UIMA Version 3 User's Guide

Written and maintained by the Apache UIMA™ Development Community

Version 3.0.0-alpha

Publication date January, 2017

Table of Contents

1. Overview 1
    1.1. What's new 1
    1.2. Java 8 is required 3
2. Backwards Compatibility 5
    2.1. JCas and non-JCas APIs 5
    2.2. Serialization forms 5
        2.2.1. Delta CAS Version 2 Binary deserialization not supported 5
    2.3. APIs for creating and modifying Feature Structures 6
    2.4. PEAR support 6
    2.5. toString() 7
    2.6. Type System sharing 7
    2.7. Some checks moved to native Java 7
    2.8. Some class hierarchies have been modified 7
3. New/Extended APIs 9
    3.1. JCas additional static fields 9
    3.2. Java 8 integrations 9
        3.2.1. Built-in UIMA Arrays and Lists integration with Java 8 9
    3.3. UIMA FSIterators improvements 9
    3.4. New Select API 10
    3.5. New custom Java objects in the CAS framework 10
    3.6. Built-in lists and arrays 10
        3.6.1. Built-in lists and arrays have common super classes / interfaces 10
    3.7. Annotation comparator methods 10
    3.8. Reorganized APIs 11
    3.9. Other changes 11
4. Select framework 13
    4.1. Select's use of the builder pattern 13
    4.2. Sources of Feature Structures 13
        4.2.1. Use of Type in selection of sources 15
        4.2.2. Sources and generic typing 15
    4.3. Selection and Ordering 16
        4.3.1. Boolean properties 17
        4.3.2. Configuration for any source 17
        4.3.3. Configuration for any index 17
        4.3.4. Configuration for sort-ordered indexes 18
        4.3.5. Bounded sub-selection within an Annotation Index 18
        4.3.6. Variations in Bounded sub-selection within an Annotation Index 19
        4.3.7. Defaults for bounded selects 20
        4.3.8. Following or Preceding 20
    4.4. Terminal Form actions 20
        4.4.1. Iterators 21
        4.4.2. Arrays and Lists 21
        4.4.3. Single Items 22
        4.4.4. Streams 22
5. CAS Java Objects 23
    5.1. Tutorial example 23
    5.2. Semi-Built-in UIMA Types for Java Objects 25
        5.2.1. FSArrayList 26
        5.2.2. IntegerArrayList 26
        5.2.3. FSHashSet 26
    5.3. Design for reuse 26

6. Migrating JCas 27
6.1. How to migrate 27
6.1.1. Maven v2/v3 simultaneous support 27
6.2. Migrating JCas classes 27
6.2.1. Running the migration tool 28
6.2.2. Understanding the reports 30
6.2.3. Examples 31
7. PEAR support 33
7.1. JCas issues 33
7.2. Custom Java Objects 34
8. Migration aids 35
8.1. Properties Table 35

Chapter 1. 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.

1.1. What's new in UIMA Java SDK version 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 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 built-in UIMA types. These are implemented in a "lazy" style, avoiding extra computation until needed.

FSArrayList

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

IntegerArrayList

a variable length int array. Supports OfInt iterators.

FSHashSet

a Java HashSet 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.

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.

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`.

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()))));
```

1.2. Java 8 is required

The UIMA Java SDK Version 3 requires Java 8 or later.

Chapter 2. Backwards Compatibility

Because users have made substantial investment in developing applications using the UIMA framework, a goal 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 a new set of JCas classes (if these were previously being used). The creation of this new set of JCas classes is mostly automated via a migration tool that handles converting the existing JCas classes, described in a later chapter.

2.1. 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.

New internal-use methods and fields have been added to the JCas classes. The names for these have been carefully designed to reduce the likelihood of collision with previously existing user code names; the usual technique is to start the names with a leading underscore character. Users should consider these methods as internal use and subject to change with new releases.

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.

2.2. 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).

Some formats like Binary and BinaryCompressedForm4 require the type systems match exactly. Version 3 adds some new built-in types, so the type systems won't match exactly; an accommodation is made when the version 3 deserialization detects that the serialized CAS was serialized with version 2, that enables version 3 to load version 2 serialized CASs, even for binary formats. A planned extension for UIMA-AS will allow version 3 clients to detect when a service is running version 2 code, and have the serialization altered to be compatible with version 2 (provided that the new, built-in, types are not being used).

2.2.1. 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 delta CAS is attempted.

2.3. APIs for creating and modifying Feature Structures

There are 3 sets of APIs for creating and modifying Feature Structures.

- 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 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 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 coursse, the CAS is reset).

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

2.4. 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.

2.5. 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.

2.6. Type System sharing

Type System definitions are shared when they are equal. After type systems have been built up from type definitions, and are committed, a check is made to see if an identical type system already exists. 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 type system is already created, then the commit operation returns the already created one, and the one just built is discarded. Normally, this is not an issue. However, some application code (for example, test cases) may construct type systems programmatically, and along the way save references to defined types and features. These references can then become invalid when the type system is created and perhaps replaced with an already existing one.

Application code may code around this by re-acquiring references to type and feature objects, if the type system returned from `commit` is not identical (==) to the one being committed.

2.7. 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.

2.8. 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.

Chapter 3. New and Extended APIs

3.1. JCas additional static fields

(Also in UIMA Version 2 after release 2.10.0) 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.

3.2. Java 8 integrations

Several of the the JCas cover classes provide additional integrations with Java 8 facilities.

3.2.1. Built-in UIMA Arrays and Lists integration with Java 8

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

The built-in collection 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.

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

3.3. UIMA FSIterators improvements

To enable more seamless integration with popular Java idioms, the UIMA iterators for iterating over UIMA Indexes (the FSIterator interface) now implements the Java ListIterator Interface.

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 context, to mean that within a single thread, while an iterator is active, some modifications are being done 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).

3.4. 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.

3.5. 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. The following chapter describes this facility, and some new semi-built-in types that make use of it.

3.6. Built-in lists and arrays

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

For lists, new static methods `getEmptyList(JCas jcas)` on each of the 4 kinds of built-in lists (FS, Integer, Float, and String) retrieve a shared, common instance of the EmptyXxxList for a CAS.

For lists, a new `push(item)` API on an existing list node creates a new non-empty node, sets its head to `item` and its tail to the existing list node. This allows easy construction of a list in backwards order. There is also a `pushNode()` which just creates and links in a new node to the front of the list. And finally, there's a `createNonEmptyNode()`, which just creates a node of the same type, in the same CAS, without linking it.

3.6.1. 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.

3.7. Annotation comparator methods

The built-in type Annotation has 4 new methods to allow comparing two annotations. The first method (`compareAnnotation`) uses the standard annotation comparator (two keys:

begin (ascending) and end (descending)). A second method (`compareAnnotation(other, linear_type_order)`) adds a 3rd comparison, used if the Annotations compare equal), which uses a linear_type_order to compare the two types. Another two methods extend these two methods with an additional key - the Annotation's ID, used only if the previous comparsions are all equal. All of these return the standard Java compare result allowing discrimination between equal, >, and <.

3.8. 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.

3.9. Other changes

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`.

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.

Chapter 4. The select framework for working with CAS data

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

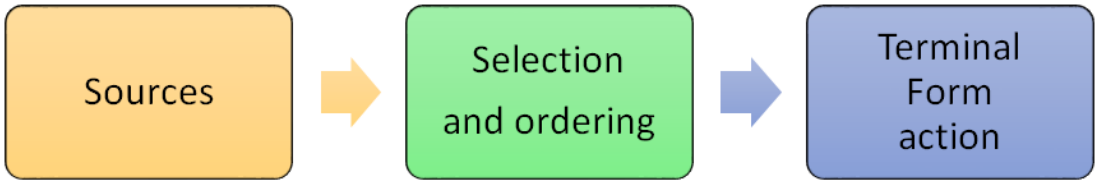

Figure 4.1. Select - the big picture

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.

4.1. 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.

4.2. 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 built-in types `FSArrayList` and `FSHashSet`; user-defined JCas classes for user types may also choose to implement this. All of these sources may be used with `select`.

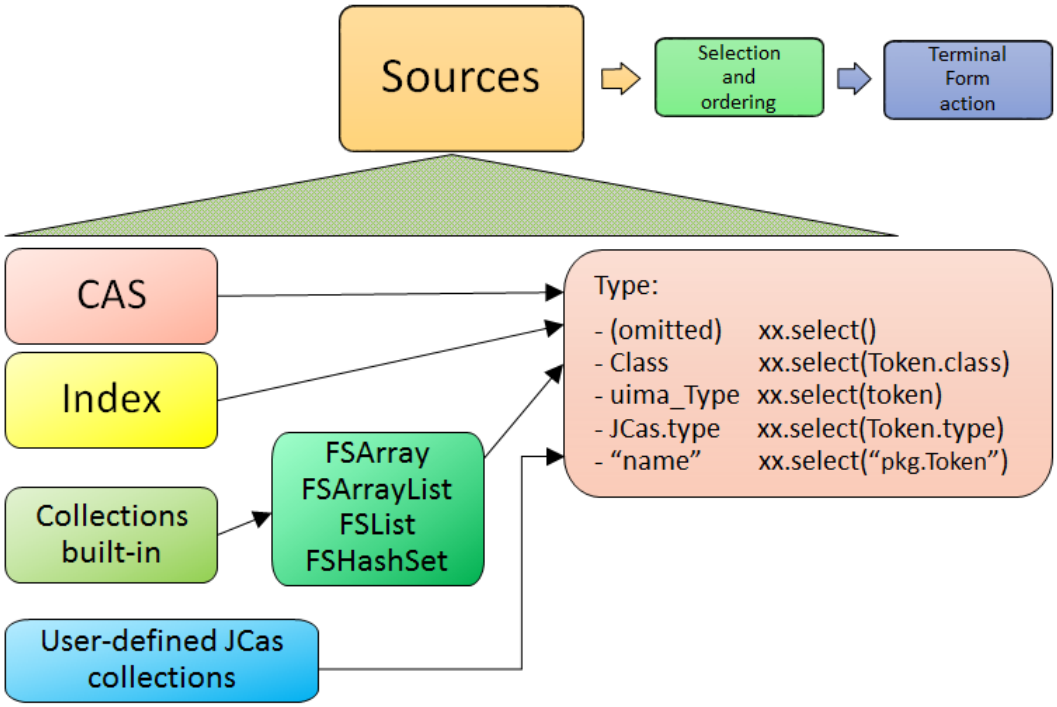

Figure 4.2. select method with type

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 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.

4.2.1. 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.

4.2.2. 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.

```
// this works
// the generic type for Token is passed as an argument to select
FSIterator<Token> token_it = cas.select(Token.class).fsIterator();

FSIndex<Token> token_index = ... ; // generically typed

// 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 = token_index.select().fsIterator();

// You can overcome this in two ways:
// pass in the type as an argument to select
// using the JCas cover type.
FSIterator<Token> token_iterator =
    token_index.select(Token.class).fsIterator();

// You can also use the static form of select
// to avoid repeating the type information
FSIterator<Token> token_iterator =
    SelectFSs.select(token_index).fsIterator();

// Finally, you can also explicitly set the generic type
// that select() should use, like a special kind of type cast, like this:
FSIterator<Token> token_iterator =
    token_index.<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 Sentences (and their subtypes):

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

4.3. 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.

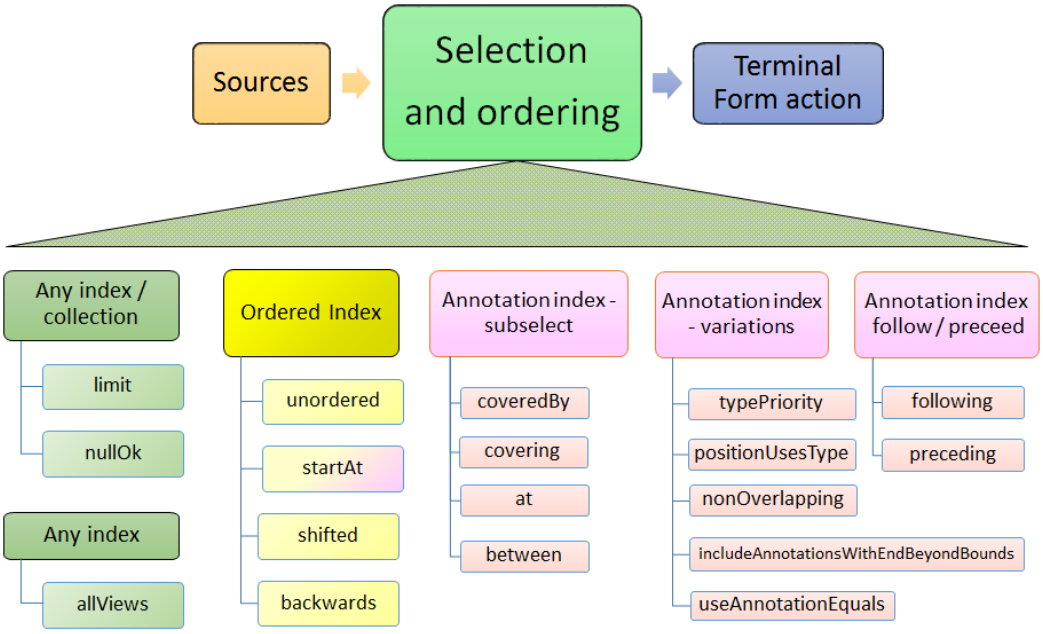

Figure 4.3. Selection and Ordering

4.3.1. 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.

4.3.2. 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 some terminal_form actions, which would otherwise throw an exception if a null result happened.

4.3.3. 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.

4.3.4. 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(xxx)` takes two forms, each of which has, in turn 2 subforms. The form using `begin, end` is only valid for Annotation Indexes.

```
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


// the next two forms are only valid for AnnotationIndex sources

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
```

backwards
specifies a backwards order (from last to first position) for subsequent operations

4.3.5. 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) within the span of another, bounding Annotation, such as 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 (or are equal to) 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. 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`.

4.3.6. 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.

positionUsesType
When type priorities are not being used, Annotations with the same begin and end and type will be together in the index. The starting position, when there are many Annotations which might compare equal, is the left-most (earliest) one of these. In this comparison for equality, by default, the `type` of the bounding Annotation is ignored; only its begin and end values are used. If you want to include the type of the bounding Annotation in the equal comparison, set this to true.

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.

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

When this variant is specified, in addition to that, any Annotation which has the same begin, end, and (maybe) type is also skipped. The `positionUsesType` setting is used to specify in this variant whether or not the type is included when doing the equals test. Note that `typePriority` implies `positionUsesType`.

4.3.7. 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: type priorites are not used when determining bounds in bounded selects. Subiterators, in contrast, use type priorities.

positionUsesType
default: the type of the bounding Annotation is ignored when determining bounds in bounded selects; only its begin and end position are used

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.

useAnnotationEquals
default: only the single Annotation with the same _id() is skipped when doing sub selecting. Use this setting to expand the set of skipped Annotations to include all those equal to the bound's begin and end (and maybe, type, if positionUsesType or typePriority specified).

4.3.8. Following or Preceding

For an sorted Index, you can specify all Feature Structures following or preceding a position. The position can be specified either as a Feature Structure, or (for AnnotationIndexes only) by using begin and end values. The arguments are identical to those of the `startAt` specification, but are interpreted differently.

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

preceding
Position the iterator according to the argument, save that Annotation's `begin` value, and then move it backwards until the Annotation's (at that position) `end` value is <= to the saved `begin`value.

4.4. 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.

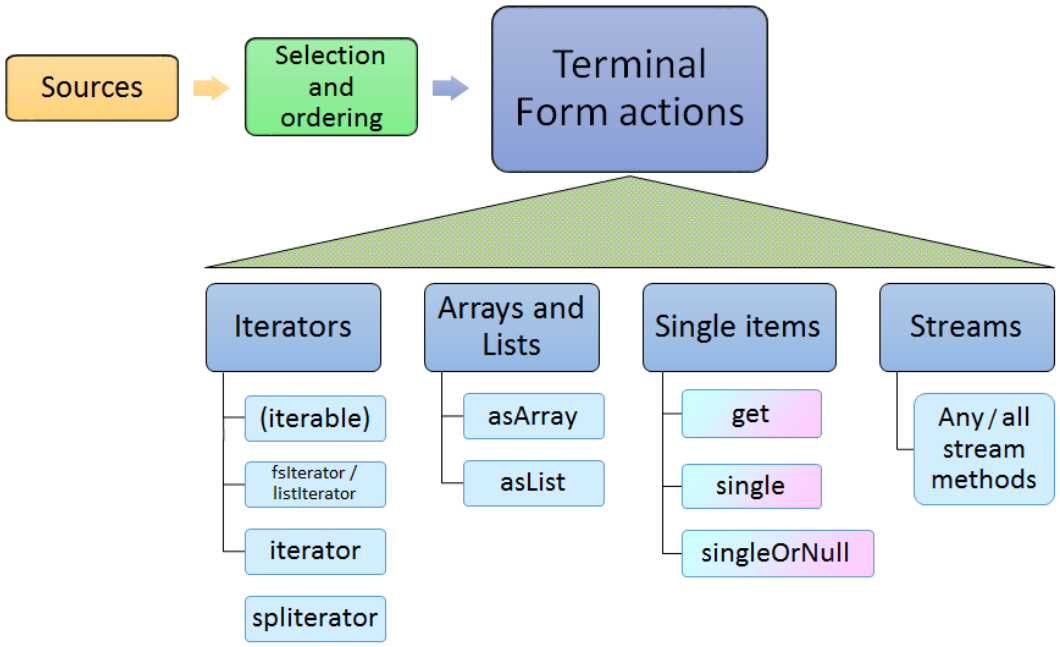

Figure 4.4. Select Terminal Form Actions

4.4.1. 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.

4.4.2. 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.

4.4.3. 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.

> **Note:** 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, or null. If nullOk(false) is configured, then if the result is null, an exception will be thrown.

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.

4.4.4. Streams

any stream method
Select supports all the stream methods. The first occurance 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:

```
Set<Type> foundTypes =
    // items of MyType or subtypes
   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)));
```

Chapter 5. Defining 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.

5.1. 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.

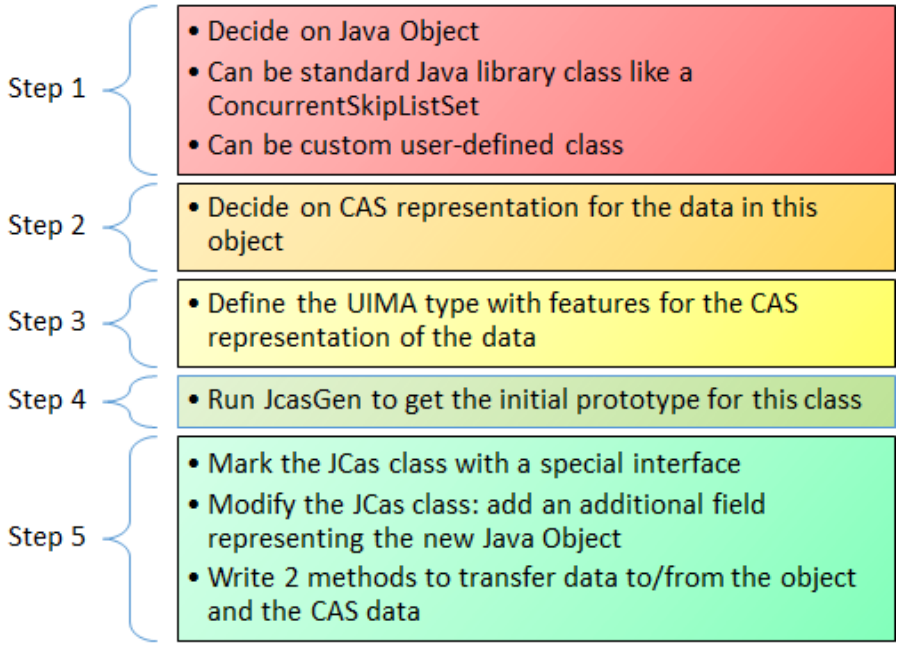

Figure 5.1. Creating a custom Java CAS-stored Object

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.

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.

Add the Java Object as a field to the class

We'll define a new field:

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

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

5.2. 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.)

5.2.1. FSArrayList

This 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.

5.2.2. IntegerArrayList

This 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.

5.2.3. FSHashSet

This type stores Feature Structures in a 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.

5.3. 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.

Chapter 6. Migrating JCas classes from Version 2 to 3

6.1. 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 two 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.)

Some Maven projects use the JCasGen maven plugin; these project's JCasGen maven plugin, if switched to UIMA V3, automatically generate the V3 versions.

You can use any of the methods of invoking JCasGen to generate the new V3 versions. If you already have the source or class files, you can also migrate those using the migration tool described in this section.

For simple testing, after users have migrated or generated any user-supplied JCas classes, users typically will JAR-up these migrated classes, and then place that JAR in the class path ahead of the original V2 pipeline's classes so these will replace the corresponding classes.

6.1.1. How to modify a Maven project to support both v2 and v3

When migrating an existing Maven project to v3, you can of course, just change the dependencies, upgrade to Java 8 if needed, and migrate/regenerate any user-defined JCas class definitions. Nothing special is needed for this approach.

If you want to simultaneously support both v2 and v3, you can take advantage of the binary compatibility support in v3, and have one set of maven artifacts for both (compiled with Java 7), plus an additional Maven project just for the generated JCas classes v3-style. This additional maven project would specify Java 8 and UIMA v3 dependencies, and produce a Jar file with the V3 version, either from migrated Java sources, or using the JCasGen maven plugin, generating from the type system description.

The using community would see the original output artifacts, plus one new Jar artifact, which, if they were wanting to use V3, they would need to place ahead of the original artifacts in the running pipeline's classpath.

6.2. Migrating JCas classes

Some projects are built using Maven, and use the JCasGen Maven plugin as part of the build. These builds, when switched to depend on version 3, will automatically generate the new JCas class definitions, so nothing special need be done.

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 an Eclipse launch configuration.

This tool is run against a directory tree, and scans all the files in the tree, looking for JCas cover classes. It can process either source files, or, if those are not available, compiled class files. It can process these inside directories, as well as inside Jar files within those directories.

Run this tool using a Java 8 JDK (as opposed to a Java JRE) in order to have the Java compiler available.

The results of the migration include the migrated class files and a set of logs, 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, update your UIMA application to use these classes in place of the version 2 JCas classes.

The migration tool is able to scan multiple directory trees, looking for existing Java source (.java) or class (.class) files, even inside Jars and PEAR archives. If compiled class files are used as input, a Java decompiler is used to decompile these to source. For PEARs and Jars, it replaces the migrated and compiled classes in copies of the PEARs and Jars.

The actual migration step is a source-to-source transformation, done using a Java parse of the source files. The parts in the source which are version 2 specific 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. Detailed reports detailing any issues encountered are written to log files.

> **Note:** The compilation step (done after the source-to-source transformation) requires that a Java compiler is available (which will be the case if you're running with a Java JDK, not a JRE).

> **Note:** 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.

6.2.1. 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 source files, the other for compiled classes), which are available if you have the uimaj-examples project (included in the binary distribution of UIMA) in your Eclipse workspace.

6.2.1.1. 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 classes roots, the classes must not have compile errors, and may contain Jars and PEARs.

To use these launchers,

- First select the eclipse project; this project's "build path" will supply the classpath used during migration for decompiling and compiling the sources.
- run the launcher - it will prompt for two values:
  - the root directory to scan recursively looking for JCas sources or classes/Jars/Pears to be converted
  - an output directory - a writable folder, such as /temp/migrate

No matter how the tool is started, the input to the tool is one or more directory tree roots; the tool will scan the file system under these roots (including inside Jars and PEARs when running from compiled classes), looking for definitions to migrate.

6.2.1.2. 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).

  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).

Command line: Specifying a classpath for the migration

A classpath is required for the proper operation of the decompiling and compiling steps of the migration. This is provided using 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 PEAR classes to the classpath.

6.2.1.3. 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 detects this, and migrates all non-identical instances, using a convention to store them in the output directory in a manner where different versions can be conveniently compared using tooling such as Eclipse's file compare.

> When there are non-identical duplicate definitions, the user must manually compare these and decide which version to use.

6.2.2. 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.

> These are arranged in parallel directories, allowing Eclipse's multi-file directory "compare" to work on both directory collections and conveniently show for all the migrated files the change details.

In the case of non-identical duplicates, an increasing integer starting with 1 is inserted into the output directory tree for each migrated class.

The overall directory output directory tree looks like:

```
Directory structure, starting at -outputDirectory
   converted/
     v2/
       x/y/z/javapath/.../Classname.java
       x/y/z/javapath/.../Classname.java
       ...
     v3/
       x/y/z/javapath/.../Classname.java
       x/y/z/javapath/.../Classname.java
       ...


       1/     << for non identical duplicates
         x/y/z/javapath/.../Classname.java
         x/y/z/javapath/.../Classname.java
         ...
       2/     << for non identical duplicates
         x/y/z/javapath/.../Classname.java
         x/y/z/javapath/.../Classname.java
         ...
     v3-classes/
         ...

   not-converted/
   logs/
     processed.txt
     failed.txt
     skippedBuiltins.txt
     NonJCasFiles.txt
     workaroundDir.txt
     deletedCheckModified.txt
     manualInspection.txt
     pearsFileUpdates.txt
     jarsFileUpdates.txt
   pears/
     xyz_converted_pear.pear
     ...
   jars/
     ...
```

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.

workaroundDir.txt
When running conversions on a windows system for files from a linux system, sometimes there is a clash caused by the fact that Windows doesn't recognize upper vs lower case in file names. When this happens, an entry is logged here, and the conflicting name is suffixed with a "_c".

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.

6.2.3. 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:

```
Load Eclipse workspace containing the project to be migrated,
  and the version 3 uimaj-migration project.

Select the Java project having the classpath used when running.

Eclipse -> menu -> Run -> Run configurations
```

```
  Use the search box to find "Migrate JCas" launcher, and launch that.

In the first prompt,
  enter the path to the root of the compiled classes or Jar or PEAR
In the second prompt,
  enter a path to where you want the output, e.g. /temp/migrateJCas
```

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

Chapter 7. 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).

7.1. 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 built-in 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.

7.2. 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).

The new built-in types that reference Feature Structures *do not* include the extra facility to support this conversion; they store whatever they're given. This means that they may be used inside or outside a PEAR, but cannot be used to hold onto Feature Structures set in one context, but referenced in the other.

If there is interest or need, additional documentation can be written describing how the core framework handles this, and providing a cookbook for others to implement this capability for custom Java objects.

Chapter 8. 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.

8.1. 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**	
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 finding all Feature Structures by their int ID	uima.enable_id_to_feature_structure_map_for_all_fss Default: normally created Feature Structures are not kept in a map. In version 3, normally, Feature Structures are not added to the map used by the Low Level CAS API to map from int ids to Feature Structures. This has the benefit that no longer referenced Feature Structures may be garbaged collected. This behavior may be overridden by this property.	
Trading off runtime checks for speed		
Disabling runtime feature validation	uima.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.	