: ::Now, we need to add the code that translates between the two UIMA Features "keys" and "values" and the map, and vice-versa. We put this code into two methods, called `\_init_from_cas_data` and `\_save_to_cas_data`. These are special methods that are part of this new framework extension; they are called by the framework at critical times during deserialization and serialization. Their purpose is to encapsulate all that is needed to convert from transportable normal CAS data, and the Java Object(s).

+ In this example, the `\_init_from_cas_data` method would iterate over the two Features, together, and add each key value pair to the Java Object. Likewise, the `\_save_to_cas_data` would first create two FSArray objects for the keys and values, and then iterate over the hash map and extract these and set them into the key and value arrays.

+

```
public void _init_from_cas_data() {
  FSArray keys = getKeys();
  FSArray values = getValues();
  fs2fsMap.clear();
  for (int i = keys.size() - 1; i >=0; i--) {
    fs2fsMap.put(keys.get(i), values.get(i));
  }
}

public void _save_to_cas_data() {
  int i = 0;
  FSArray keys = new FSArray(this, fs2fsMap.size());
  FSArray values = new FSArray(this, fs2fsMap.size());
  for (Entry<TOP, TOP> entry : fs2fsMap.entrySet()) {
    keys.set(i, entry.getKey());
    values.set(i, entry.getValues());
    i++;
  }
  setKeys(keys);
  setValues(values);
}
```

+ Beyond this simple implementation, various optimization can be done. One typical one is to treat the use case where no updates were done as a special case (but one which might occur frequently), and in that case having the _save_to_cas_data operation do nothing, since the original CAS data is still valid.

+ One additional "boilerplate" method is required for all of these classes:

+

```
public FeatureStructureImplC _superClone() {return clone();}`
```

+ For custom types which hold collections of Feature Structures, you can have those participate in the `Select` framework, by implementing the optional Interface `SelectViaCopyToArray`.

For more examples, please see the implementations of the semi-built-in classes described in the following section.

Additional semi-built-in UIMA Types for some common Java Objects

Some additional semi-built-in UIMA types are defined in Version 3 using this new mechanism. They work fully in Java, and are serialized or transported to non-Java frameworks as ordinary CAS objects.

Semi-built-in means that the JCas cover classes for these are defined as part of the core Java classes, but the types themselves are not "built-in". They may be added to any tyupe system by importing them by name using the import statement:

```
<import name="org.apache.uima.semibuiltins"/>
```

If you have a Java project whose classpath includes uimaj-core, and you run the Component Descriptor Editor Eclipse plugin tool on a descriptor which includes a type system, you can configure this import by selecting the Add on the Import type system subpanel, and import by name, and selecting org.apache.uima.semibuiltins. (Note: this will not show up if your project doesn't include uimaj-core on its build path.)

FSArrayList

`org.apache.uima.jcas.cas.FSArrayList` is like the current FSArray, except that it implements the List API and supports adding to the array, with automatic resizing, like an ArrayList in Java. It is implemented internally using a Java ArrayList.

The CAS data form is held in a plain FSArray feature.

The `equals()` method is true if both FSArrayList objects have the same size, and contents are `equal` item by item. The list of supported operations includes all of the operations of the Java `List` interface. This object also includes the `select` methods, so it can be used as a source for the `select` framework.

IntegerArrayList

`org.apache.uima.jcas.cas.IntegerArrayList` is like the current IntegerArray, except that it implements the List API and supports adding to the array, with automatic resizing, like an ArrayList in Java.

The CAS data form is held in a plain IntegerArray feature.

The `equals()` method is true if both IntegerArrayList objects have the same size, and contents are `equal` item by item. The list of supported operations includes a subset of the operations of the Java `List` interface, where certain values are changed to Java primitive `ints`. To support the `Iterable` interface, there is a version of `iterator()` where the result is "boxed" into an Integer. For efficiency, there's also a method intListIterator, which returns an instance of IntListIterator, which permits iterating forwards and backwards, without boxing.

FSHashSet and FSLinkedHashSet

`org.apache.uima.jcas.cas.FSHashSet` and `org.apache.uima.jcas.cas.FSLinkedHashSet` store Feature Structures in a (Linked) HashSet, using whatever is defined as the Feature Structure's `equals` and `hashcode`.

> You may customize the particular equals and hashcode by creating a wrapper class that is a subclass of the type of interest which forwards to the underlying Feature Structure, but has its own definition of `equals` and `hashcode`.

The CAS data form is held in an FSArray consisting of the members of the set.

If you want a predictable iteration order, use FSLinkedHashSet instead of FSHashSet.

Int2FS Int to Feature Structure map

Some applications find it convenient to have a map from ints to Feature Structures. In UIMA V2, they made use of the low level CAS APIs that allowed getting an Feature Structure from an int id using `ll_getFSForRef(int)`.

In v3, use of the low level APIs in this manner can be enabled, but is discouraged, because it prevents garbage collection of non-reachable Feature Structures.

`org.apache.uima.jcas.cas.Int2FS<T>` maps from `int`s to `Feature Structure`s of type T. This provides an alternative way to have int → FS maps, under user control of what exactly gets added to them, supporting removes and clearing, under application control

The `iterator()` method returns an Iterator over `IntEntry<T>` objects - these are like java `Entry<K, V>` objects except the key is an int.

Design for reuse

While it is possible to have a single custom JCas class implement multiple Java Objects, this is typically not a good design practice, as it reduces reusability. It is usually better to implement one custom Java object per JCas class, with an associated UIMA type, and have that as the reusable entity.

Logging

Logging has evolved; two major changes now supported by V3 are

- using a popular open-source standard logging facade, SLF4j, that can at run time discover and hook to a user specified logging framework.
- Support for both old-style and new style substitutable parameter specification.

For backwards compatibilit, V3 retains the existing V2 logging facade, so existing code will continue to work. The APIs have been augmented by the methods available in the SLF4j `Logger` API, plus the Java 8 enabled APIs from the Log4j implementation that support the `Supplier` Functional Interface.

The old APIs support messages using the standard Java Util Logging style of writing substitutable parameters using an integer, e.g., {0}, {1}, etc. The new APIs support messages using the modern substitutable parameters without an integer, e.g. {}.

The implementation of this facade in V2 was the built-in-to-Java (java.util) logging framework. For V3, this is changed to be the SLF4j facade. This is an open source, standard facade which allows deferring until deployment time, the specific logging back end to use.

If, at initialization time, SLF4J gets configured to use a back end which is either the built-in Java logger, or Log4j-2, then the UIMA logger implementation is switched to UIMA's implementation of those APIs (bypassing SLF4j, for efficiency).

The SLF4j and other documentation (e.g., https://logging.apache.org/log4j/2.x/log4j-slf4j-impl/index.html for log4j-2) describe how to connect various logging back ends to SLF4j, by putting logging back-end implementations into the classpath at run time. For example, to use the back end logger built into Java, you would include the `slf4j-jdk14` Jar. This Jar is included in the UIMA binary distribution, so that out-of-the-box, logging is available and configured the same as it was for V2.

The Eclipse UIMA Runtime plugin bundle excludes the slf4j api Jar and back ends, but will "hook up" the needed implementations from other bundles.

Logging Levels

There are 2 logging level schemes, and there is a mapping between them. Either of them may be used when using the UIMA logger. One of the schemes is the original UIMA v2 level set, which is the same as the built-in-to-java logger levels. The other is the scheme adopted by SLF4J and many of its back ends.

Log statements are "filtered" according to the logging configuration, by Level, and sometimes by additional indicators, such as Markers. Levels work in a hierarchy. A given level of filtering passes that level and all higher levels. Some levels have two names, due to the way the different logger back-ends name things. Most levels are also used as method names on the logger, to indicate logging for that level. For example, you could say `aLogger.log(Level.INFO, message)` but you can also say `aLogger.info(message)`). The level ordering, highest to lowest, and the associated method names are as follows:

- SEVERE or ERROR; error(...)
- WARN or WARNING; warn(...)
- INFO; info(...)
- CONFIG; info(UIMA_MARKER_CONFIG, ...)
- FINE or DEBUG; debug(...)
- FINER or TRACE; trace(...)
- FINEST; trace(UIMA_MARKER_FINEST, ...)

The CONFIG and FINEST levels are merged with other levels, but distinguished by having `Markers`. If the filtering is configured to pass CONFIG level, then it will pass the higher levels (i.e., the INFO/WARN/ERROR or their alternative names WARNING/SEVERE) levels as well.

Context Data

Context data is kept in SLF4j MDC maps; there is a separate map per thread. This information is set before calling Annotator's process or initialize methods. The following table lists the keys and the values recorded in the contexts; these can be retrieved by the logging layouts and included in log messages.

Because the keys for context data are global, the ones UIMA uses internally are prefixed with "uima_".

Key Name	Description
uima_annotator	the annotator implementation name.
uima_annotator_context_name	the fully qualified annotator context name within the pipeline. A top level (not contained within any aggregate) annotator will have a context of "/".
uima_root_context_id	A unique id representing the pipeline being run. This is unique within a class-loader for the UIMA-framework.
uima_cas_id	A unique id representing the CAS being currently processed in the pipeline. This is unique within a class-loader for the UIMA-framework.

Markers used in UIMA Java core logging

Not (yet) implemented; for planning purposes only.

Defaults and Configuration

By default, UIMA is configured so that the UIMA logger is hooked up to the SLF4j facade, which may

or may not have a logging back-end. If it doesn't, then any use of the UIMA logger will produce one warning message stating that SLF4j has no back-end logger configured, and so no logging will be done.

When UIMA is run as an embedded library in other applications, slf4j will use those other application's logging frameworks.

Each logging back-end has its own way of being configured; please consult the proper back-end documentation for details.

For backwards compatibility, the binary distribution of UIMA includes the slf4j back-end which hooks to the standard built-in Java logging framework, so out-of-the-box, UIMA should be configured and log by default as V2 did.

Throttling logging from Annotators

Sometimes, in production, you may find annotators are logging excessively, and you wish to throttle this. But you may not have access to logging settings to control this, perhaps because UIMA is running as a library component within another framework. For this special case, you can limit logging done by Annotators by passing an additional parameter to the UIMA Framework's produceAnalysisEngine API, using the key name `AnalysisEngine.PARAM_THROTTLE_EXCESSIVE_ANNOTATOR_LOGGING` and setting the value to an Integer object equal to the the limit. Using 0 will suppress all logging. Any positive number allows that many log records to be logged, per level. A limit of 10 would allow 10 Errors, 10 Warnings, etc. The limit is enforced separately, per logger instance.

This only works if the logger used by Annotators is obtained from the Annotator base implementation class via the `getLogger()` method.

Migrating to UIMA Version 3

Migrating: the big picture

Although UIMA V3 is designed to be backwards compatible with UIMA V2, there are some migration steps needed. These fall into two broad use cases:

- if you have an existing UIMA pipeline / application you wish to upgrade to use V3
- if you are "consuming" the Maven artifacts for the core SDK, as part of another project

How to migrate an existing UIMA pipeline to V3

UIMA V3 is designed to be binary compatible with existing UIMA V2 pipelines, so compiled and/or JAR-ed up classes representing a V2 pipeline should run with UIMA v3, with three changes:

- Java 8 is required. (If you're already using Java 8, nothing need be done.)
- Any defined JCas cover classes must be migrated or regenerated, and used instead. (If you do not define any JCas classes or don't use JCas in your pipeline, then nothing need be done.) A quick way to do this is to create a Jar with the migrated JCas classes, and put it into the classpath ahead of the other JCas class definitions.
- The runtime classpath needs to include the slf4j-api Jar, and an appropriate slf4j bridging Jar, for details, see next.

Some adjustments may need to be made to logging setup, typically by including additional Jars (provided in the UIMA Binary distribution) in your application's classpath. If you are using the standard UIMA Launch scripts, this is already done. For custom application setups, insure that the classpath includes the (now) required jar "slf4j-api-xxxx.jar" (replace xxxx with the version). If you were using the standard UIMA based logging, to get the similar behavior, include the slf4j-jdk14-xxxx.jar; this enables the standard Java Utility Logging facility.

Some Maven projects use the JCasGen maven plugin; these projects' JCasGen maven plugin, if switched to UIMA V3, automatically generate the V3 versions. For proper operation, please run maven clean install; the clean operation ought to remove the previously generated JCas class, including the UIMA V2 `xxx_Type` classes. These are no longer used, and won't compile in V3.

You can use any of the methods of invoking JCasGen to generate the new V3 versions. If using the Eclipse plugins (i.e., pushing the `JCasGen`) button in the configuration editor, etc.), the V3 version of the plugin must be the one installed into Eclipse.

If you have the source or class files, you can also migrate those using the migration tool described in this section. This approach is useful when you've customized the JCas class, and wish to preserve those customizations, while converting the v2 style to the v3 style.

Migrating JCas classes

If you have customized JCasGen classes, these can be migrated by running the migration tool, which

is available as a stand-alone command line tool (`runV3migrateJCas.sh` or `⋯bat`), or as Eclipse launch configurations.

This tool can migrate either sets of

- Java source files (xxx.java) or
- Compiled Java class files (including those contained in JARs or PEARs)

```
 Usually, if you have the source code it is best to migrate the sources.
Otherwise, you can migrate the compiled classes.
The compiled classes are run through a decompiler, and then the derived sources are
migrated.
```

When migrating **source** files, you specify one or more "roots" - places in a file directory, or a single java JCas source file (the one not ending in "_Type"). When directories are specified, the tool scans those directories recursively (including inside Jars and PEARs), looking for JCas source files. If just one source file is specified, it work on just that one source file. When a source file is processed, it is copied to the output spot and migrated. The output is arranged in parallel directories (before and after migration), for easy side-by-side comparing in a tool such as Eclipse file compare.

After checking the migration results, including comparing the files, you replace the original source with the migrated versions. Also, the original V2 source would contain a source file for each JCas class ending in "_Type"; these are not used in version 3 and should be deleted.

You may also migrate **class** files; this can be used when the source files are not available. This option has a decompilation step, to produce the source to be migrated and requires a classpath (passed as the `migrationClasspath` parameter); this classpath is used to resolve symbols during the decompilation, and should be the classpath used when running those classes. For class files, the migration tool attempts to compile the results and, for Jars and PEARs, to update those migrated classes in a copy of the original packaging (meaning, within Jars or PEARs):

- The **classesRoots** are used to locate .class files, perhaps within Jars and PEARs.
- These are decompiled, using special versions of the migrateClasspath.
- The resultant sources are migrated.
- The migrated sources are compiled.
- If the original classes came from Jars or PEARs, copies of these are made with the migrated classes replaced.

When scanning directories from source or class roots, if a Jar or a PEAR is encountered, it is recursively scanned.

When migrating from compiled classes:

- The class is decompiled, and the resulting source is migrated.
- The next 2 steps are skipped if no Java compiler is available. A compiler is available if the migrate utility is being run using a JDK (as opposed to a JRE version of Java).

- The migrated classes are compiled. During this processes, the classpath used is the same as the decompile classpath, except that the uima-core Jar for version 3 (from the classpath used to run the migration tool) is prepended so that the migrated version can be compiled.
- Finally, if the original "packaging" of the class files is a Jar or PEAR, it is copied and updated with the migrated classes (provided there was no compile error).

The results of the migration include the migrated files, a set of logs, and for classesRoots: the compiled classes, and repackaging of them into copies of original Jars and/or PEARs. The migration operation is summarized in the console output, detailing anything that might need inspection to verify the migration was done correctly.

> If all is OK, the migration will say that it "finished with no unusual conditions", at the end.

To complete the migration, fix any reported issues that need fixing, and then update your UIMA application to use these classes/Jars/PEARs in place of the version 2 ones.

The actual migration step is a source-to-source transformation, done using a parse of the source files. The parts in the source which are version 2 specific are replaced with the equivalent version 3 code. Only those parts which need updating are modified; other code and comments which are part of the source file are left unchanged. This is intended to preserve any user customization that may have been done.

After running the tool, it is important to examining the console output and logs. You can confirm that the migration completed without any unusual conditions, or, if something unusual was encountered, you can take corrective action.

Running the migration tool

The tool can be run as a stand-alone command, using the launcher scripts `runV3migrateJCas`; there are two versions of this — one for windows (ending it `.bat`) and one for linux / mac (ending in `.sh`). If you run this without any arguments, it will show a brief help for the arguments.

There are also a pair of Eclipse launch configurations (one for migrating source file(s), the other for compiled classes and JARs and PEARs), which are available if you have the uimaj-examples project (included in the binary distribution of UIMA) in your Eclipse workspace.

Using Eclipse to run the migration tool

There are two Eclipse launch configurations; one works with source code, the other with compiled classes or Jars or PEARs. The launch configurations are named:

- UIMA Run V3 migrate JCas from sources roots
- UIMA Run V3 migrate JCas from classes roots

When running from class directory roots, the classes must not have compile errors, and may contain Jars and PEARs. Both launchers write their output to a temporary directory, whose name is printed in the Eclipse console log.

To use the Eclipse launcher to migrate from source code,

- First select the eclipse project containing the source code to transform; this project's "build path" will also supply the classpath used during migration.

  Alternatively, you may select just one source file to migrate.

- run the migrate-from-sources launcher.

This will scan the directory tree of the project, looking for source files which are JCas files, and migrate them, or alternatively, just work on the single selected source file. No existing files are modified; everything is written to the output directory.

To use the launcher for compiled code,

- First select the eclipse project that provides the classpath for the compiled code. This is required for proper "decompiling" of the classes and recompiling the transformed results.
- The launcher will additionally prompt you for another directory which the migration tool will use as the top of a tree to scan for compiled Java JCas classes to be migrated.

Running from the command line

Command line: Specifying input sources

Input is specified using these arguments:

"-sourcesRoots"

a list of one or more directories, separated by the a path separator character (";" for Windows, ":" for others), or a single source file

Migrates each candidate source file found in any of the file tree roots, skipping over non-JCas classes.

"-classesRoots"

a list of one or more directories containing class files or Jars or PEARs, separated by the a path separator character (";" for Windows, ":" for others).

Decompiles, then migrates each candidate class file found in any of the file tree roots (skipping over non-JCas classes). You can specify either of these, but not both.

Command line: Specifying a classpath for the migration

When migrating from compiled classes, a classpath is required to locate and decompile the JCas classes to be migrated. This classpath should include the JCas classes to be decompiled. The compiled classes must not have compile errors.

When migrating from sourcesRoots, this argument is required only if the JCas classes have references to other non-migrated classes (other than core UIMA classes). For example, if your JCas class had a reference to a user defined Utility class, that would need to be in the classpath. For plain, non-customized JCas classes, this argument is unnecessary.

To specify this parameter, use the argument `-migrateClasspath`. The Eclipse launcher "UIMA run V3 migrate JCas from classes roots" sets this argument using the selected Eclipse project's classpath. When migrating within a PEAR, the migration tool automatically adds the classpath specified by the PEAR (if any) to the classpath.

Handling duplicate definitions

Sometimes, a classpath or directory tree may contain multiple instances of the same JCas class. These might be identical, or they might be different versions.

The migration utility handles this by migrating each instance. The migrated forms are stored in the output directory prefixed by the root-id (see above), as the parent directory. The different versions can then be conveniently compared using tooling such as Eclipse's file compare.

Understanding the reports

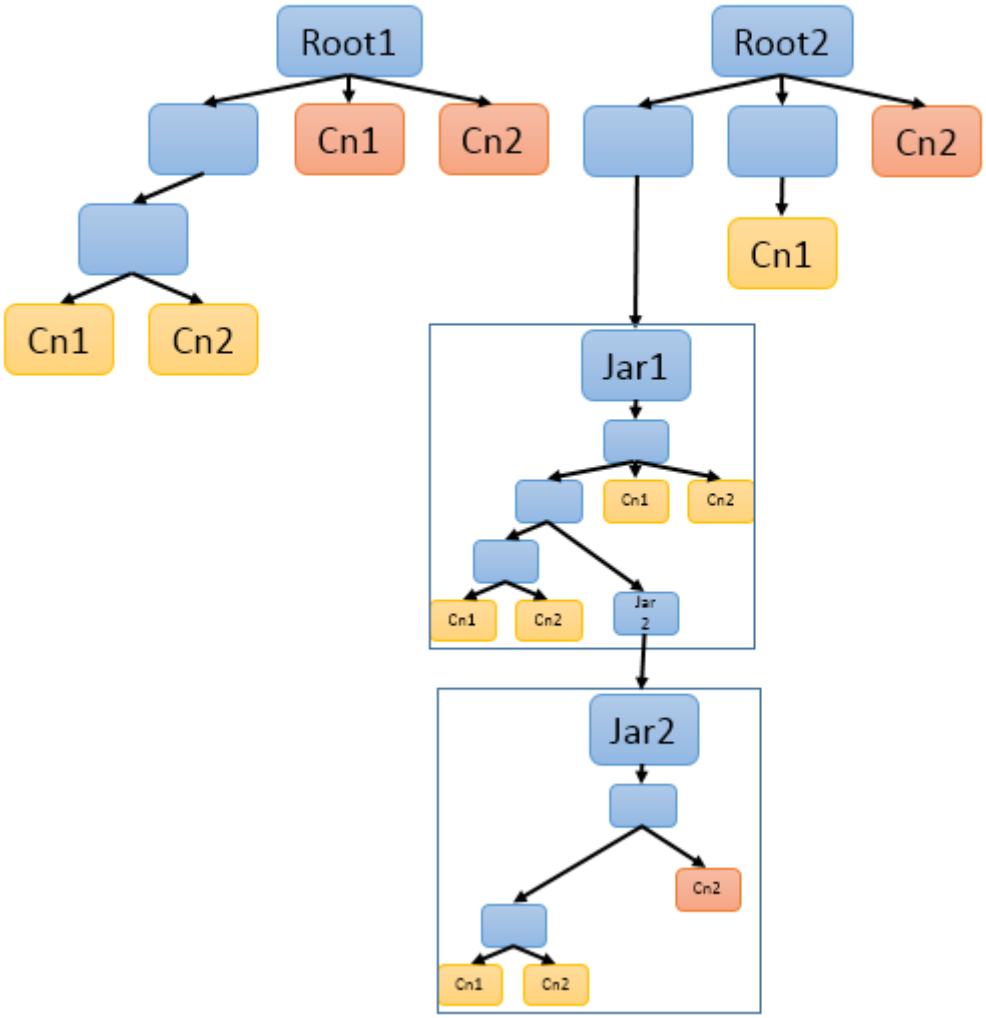
The output directory contains a logs directory with additional information. A summary is also written to System.out.

Each file translated has both a v2 source and a v3 source. When the input is ".class" files, the v2 source is the result of the decompilation step, prior to any migration.

The process of scanning directories to find JCas class to migrate may come across multiple instances of the same class. There are two subcases:

- The instances are the same.
- The instances are different (two non-identical definitions for the same class). Sometimes these arise when migrating from compiled classes, where the compilation was done by different versions of the Java compiler, and the resulting decompilations are logically equal but have some fields or methods in a different order.

This diagram illustrates some of the potentials for identical and non-identical duplicate definitions for the same classname, that the tool may encounter. The blue boxes represent ordinary file directories or Jars, and the other boxes with labels Cn1 and Cn2 represent the definitions for a classes named Cn1 and Cn2; the different colors represent non-identical definitions, as an example. Note that a definition for a class might appear sometimes not within a Jar (or a PEAR, not shown here), as well as with that.




```
 The migration tool allows for all of these variants.
It will  migrate all versions, and will (when migrating from compiled Jars and PEARs)
compile and reassemble these.
```

The output directories prefix the package/classname holding the source code with a prefix of "a0", "a1", etc. The "a" stands for alternative, and the 0 is for the first alternative, and the 1, 2, ... are for other non-equal alternatives.

When the migration is run from compiled classes, then, if possible, the resulting migrated classes are recompiled and if from Jars or PEARs, reassembled into copies of those artifacts. The compilation for the same classname, with the same sourcecode, could be different for different containers because each compilation is done with that container's classpath (e.g. Jar or Pear) and with respect to the compilation units of that container.

Because of this, the compiled results for a given source instance, are done separately, and kept in output directories, indexed additionally by the container number, as "c0", "c1", ... A list of all container numbers and the migrated classes within those containers, is printed out to enable correlating these by hand when necessary.

The overall directory output directory tree looks like:

```
Directory structure, starting at -outputDirectory
   converted/
     v2/
       a0/pkg/name.../Classname.java
                           /Classname2.java etc.
       a1/pkg/name.../Classname.java  if there are multiple
                                              different versions
       ...
     v3/
       a0/pkg/name.../Classname.java
                       /Classname2.java etc.
       a1/pkg/name.../Classname.java  if there are multiple
                                              different versions
       ...

     v3-classes/   for Jars and PEARs, the compiled class
       // xyz is the path in the container to the
       //       start of the pkg/name.../Classname.class
       // the "a0", "a1", ... is extra but serves to
       //      identify which alternative of the source
       23/a0/xyz/pkg/name.../Classname.class
       33/a0/xyz/pkg/name.../Classname.class
       42/a0/xyz/pkg/name.../Classname.class
         ...

     pears/
       // xyz_updated_pear_copy is the path
       //    relative to the container, of the PEAR
       33/xyz_updated_pear_copy.pear
         ...

     jars/
       // xyz_updated_jar_copy is the path
       //    relative to the container, of the Jar
       42/xyz_updated_jar_copy.jar
         ...

   not-converted/

   logs/
     processed.txt
     failed.txt
     skippedBuiltins.txt
     nonJCasFiles.txt
     workaroundDir.txt
     deletedCheckModified.txt
     manualInspection.txt
     pearFileUpdates.txt
```

```
    jarFileUpdates.txt

    ...
```

The converted subtree holds all the sources and migrated versions that were successfully migrated. The not-converted subtree hold the sources that failed in some way the migration. The logs contain many kinds of entries for different issues encountered:

processed.txt

List of successfully processed classes

failed.txt

List of classes that failed to migrate

skippedBuiltins.txt

List of classes representing built-ins that were skipped. These need manual inspection to see how to merge with new v3 built-ins.

NonJCasFiles.txt

List of files that were thought to be JCas classes but upon further analysis appear to not be. These need manual inspection to confirm.

deletedCheckModified.txt

List of class where a version 2 if statement doing the "featOkTst" was apparently modified. In the migrated code, this statement was deleted, perhaps incorrectly. These need manual inspection to confirm.

manualInspection.txt

List of files where the migration found a get or set method, where the version 2 code was accessing a casFeatCode with the feature name not matching. These need manual inspection.

jarsFileUpdates.txt

List of Jar files and classes which were replace in them.

pearsFileUpdates.txt

List of Pear files and classes which were replace in them.

Examples

Run the command line tool:

```
cd $UIMA_HOME

bin/runV3migrateJCas.sh

  -migrateClasspath /home/me/myproj/xyz.jar:$UIMA_HOME/lib/uima-core.jar

  -classesRoots /home/me/myproj/xyz.jar:/home/me/myproj/target/classes

  -outputDirectory /temp/migratejcas
```

Run the Eclipse launcher:

First, make sure you've installed the V3 UIMA plugins into Eclipse!

```
Startup an Eclipse workspace containing the project
with JCas source files to be migrated.

Select the Java project with the JCas sources to be migrated.

Eclipse -> menu -> Run -> Run configurations
  Use the search box to find
  "UIMA run V3 migrate JCas from sources" launcher.
```

Please read the console output summarization to see where the output went, and about any conditions found during migration which need manual inspection and fixup.

Consuming V3 Maven artifacts

Projects may have tests which write to the UIMA log. Because V3 switched to SLF4J as the default logger, unless SLF4J can find an adapter to some back-end logger, it will issue a message and substitute a "NO-OP" back-end logger. If your test cases depend on having the V2 default logger (which is the one built into Java), you need to add a "test" dependency that specifies the SLF4J-to-JDK14 adapter to your POM. Here's the xml for that:

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-jdk14</artifactId>
  <version>1.7.24</version>  <!-- or some version you need -->
  <scope>test</scope>
</dependency>
```

PEAR support

PEARs continue to be supported in Version 3, with the same capabilities as in version 2. Here's a brief review.

PEARs are both a packaging facility, and an isolation facility. The packaging facility allows putting together into one PEAR file all the parts needed for a particular (reusable) UIMA pipeline, including annotators and other data resources, and a classpath to use. PEARs are loaded using special class loaders that load first from whatever classpath is specified by the PEAR; this serves to isolate dependencies and insure that the PEAR makes use of whatever versions of classes it depends on (and specifies in its classpath).

PEARs establish a boundary within a UIMA pipeline — annotator code is running either inside a PEAR, or not. Note that PEARs cannot be nested. The CAS, flowing through a pipeline, is dynamically updated with the current PEAR context (if any).

JCas issues

JCas classes defining Java implementations for UIMA Types may be defined within a PEAR. These are loaded using the isolating Classloader, just like all the other PEAR resources. As a result, this may cause some issues if the same JCas class is also defined outside the PEAR boundary, and loaded with the normal UIMA classloader. The result of having the same JCas class both on the PEAR classloader and outside that classloader will be that Java will have both classes loaded, and code within the PEAR will be linked with one of them, and code outside the PEAR will be linked with the other.

Sometimes, this is exactly what you might want. For example, you might have in the pear, a special JCas definition of a UIMA type "Token" which the PEAR uses, while you might have another JCas definition for that same UIMA type outside of the PEAR. Note that UIMA will always merge Type definitions from inside and outside of PEARs, when it sets up a pipeline - it merges all type definitions found for the whole pipeline.

A consequence of having two loaded class definitions in two contexts for the same UIMA type means that the classes have the same names, but are different (because of different loading classloaders), and assigning one to the other in Java will produce a ClassCast exception.

Othertimes, you may not want different classes. For instance, the class definitions might be identical, and you want to create some "Token" annotations within the PEAR, and have them used by JCas references outside of the PEAR.

In this case, the simplest thing to do is to install the PEAR, but then update its classpath so it no longer includes the JCas classes that came with the PEAR. When classes are not found with the special PEAR class loader, that loader delegates to its parent, which is the normal UIMA class loader. This action will cause the PEAR to use the identically same JCas class within the PEAR as is used outside of the PEAR, and no Class Cast Exception issues will arise. This is the most efficient way to run with PEARs that use JCas classes where you want to share results inside and outside of PEARs.

Version 3 has special support for the case where there are different definitions of JCas classes for the same UIMA type, inside and outside the PEAR. It does this using what are called PEAR Trampolines. When there are multiple JCas definitions, the one defined outside of the PEAR is the one stored internally in UIMA's indexes and types that have references to Feature Structures. Accessing the Feature Structures checks (by asking the CAS) to see if its in a particular PEAR context (there may be several in one pipeline), and if so, a trampoline instance of the Feature Structure is created / used / accessed. The trampoline instance shares internally the CAS data with the base instance, but is a separate instance of the PEAR's JCas class definition. This allows seamless access both inside and outside of the PEAR context to the particular JCas class definition needed.

Custom Java Objects

Custom Java Objects may store references to Feature Structures. If it is desired to create these inside a PEAR, and yet have the references work outside a PEAR, the implementor of these must insure that the actual stored JCas class for a Feature Structure is the base version, not the PEAR version, and also insure that any references are properly converted (while within a PEAR context).

Refer to the implementation of `FSHashSet` and `FSArrayList` to see what needs to be done to make these "Pear aware".

Migration aids

To aid migration, some features of UIMA V3 which might cause migration difficulties can be disabled. Users may initially want to disable these, and get their pipelines working, and then over time, re-enable these while fixing any issues that may come up, one feature at a time.

Global JVM properties for UIMA V3 that control these are described in the table below.

Properties Table

This table describes the various JVM defined properties; specify these on the Java command line using `-Dxxxxxx`, where the `xxxxxx` is one of the properties starting with `uima.` from the table below.

Title	Property Name & Description
Use UIMA V2 format for toString() for Feature Structures	`uima.v2_pretty_print_format` The native v3 format for pretty printing feature structures includes an id number with each FS, and some other minor improvements. If you have code which depends on the exact format that v2 UIMA produced for the toString() operation on Feature Structures, then include this flag to revert to that format.
Disable Type System consolidation	`uima.disable_typesystem_consolidation` Default: equal Type Systems are consolidated. When type systems are committed, the resulting Type System (Java object) is considered read-only, and is compared to already existing Type Systems. Existing type systems, if found, are reused. Besides saving storage, this can sometimes improve locality of reference, and therefore, performance. Setting this property disables this consolidation.

Enable strict type source checking	`uima.enable_strict_type_source_check` Default: checking whether the type actually belongs to the index/CAS is performed but only logs a warning, no exception. When creating a new feature structure or when adding or removing a feature structure to/from an index, it is checked that the type system the type belongs to is exactly the same instance as the type system of the CAS it is created in or the index it is added to. Due to the type system consolidation feature, this should always be the case. Setting this property causes an exception to be thrown - otherwise a warning is logged.
Disable subtype of FSArray creation	`uima.disable_subtype_fsarray_creation` Default: Subtypes of FSArrays can be created and are created when deserializing CASes. UIMA has some limited support for typed arrays. These are declared in type system descriptors by including an elementType specification for a feature whose range is FSArray. See &uima_docs_ref; . The XCAS and the Xmi serialization forms serialize these as FSArray, with no element type specification included in the serialized form. The deserialization code, when deserializing these, looks at the type system's feature declaration to see if it has an elementType, and if so, changes the type of the Feature Structure to that type. UIMA Version 2's CAS API did not have the ability to create typed FSArrays. This was added in V3, but will be disabled if this flag is set. Setting this flag will cause all FSArray creations to be untyped.

Default CASs to support V2 ID references	`uima.default_v2_id_references` In version 3, Feature Structures are managed somewhat differently from V2. * Feature Structure creation doesn't remember a map from the id to the FS, so the LowLevelCas method getFSForRef(int) isn't supported. (Exception: Feature Structures created with the low level API calls are findable using this). * Creation of Feature Structures assign "ids" as incrementing integers. In V2, the "id" is the address of the Feature Structure in the v2 Heap; these ids increment by the size of the Feature Structure on the heap. * Serialization only serializes "reachable" Feature Structures. When this mode is set, the behavior is modified to emulate V2's. * Feature Structures are added to an id-to-featureStructure map. * IDs are assign incrementing by the size of what the Feature Structure would have been in V2. * Serialization includes unreachable Feature Structures (except for Xmi and XCAS - because this is how V2 operates)) This property sets the default value, per CAS, for that CAS's `ll_enableV2IdRefs` mode to true. This mode is is also programmatically settable, which overrides this default. For more details on how this setting operates and interacts with the associated APIs, Preserving V2 ids, with low level CAS Api accessibility

Trading off runtime checks for speed

Title	Property Name & Description
Disabling runtime feature validation	`uima.disable_runtime_feature_validation` Once code is running correctly, you may remove this check for performance reasons by setting this property.

Disabling runtime feature *value* validation	`uima.disable_runtime_feature_value_validation` Default: features being set into FS features which are FSs are checked for proper type subsumption. Once code is running correctly, you may remove this check for performance reasons by setting this property.

Reporting

Title	Property Name & Description
Report feature structure pinning	`uima.report.fs.pinning="nnn"` Default: not enabled; nnn is the maximum number of reports to produce. If nnn is omitted, it defaults to 10. When enabled, this flag will cause reports to System.out with call traces for the first nnn instances of actions which lead to pinning Feature Structures in memory. Typically, this should not happen, and no-longer-reachable Feature Structures are garbage collected. But some operations (such as using the CAS low level APIs, which return integer handles representing Feature Structures) pin the Feature Structures, in case code in the future uses those integer handles to access the Feature Structure. It is recommended that code be improved over time to use JCas access methods, instead of low-level CAS APIs, to avoid pinning unreachable Feature Structures. This report enables finding those parts of the code that are pinning Feature Structures.