

UIMA Version 3 User's Guide

**Written and maintained by the Apache
UIMA™ Development Community**

Version 3.0.0-SNAPSHOT

Copyright © 2006, 2016 The Apache Software Foundation

Copyright © 2004, 2006 International Business Machines Corporation

License and Disclaimer. The ASF licenses this documentation to you under the Apache License, Version 2.0 (the "License"); you may not use this documentation except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, this documentation and its contents are distributed under the License on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Trademarks. All terms mentioned in the text that are known to be trademarks or service marks have been appropriately capitalized. Use of such terms in this book should not be regarded as affecting the validity of the the trademark or service mark.

Publication date October, 2016

Table of Contents

1. Select framework	1
1.1. Select's use of the builder pattern	1
1.2. Sources of Feature Structures	1
1.2.1. Sources and generic typing	3
1.3. Selection and Ordering	3
1.3.1. Boolean properties	4
1.3.2. Configuration for any source	4
1.3.3. Configuration for any index	5
1.3.4. Configuration for ordered indexes	5
1.3.5. Bounded sub-selection within an Annotation Index	5
1.3.6. Variations in Bounded sub-selection within an Annotation Index	6
1.3.7. Defaults for bounded selects	7
1.3.8. Following or Preceding	7
1.4. Processing actions	8
1.4.1. Iterators	8
1.4.2. Arrays and Lists	9
1.4.3. Single Items	9
1.4.4. Streams	9

Chapter 1. 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

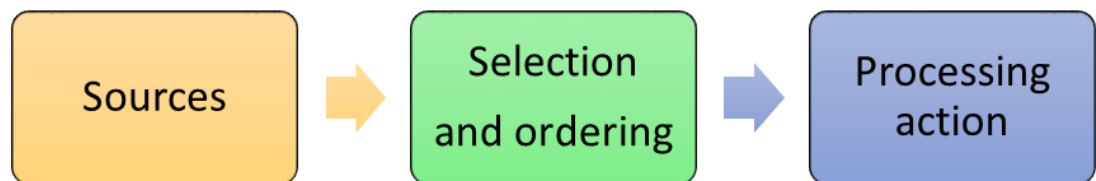


Figure 1.1. Select - the big picture

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.

1.1. 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.

1.2. Sources of Feature Structures

Feature Structures are kept in the CAS, and are accessed using UIMA Indexes. Feature Structures may, additionally, be kept in `FSArrays` and `FSLists`. All three of these sources may be used with `select`.

For CAS sources, there are separate sets of indexes per CAS view; a typical CAS source is the Feature Structures belonging to a particular index, in a particular CAS view, perhaps filtered by UIMA Type.

You can omit the index; in that case, the default is

- all Feature Structures in a Cas View, or
- (if the selection and ordering specifications require an AnnotationIndex), all Feature Structures in the view's AnnotationIndex.

You can extend any selection to be an aggregation over all the CAS views.

A UIMA index is the usual source. If a CAS is used, all Feature Structures that were added to the index in the specified CAS view are used as the source. The FSArray and FSList sources have more limited configurability, because they are considered non-sorted, and therefore cannot be used for an operations which require a sorted order.

There are 4 sources of Feature Structures supported:

- a CAS view: all the FSs that were added to the indexes for this view (in arbitrary order)
- an Index over a CAS view
- Feature Structures in a UIMA FSArray instance
- Feature Structures in a UIMA FSList

Each of these sources has a new API method, `select(...)`, which initiates the select specification. The select method can take an optional positional parameter, specifying the UIMA type to return.

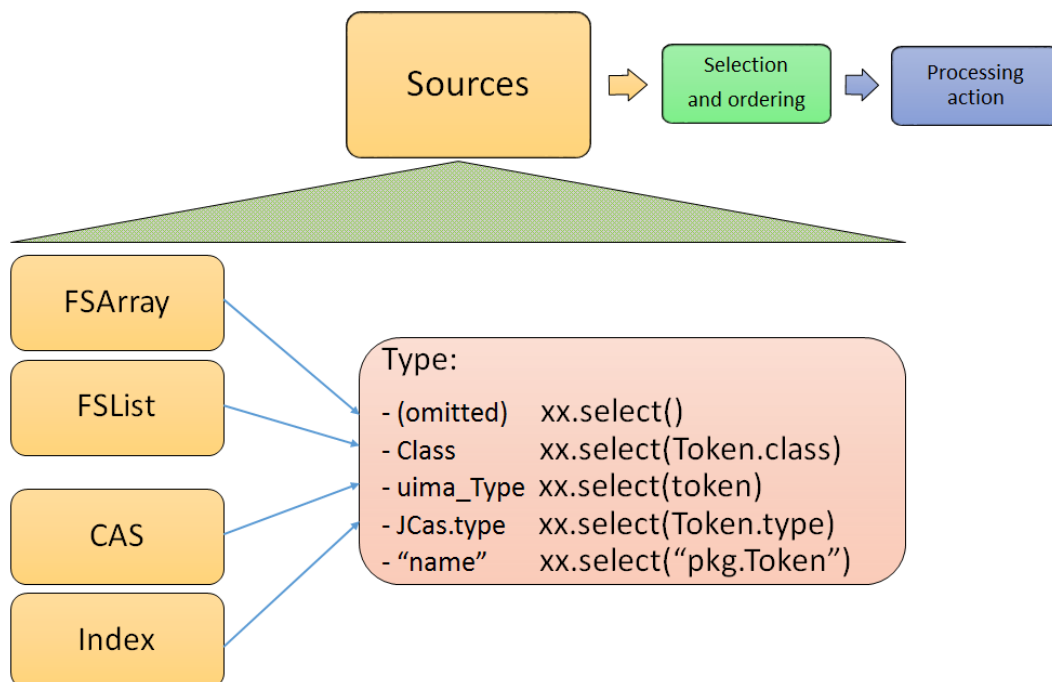


Figure 1.2. *select* method with type

A UIMA index is the usual source. If a CAS is used, all Feature Structures that were added to the index in the specified CAS view are used as the source. The FSArray and FSList sources have more limited configurability, because they are considered non-sorted, and therefore cannot be used for an operations which require a sorted order.

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 the TOP type is used (meaning all types).

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.

1.2.1. 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.

A static version of the `select` method (named `sselect`) gets around this by providing the generically typed information as an argument, rather than having it come from the receiver.

```
// 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:
// explicitly set the generic type select() should use, like this:
FSIterator<Token> token_iterator =
    token_index.<Token>select().fsIterator();

// You can also use the static form of select
FSIterator<Token> token_iterator = sselect(token_index).fsIterator();
// Java makes use of the generic information from the index,
// coming in as an argument
```

The `sselect` method may be statically imported into code that uses it, to avoid repeatedly qualifying this with its class, `SelectFSs`.

1.3. 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.

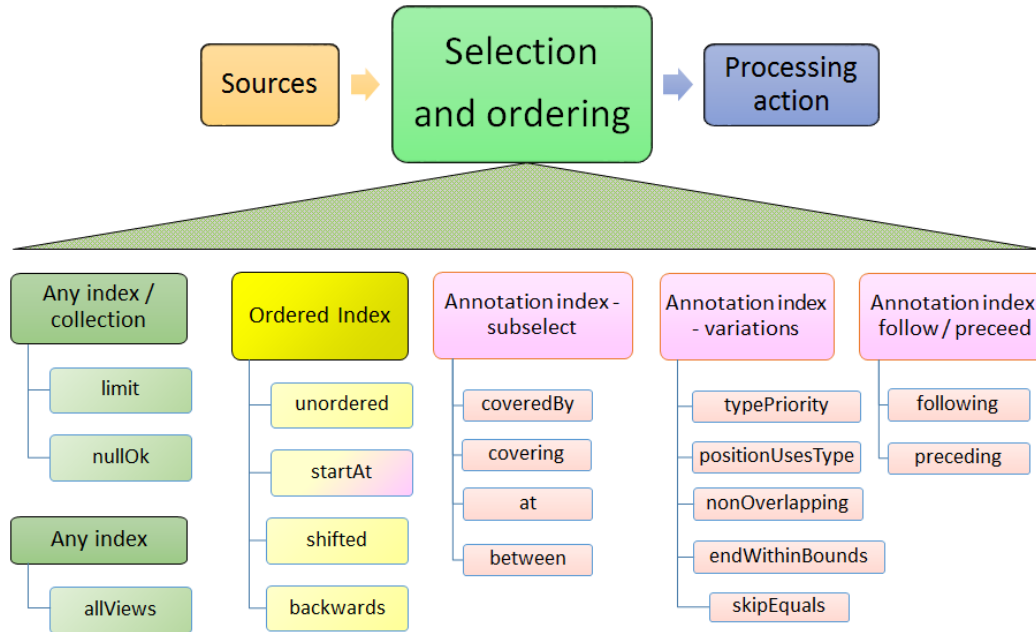


Figure 1.3. Selection and Ordering

1.3.1. 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.

1.3.2. 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 processing actions, which would otherwise throw an exception if a null result happened.

1.3.3. 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, in no particular order, of the underlying selections, one per view in the CAS. Because of this implementation, the items in the selection may not be unique - that is a single Feature Structure may be in multiple views.

1.3.4. Configuration for ordered indexes

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

unordered

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

1.3.5. Bounded sub-selection within an Annotation Index

When selecting Feature Structures to process, frequently you may want to select only those which have a relation to a bounding Feature Structure. A commonly done selection is to select all Feature Structures (of a particular type) within the span of another, bounding Feature Structure, 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 Feature Structure (except the `between` which is based on two bounding Feature Structures).

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

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

The bounding information is specified either as an Annotation Feature Structure (or a subtype of Annotation), or the `begin` and `end` can be directly specified.

The returned Feature Structures 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 Feature Structures within the bound

covering

iterates over Feature Structures that span (or are equal to) the bound.

at

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

between

uses two feature structures, and returns Feature Structures 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 Feature Structure's `begin` has to be \geq the earlier bound's `end`, and the Feature Structure's `end` has to be \leq the later bound's `begin`.

1.3.6. 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 Feature Structures 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

This is also called *unambiguous* iteration. If specified, then after the iterator reaches a position, the `moveToNext()` operation moves to the next Annotation which has a `begin` offset \geq to

the previous Annotation's end position. 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.

endWithinBounds

This is also called *strict*. For `coveredBy` selection, if specified, then any Annotations whose end position is > the end position of the bounding Annotation is skipped. The `between` selection always behaves as if this is set. This variant is ignored for `covering` selection.

skipEquals

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

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

positionUsesType

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

nonOverlapping

default: this mode is ignored. It corresponds to the "unambiguous" mode in Subiterators, so the default is "ambiguous".

endWithinBounds

default: this mode is ignored. In any case, it only is used for `coveredBy` selections; the other subselect operations ignore it. This corresponds to Subiterator's "strict" option, so the default is "not strict".

skipEquals

default: only the single Feature Structure with the same `_id()` is skipped when doing subselecting. Subiterators, in contrast, skip all Feature Structures which compare equal using the `AnnotationIndex` comparator.

1.3.8. Following or Preceding

For an Annotation Index, you can specify all Annotations following or preceding a position. The position can be specified either as a Annotation, or by using begin and end values. The arguments are identical to those of the `startAt` specification, but are interpreted differently.

The underlying iteration can be any of the kinds supported by the Annotation Index, except that `endWithinBounds` is forced on.

following

Position the iterator according to the argument, get that Feature Structure's `end` value, and then move the iterator forwards until the Annotation at that position has its `begin` value \geq 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 \leq to the saved `beginvalue`.

1.4. Processing actions

After the sources and selection and ordering options have been specified, one processing action may be specified. This can be an iterator, something that converts to an array or list, something that retrieves a single value with various extra checks, or a stream operation. A stream operation converts the object to a stream; from that point on, any stream operation may be used.

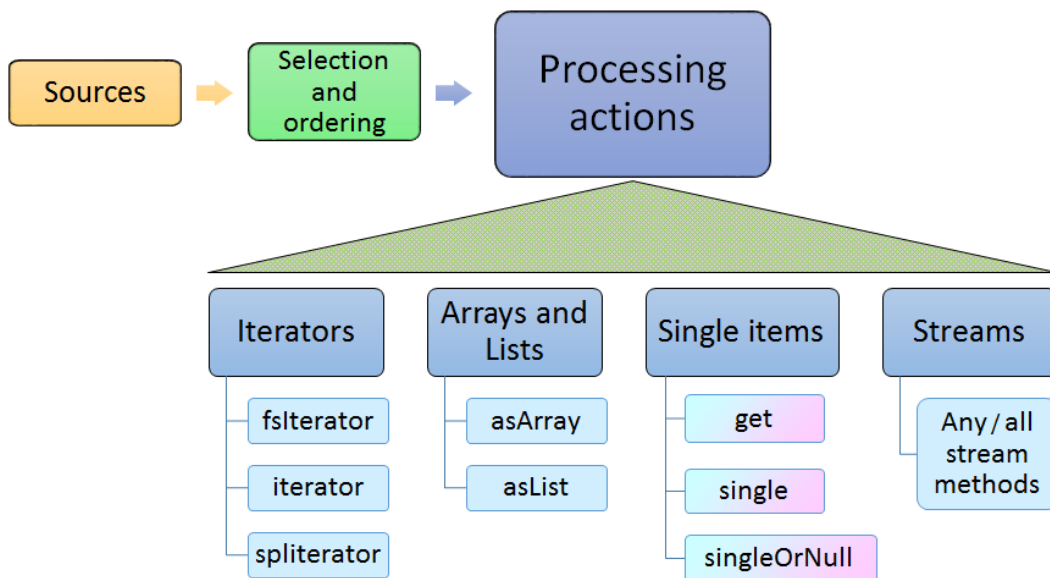


Figure 1.4. Select Processing Actions

1.4.1. Iterators

fsIterator

returns a configured `fsIterator` or `subIterator`. This iterator implements `ListIterator` as well. 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.

1.4.2. 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.

1.4.3. 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 Feature Structure or `begin` and `end` require an Annotation Index.

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.

1.4.4. Streams

any stream method

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

