UV3 Iterator support

This page describes additional UIMA Version 3 iterator support for accessing Feature Structures. It draws on the uimaFIT design, with these goals:

- uniformity (ease of learning)
- conciseness (in expression)
- alignment with / exploitation of Java 8 styles, including streams
- hiding implementation internals

PDF of chapter on select documentation for review

This is the version 2 after incorporating most of Richard's suggestions on version 1, uploaded Monday 17 Oct at 3:45 EDT.

Here: 

Conceptual overview MS

What gets iterated over

FSs, of some type, usually including all subtypes

Source

	Examples
FSs in a CAS - usually restricted to one view	v. myView. cas.
FSs in an FSArray, or in an FSList	fsa. myList.
FSs in an fsIndex	

Top Method: "select()"

some select() alternative
may include positional args

- v.select()
- myList.select()
- myIndex.select()

How Results Manifested

FSIterator - forwards / backwards - moveTo Fs	.fsIterator()	
Java Iterator	.iterator()	
Java List -- supports size(), for (xx : yy), ListIterator	.asList()	list allows forward and reverse iteration
Java Spliterator	.spliterator()	
Java Stream (augmented)	(is default)	
singleton	get()	the FS, throws if more than 1

Modifiers - what gets iterated over, what index is used

for CAS source: all CAS views	.allViews()	invalid for AnnotationIndex
for CAS source: named index - if omitted, -- all FSs for type if type is subtype of annotation, ---- annotation index -- if operation needs Annotation --- annotationindex or subtype of it (if type is specified)	.index("index name") .index(fsIndex)	
type - if omitted, -- Annotation for AnnotationIndex or if FSArray/List is over subtype of Annotation or operation needs Annotation -- TOP	.type(MyClass.class) .type(MyClass.type) .type(myUimaType) .type(string)	string is name of type, can be short name if unambiguous

Modifiers - how the iteration proceeds

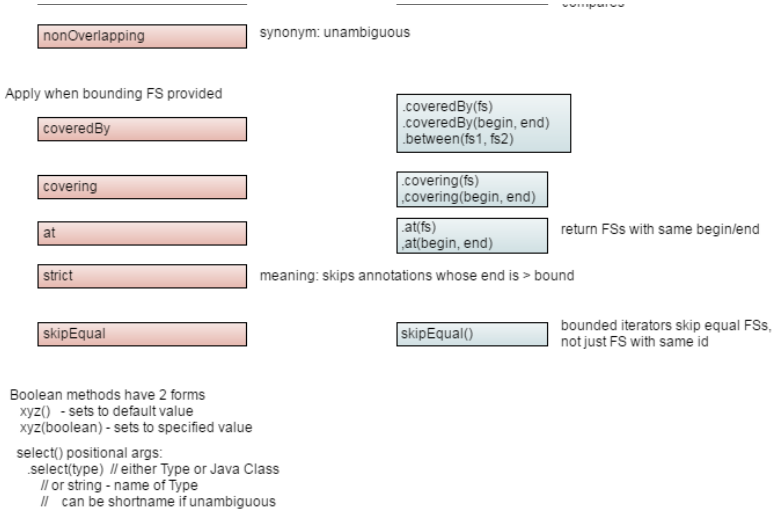
snapshot	.snapshot()	
matchType	.matchType()	meaning: only return FS whose type is the same as the specified type, no subtypes

Apply to ordered

unordered		meaning: for ordered indexes, can use non-rattling (more efficient) iterators
backwards		
startAt	.startAt(fs)	meaning: does a moveTo operation
	.startAt(begin, end)	
	.startAt(fs, +- offset) .startAt(begin, end, +- offset)	

Apply to Annotations

typePriority	.typePriority()	
positionUsesType	.positionUsesType()	when typePriorites not used, says to use types == in compares



Translations of uimaFIT select methods to the new API

Format: the samples not in a box are uimaFIT styles; in the boxes are possible new API styles (sometimes multiples), with some commentary.

select

Convenience method to iterator over all features structures of a given type.

select(JCas, Class<T>)

```
jcas.select() // selects an implicit index,
              // starts with top level default bag index
              // switches to AnnotationIndex if any subsequent build operations imply needing this
              // switches to explicitly specified index if subsequently specified
              // can subsequently specify a top-level type (must be the type or a subtype of the type defined
in the index)

jcas.select(Token.class)  // note: in UV3, these should also work with cas (not just with jcas)?
                          //       We're trying to reduce the distinction between these because it's less
needed in UV3.
jcas.select(Token.type)
jcas.select("my.package.Token")

for (Token.class : jcas.select(Token.class)) {
  ...
}

jcas.select(Token.class).forEach( t -> System.out.println(t) );
jcas.select(Token.class).forEach(System.out::println);
```

select(FSArray, Class<T>)
select(FSList, Class<T>

```
myArray.select(Token.class)  // select also on arrays and fslists
myList.select(Token.class)
```

myIndex.select(FSList, Class<T>

```
myIndex.select(Token.class)  // select also works on uima indexes
```

selectAll

Convenience method to iterate over all features structures indexed in a particular index in a particular view.
(.allViews()) not part of uimaFIT).

selectAll(JCas)

```
jcas.select() // leave off type specification
```

selectAt

Get all annotations of the given type at the specified offsets, i.e. all annotations with exactly the given start and end offset.

MS: Does this mean covered-by (limit the FSs returned) or startAt - no limit?

REC: It does not mean covered-by because only annotations at the exact specified offset are returned. If there are multiple annotations at the given offset, then all of the specified type are returned.

selectAt(JCas, Class<T>, int, int)

```
jcas.select(Token.class).at(1, 3)
```

selectBetween

Get a list of annotations of the given annotation type located between two annotations. Does not use subiterators and does not respect type priorities. Zero-width annotations what lie on the borders are included in the result, e.g. if the boundary annotations are [1..2] and [2..3] then an annotation [2..2] is returned. If there is a non-zero overlap between the boundary annotations, the result is empty. The method properly handles cases where the second boundary annotations occurs before the first boundary annotation by switching their roles.

selectBetween(Class<T>, AnnotationFS, AnnotationFS)

Example same as **selectBetween(JCas, Class<T>, AnnotationFS, AnnotationFS)** below unless we introduce a static method.

selectBetween(JCas, Class<T>, AnnotationFS, AnnotationFS)

```
jcas.select(Token.class).between(fs1, fs2) // might make more sense because we are using offsets, not index
positions
jcas.select(Token.class).startAt(fs1).endAt(fs2) // this looks more like index positions...
```

selectByIndex

selectByIndex(JCas, Class<T>, int)

```
jcas.select(Token.class).skip(indexPosition)  // skip is standard stream operator
```

selectCovered

Get a list of annotations of the given annotation type constrained by a 'covering' annotation. Iterates over all annotations of the given type to find the covered annotations. Does not use subiterators.

The covering annotation is never returned itself, even if it is of the queried-for type or a subtype of that type.

MS: Not sure why this is special-cased?

REC: Because the covering annotation itself would always be included in the result and it is almost never needed. So handling it specifically (since we already know it) is easier than tediously filtering it out from the result in 99.9% of the cases.

MS: Good point. Just for completeness, I'll point out a somewhat surprising fact that the covering annotation is not always included; it is only returned if

- it is in the index
- it's type is the index's type or subtype.
- You can make annotations that don't fit these criteria, and use them as "covering" spec.

selectCovered(Class<T>, AnnotationFS)

Same as **selectCovered(JCas, Class<T>, AnnotationFS)** below unless we introduce a static method.

selectCovered(JCas, Class<T>, AnnotationFS)

```
jcas.select(Token.class).within(contextAnnotation) // within is variant of coveredby
```

selectCovered(JCas, Class<T>, int, int)

```
jcas.select(Token.class).within(1,3)
```

selectCovering

Get a list of annotations of the given annotation type constraint by a certain annotation. Iterates over all annotations to find the covering annotations.

selectCovering(Class<T>, AnnotationFS)

Same as **selectCovering(JCas, Class<T>, AnnotationFS)** below unless we introduce a static method.

selectCovering(JCas, Class<T>, AnnotationFS)

```
 jcas.select(Token.class).containing(contextAnnotation) // containing is variant of covering
```

selectCovering(JCas, Class<T>, int, int)

```
 jcas.select(Token.class).containing(1,3)
```

selectFollowing

Returns the n annotations following the given annotation.

selectFollowing(Class<T>, AnnotationFS, int)

Same as **selectFollowing(JCas, Class<T>, AnnotationFS, int)** below unless we introduce a static method.

selectFollowing(JCas, Class<T>, AnnotationFS, int)

```
// Initial brainstorming
jcas.select(Token.class).startAt(contextAnnotation).limit(10) // it is IMHO not entirely clear here that
contextAnnotation is not included in the result...
jcas.select(Token.class).seek(contextAnnotation).skip(1).limit(10) // somehow too complex...
jcas.select(Token.class).following(contextAnnotation).limit(10)

// Suggestion MS - deleted - I like REC's better with the fs first.


// Suggestion REC 1
jcas.select().following(fs, 10)  // the 10 FSs >= fs
jcas.select().following(fs, 10, -3) // the 10 FSs >= { fs , after a -3 offset }
jcas.select().following(2, 20, 10, -3) // the 10 FSs >= { a bounding FS with begin=10, end=100, offset by -3}
jcas.select().preceding(10, 100, 3)  // the 3 FSs < a bounding FS with begin=10, end=100, in reverse order(?)
// Suggestion REC 2
jcas.at(fs).select(Token.class).following(10)  // the 10 FSs >= fs
jcas.at(fs).select(Token.class).skip(-3).following(10) // the 10 FSs >= { fs , after a -3 offset }
jcas.at(10, 100).select(Token.class).preceding(3)  // the 3 FSs < a bounding FS with begin=10, end=100, in
reverse order(?)
// MS: following(10) is the same as the standard stream method limit(10), I think.
// Suggestion REC 3
jcas.select(Token.class, fs).following(10)  // the 10 FSs >= fs
jcas.select(Token.class, fs).skip(-3).following(10) // the 10 FSs >= { fs , after a -3 offset }
jcas.select(Token.class, 10, 100).preceding(3)  // the 3 FSs < a bounding FS with begin=10, end=100, in reverse
order(?)
// MS: basic idea: extend select syntax, with 2nd positional argument, representing a position or a bound
```

REC: I would put the context to the front and the conditions to the back...

selectPreceding

Returns the n annotations preceding the given annotation.

selectPreceding(Class<T>, AnnotationFS, int)

Same as **selectPreceding(JCas, Class<T>, AnnotationFS, int)** below unless we introduce a static method.

selectPreceding(JCas, Class<T>, AnnotationFS, int)

```
jcas.select(Token.class).startAt(contextAnnotation).reverse().limit(10)
jcas.select(Token.class).startAt(contextAnnotation).limit(-10) // probably not because limit is standard stream
method
                                                               // throws exception on negative arg
```

selectSingle

Get the single instance of the specified type from the JCas.

selectSingle(JCas, Class<T>)

```
jcas.select(DocumentMetaData.class).single()
jcas.select(DocumentMetaData.class).single().getDocumentUri()
jcas.select(DocumentMetaData.type).get()  // alternative name
jcas.select(DocumentMetaData.type).get(1) // arg is offset
```

selectSingleAt

Get a single annotations of the given type at the specified offsets.

selectSingleAt(JCas, Class<T>, int, int)

```
 jcas.select(Token.class).at(begin, end).single()
 jcas.select(Token.class).at(begin, end).get()
```

selectSingleRelative

Return an annotation preceding or following of a given reference annotation.

selectSingleRelative(Class<T>, AnnotationFS, int)

Same as **selectSingleRelative(JCas, Class<T>, AnnotationFS, int)** below unless we introduce a static method.

selectSingleRelative(JCas, Class<T>, AnnotationFS, int)

```
jcas.select(Token.class).startAt(contextAnnotation).skip(10).single()
jcas.select(Token.class).startAt(contextAnnotation).reverse().skip(10).single()
jcas.select(Token.class).startAt(contextAnnotation).skip(-10).single()
jcas.select(Token.type).at(contextAnnotation, -10),get();
```