Apache UIMA, Part 2

PU Tools, Ressourcen, Infrastruktur

Nils Reiter,
nils.reiter@uni-koeln.de

January 13, 2022
(Winter term 2020/21)

Section 1

Recap

Exercise 10

`https://github.com/idh-cologne-tools-ressourcen-infra/exercise-10`

Section 2

Apache UIMA, Part 2

Writing UIMA Components

Select Interface

- UIMA indexes all feature structures, such that we can efficiently access them

```
SelectFSs<Token> selector = jcas.select(Token.class);
selector.asList(); // a list of all feature structures
selector.count(); // number of feature structures
selector.get(index); // get nth token
```

Writing UIMA Components

Select Interface

- UIMA indexes all feature structures, such that we can efficiently access them

```
SelectFSs<Token> selector = jcas.select(Token.class);
selector.asList(); // a list of all feature structures
selector.count(); // number of feature structures
selector.get(index); // get nth token
selector.coveredBy(anotherAnnotation); // feature structures
                                       // covered by another
```

Writing UIMA Components

Select Interface

- UIMA indexes all feature structures, such that we can efficiently access them

```
SelectFSs<Token> selector = jcas.select(Token.class);
selector.asList(); // a list of all feature structures
selector.count(); // number of feature structures
selector.get(index); // get nth token
selector.coveredBy(anotherAnnotation); // feature structures
                                       // covered by another
selector.following(position); // first token after position
```

Writing UIMA Components

Select Interface

- UIMA indexes all feature structures, such that we can efficiently access them

```java
SelectFSs<Token> selector = jcas.select(Token.class);
selector.asList(); // a list of all feature structures
selector.count(); // number of feature structures
selector.get(index); // get nth token
selector.coveredBy(anotherAnnotation); // feature structures
                                        // covered by another
selector.following(position); // first token after position
selector.allMatch(predicate); // get tokens for which predicate
                              // is fulfilled
```

Writing UIMA Components

Select Interface: coveredBy()

```java
// iterate over sentences
for (Sentence sentence : jcas.select(Sentence.class)) {
  // iterate over tokens in a sentence
  for (Token token : jcas.select(Token.class).coveredBy(sentence)) {
    // do stuff with each token
  }
}
```

Writing UIMA Components

Select Interface: allMatch()

Predicate<Token> pred = t -> t.getId() > 15;

```
// iterate over sentences that start with "T"
for (Sentence sentence : jcas.select(Sentence.class)
    .allMatch(s -> s.getCoveredText().startsWith("T"))) {
  // iterate over tokens in sentence with id larger than 15
  for (Token token : jcas.select(Token.class).coveredBy(sentence)
      .allMatch(t -> t.getId() > 15)) {
    // do stuff with token
  }
}
```

pred

$\boxed{\text{Lambda Expressions in Java}}$

$t \rightarrow t.getId() > 15$

```java
// long form
Predicate<Token> predicate = new Predicate<Token>() {
  public boolean test(Token t) {
    return t.getId() > 15;
  }
};

```

Lambda Expressions in Java

```java
// long form
Predicate<Token> predicate = new Predicate<Token>() {
  public boolean test(Token t) {
    return t.getId() > 15;
  }
};

// short form ("syntactic sugar")
Predicate<Token> predicate = t -> t.getId() > 15;
```

Lambda Expressions in Java

```java
// long form
Predicate<Token> predicate = new Predicate<Token>() {
  public boolean test(Token t) {
    return t.getId() > 15;
  }
};

// short form ("syntactic sugar")
Predicate<Token> predicate = t -> t.getId() > 15;
```

- `java.util.function` package (since 1.8)
- Functional interface: Classes/interfaces with a single method
  - Predicate: `T` → `boolean` (`public boolean test(T)`)
  - BiPredicate: `T`, `S` → `boolean` (`public boolean test(T, S)`)
  - Function: `T` → `S`
  - ...

Section 3

Resources and Arguments

Resources and Arguments

- Many components need resources and/or parameters
- Define fields as usual
- Components may implement a method `public void initialize(UimaContext)`
  - ≃ constructor, but frameworks handles construction as needed
- Arguments need to be passed as atomic values (or read from file)

Resources and Arguments

- Many components need resources and/or parameters
- Define fields as usual
- Components may implement a method `public void initialize(UimaContext)`
  - ≃ constructor, but frameworks handles construction as needed
- Arguments need to be passed as atomic values (or read from file)

```java
public class MyComponent extends JCasAnnotator_ImplBase {

  @Override
  public void initialize(final UimaContext context)
    throws ResourceInitializationException {

    // call super class method
    super.initialize(context);

  }

  @Override
  public void process(JCas jcas) throws AnalysisEngineProcessException { }
}
```

Resources and Arguments

- Declare parameters as class fields
- Use `@ConfigurationParameter()` annotation to mark as configuration parameter
  - Javadoc:
    `https://javadoc.io/static/org.apache.uima/uimafit-core/3.1.0/org/apache/uima/fit/descriptor/ConfigurationParameter.html`
  - Arguments
    - `name = ...` Define a name for the parameter
    - `defaultValue = ...` Default value (as a string array)
    - `description = ...` Human-readable description
    - `mandatory = ...` Defines whether it can be omitted, `boolean` value

Resources and Arguments

- Declare parameters as class fields
- Use `@ConfigurationParameter()` annotation to mark as configuration parameter
  - Javadoc:
    `https://javadoc.io/static/org.apache.uima/uimafit-core/3.1.0/org/apache/uima/fit/descriptor/ConfigurationParameter.html`
  - Arguments
    - `name = ...` Define a name for the parameter
    - `defaultValue = ...` Default value (as a string array)
    - `description = ...` Human-readable description
    - `mandatory = ...` Defines whether it can be omitted, `boolean` value
- Do I need to define `initialize(UimaContext)`?
  - Not for purely filling given argument values into fields variables
  - Only if there is something to be done with the arguments
    - E.g., loading a file

Resources and Arguments: Best Practice

```java
public class MyComponent extends JCasAnnotator_ImplBase {
  // Define a static public variable that is used instead of the name
  public static final String PARAM_WINDOW_SIZE = "window size";

  @ConfigurationParameter(name = PARAM_WINDOW_SIZE, defaultValue="10", mandatory=false)
  int windowSize = 10;

  @Override
  public void process(JCas jcas) throws AnalysisEngineProcessException {
    for (Token token : jcas.select(Token.class)) {
      // get all tokens
      List<Token> nextTokens = jcas.select(Token.class)
        // ... that follow this token
        .following(token)
        // ... and only the first windowSize, as list
        .limit(windowSize).asList();
      // do something
    }
  }
}
```

Section 4

Writer Components

Writer Components

- We often need at least one component that produces output
  - HTML files with the annotations
  - CSV/ARFF files with extracted features
  - CSV files with frequency statistics
  - ...

Writer Components

- We often need at least one component that produces output
  - HTML files with the annotations
  - CSV/ARFF files with extracted features
  - CSV files with frequency statistics
  - ...
- Technically, writer components are analysis engines like any other component
- Output can be produced
  - Per document
  - For the entire collection

Writer Components

Writing to Files in Java

- Output writing in Java uses streams
- Stream: A stream of characters that flows from your code to somewhere else

Example

```
File file = new File("PATH/TO/MY/FILE");
FileOutputStream os = new FileOutputStream(file);
OutputStreamWriter writer = new OutputStreamWriter(os);
writer.write("This is the file content I want to write");
writer.flush();
writer.close();
```

Writer Components

Output per Document

- Done in `public void process(JCas)` method
- Usual Java mechanisms for opening files and writing to them
- Helpful: Inherit from `org.dkpro.core.api.io.JCasFileWriter_ImplBase`
  - New method `getOutputStream(JCas, String)` can be used
  - Creates a file with the same name as the input file (except extension)

Writer Components

Output per Document

- Done in `public void process(JCas)` method
- Usual Java mechanisms for opening files and writing to them
- Helpful: Inherit from `org.dkpro.core.api.io.JCasFileWriter_ImplBase`
  - New method `getOutputStream(JCas, String)` can be used
  - Creates a file with the same name as the input file (except extension)

Output for entire Collection

- Implement method `public void collectionProcessComplete()`
- Called after all documents have been processed

Section 5

Next Exercise

`https://github.com/idh-cologne-tools-ressourcen-infra/exercise-11`