GATE APIs

Track II, Module 6

Third GATE Training Course
August–September 2010

Outline

1. Using Java in JAPE
   - Basic JAPE
   - Java on the RHS
   - Common idioms

2. The GATE Ontology API
   - 5 minute guide to ontologies
   - Ontologies in GATE Embedded

Outline

JAPE

Pattern matching over annotations

- JAPE is a language for doing regular-expression-style pattern matching over *annotations* rather than text.
- Each JAPE rule consists of
  - Left hand side specifying the patterns to match
  - Right hand side specifying what to do when a match is found
- JAPE rules combine to create a phase
- Phases combine to create a grammar

An Example JAPE Rule

```
Rule: University1
(
  {Token.string == "University"}
  {Token.string == "of"}
  {Lookup.minorType == city}
):orgName
-->
:orgName.Organisation =
  {kind = "university", rule = "University1"}
```

Left hand side specifies annotations to match, optionally labelling some of them for use on the right hand side.

LHS Patterns

Elements

Left hand side of the rule specifies the pattern to match, in various ways

- Annotation type: `{Token}`
- Feature constraints:
  - `{Token.string == "University"}`
  - `{Token.length > 4}`
  - Also supports `<`, `<=`, `>=`, `!=` and regular expressions `=~`, `==~`, `!~`, `!=~`.
- Negative constraints:

```
{Token.length > 4, !Lookup.majorType == "stopword"}
```

  - This matches a Token of more than 4 characters that does not start at the same location as a "stopword" Lookup.
- Overlap constraints:

```
{Person within {Section.title == "authors"}}
```

LHS Patterns

Combinations

Pattern elements can be combined in various ways

- Sequencing: `{Token}{Token}`
- Alternatives: `{Token} | {Lookup}`
- Grouping with parentheses

Usual regular expression multiplicity operators

- zero-or-one: `({MyAnnot})?`
- zero-or-more: `({MyAnnot})*`
- one-or-more: `({MyAnnot})+`
- exactly *n*: `({MyAnnot})[n]`
- between *n* and *m* (inclusive): `({MyAnnot})[n,m]`

LHS Patterns

Labelling

Groups can be labelled. This has no effect on the matching process, but makes matched annotations available to the RHS

```
(
  {Token.string == "University"}
  {Token.string == "of"}
  ({Lookup.minorType == city}):uniTown
):orgName
```

RHS Actions

On the RHS, you can use the labels from the LHS to create new annotations:

```
-->
:uniTown.UniversityTown = {},
:orgName.Organisation =
  {kind = "university", rule = "University1"}
```

The `:label.AnnotationType = {features}` syntax creates a new annotation of the given type whose span covers all the annotations bound to the label.

- so the `Organisation` annotation will span from the start of the "University" Token to the end of the Lookup.

Macros

- You may find yourself re-using the same patterns in several places in a grammar.
  - e.g. `({Token.string ==~ "[A-Z]"}({Token.string == "."})?)+` to match initials.
- JAPE allows you to define *macros* - labelled patterns that can be re-used.

```
Macro: INITIALS
({Token.string ==~ "[A-Z]"}({Token.string == "."})?)+

Rule: InitialsAndSurname
( (INITIALS)?
  {Token.orth == "upperInitial"} ):per
-->
:per.Person = {rule = "InitialsAndSurname"}
```

Templates

- Templates are to values as macros are to pattern fragments.
- Declare a template once, reference it many times.
- Template value can be a quoted string, number or boolean (**true** or **false**).
- Template reference can go anywhere a quoted string could go.

```
Template: threshold = 0.6
Template: source = "Interesting location finder"

Rule: IsInteresting
({Location.score > [threshold]}):loc
-->
:loc.Entity = { kind = "Location", source = [source]}
```

Templates (cont)

- String templates can have *parameters*, parameter values supplied in the call.
- Useful if you have many similar strings in your grammar.

```
Template:
  wp = "http://${lang}.wikipedia.org/wiki/${page}"

Rule: EnglishWPCat
({a.href =~ [wp lang="en", page="Category:"]}):wp
-->
:wp.WPCategory = { lang = "en" }
```

- In a multi-phase grammar, templates and macros declared in one phase can be used in later phases.

Outline

Beyond Simple Actions

It's often useful to do more complex operations on the RHS than simply adding annotations, e.g.

- Set a new feature on one of the matched annotations
- Delete annotations from the input
- More complex feature value mappings, e.g. concatenate several LHS features to make one RHS one.
- Collect statistics, e.g. count the number of matched annotations and store the count as a document feature.
- Populate an ontology (later).

JAPE has no special syntax for these operations, but allows blocks of arbitrary Java code on the RHS.

Java on the RHS

```
Rule: HelloWorld
(
  {Token.string == "Hello"}
  {Token.string == "World"}
):hello
-->
{
  System.out.println("Hello world");
}
```

The RHS of a JAPE rule can have any number of `:bind.Type = {}` assignment expressions and blocks of Java code, separated by commas.

How JAPE Rules are Compiled

For each JAPE rule, GATE creates a Java class

```java
package japeactionclasses;
// various imports, see below

public class /* generated class name */
    implements RhsAction {
  public void doit(
      Document doc,
      Map<String, AnnotationSet> bindings,
      AnnotationSet annotations, // deprecated
      AnnotationSet inputAS,
      AnnotationSet outputAS,
      Ontology ontology) throws JapeException {
    // ...
  }
}
```

JAPE Action Classes

- Each block or assignment on the RHS becomes a block of Java code.
- These blocks are concatenated together to make the body of the `doit` method.
  - Local variables are local to each block, not shared.
- At runtime, whenever the rule matches, `doit` is called.

Java Block Parameters

The parameters available to Java RHS blocks are:

- **doc** The document currently being processed.
- **inputAS** The `AnnotationSet` specified by the `inputASName` runtime parameter to the JAPE transducer PR. Read or delete annotations from here.
- **outputAS** The `AnnotationSet` specified by the `outputASName` runtime parameter to the JAPE transducer PR. Create new annotations in here.
- **ontology** The ontology (if any) provided as a runtime parameter to the JAPE transducer PR.
- **bindings** The bindings map. . .

Bindings

- `bindings` is a `Map` from string to `AnnotationSet`
- Keys are labels from the LHS.
- Values are the annotations matched by the label.

```
(
  {Token.string == "University"}
  {Token.string == "of"}
  ({Lookup.minorType == city}):uniTown
):orgName
```

- `bindings.get("uniTown")` contains one annotation (the `Lookup`)
- `bindings.get("orgName")` contains three annotations (two `Token`s plus the `Lookup`)

Hands-on exercises

- The easiest way to experiment with JAPE is to use GATE Developer.
- The `hands-on` directory contains a number of sample JAPE files for you to modify, which will be described for each individual exercise.
- There is an `.xgapp` file for each exercise to load the right PRs and documents.
    - Good idea to *disable* session saving using Options → Configuration → Advanced (or GATE 6.0-beta1 → Preferences → Advanced on Mac OS X).

Exercise 1: A simple JAPE RHS

- Start GATE Developer.
- Load `hands-on/jape/exercise1.xgapp`
- This is the default ANNIE application with an additional JAPE transducer "exercise 1" at the end.
- This transducer loads the file `hands-on/jape/resources/simple.jape`, which contains a single simple JAPE rule.
- Modify the Java RHS block to print out the type and features of each annotation the rule matches. You need to right click the "Exercise 1 Transducer" and reinitialize after saving the `.jape` file.
- Test it by running the "Exercise 1" application.

Exercise 1: Solution

A possible solution:

```
Rule: ListEntities
({Person}|{Organization}|{Location}):ent
-->
{
  AnnotationSet ents = bindings.get("ent");
  for(Annotation e : ents) {
    System.out.println("Found " + e.getType()
        + " annotation");
    System.out.println("  features: "
        + e.getFeatures());
  }
}
```

Imports

- By default, every action class imports java.io.*, java.util.*, gate.*, gate.jape.*, gate.creole.ontology.*, gate.annotation.*, and gate.util.*.
- So classes from these packages can be used unqualified in RHS blocks.
- You can add additional imports by putting an import block at the top of the JAPE file, before the `Phase:` line:

```
Imports: {
  import my.pkg.*;
  import static gate.Utils.*;
}
```

You can import any class available in the GATE core or in any loaded plugin.

Named Java Blocks

```
-->
:uniTown{
  uniTownAnnots.iterator().next().getFeatures()
    .put("hasUniversity", Boolean.TRUE);
}
```

- You can label a Java block with a label from the LHS
- The block will only be called if there is at least one annotation bound to the label
- Within the Java block there is a variable `labelAnnots` referring to the `AnnotationSet` bound to the label
  - i.e. `AnnotationSet xyAnnots = bindings.get("xy")`

Exceptions

- Any `JapeException` or `RuntimeException` thrown by a Java RHS block will cause the JAPE Transducer PR to fail with an `ExecutionException`
- For non-fatal errors in a RHS block you can throw a `gate.jape.NonFatalJapeException`
- This will print debugging information (phase name, rule name, file and line number) but will not abort the transducer execution.
  - However it will interrupt this rule, i.e. if there is more than one block or assignment on the RHS, the ones after the **`throw`** will not run.

Returning from RHS blocks

- You can **return** from a Java RHS block, which prevents any later blocks or assignments for that rule from running, e.g.

```
-->
:uniTown{
  String townString = doc.getContent().getContent(
        uniTownAnnots.firstNode().getOffset(),
        uniTownAnnots.lastNode().getOffset())
      .toString();
  // don't add an annotation if this town has been seen before. If we
  // return, the UniversityTown annotation will not be created.
  if(!((Set)doc.getFeatures().get("knownTowns"))
      .add(townString)) return;
},
:uniTown.UniversityTown = {}
```

Outline

Common Idioms for Java RHS

Setting a new feature on one of the matched annotations

```
Rule: LcString
({Token}):tok
-->
:tok {
  for(Annotation a : tokAnnots) {
    // get the FeatureMap for the annotation
    FeatureMap fm = a.getFeatures();
    // get the "string" feature
    String str = (String)fm.get("string");
    // convert it to lower case and store
    fm.put("lcString", str.toLowerCase());
  }
}
```

Exercise 2: Modifying Existing Annotations

- Load `hands-on/jape/exercise2.xgapp`
- As before, this is ANNIE plus an extra transducer, this time loading `hands-on/jape/resources/general-pos.jape`.
- Modify the Java RHS block to add a `generalCategory` feature to the matched `Token` annotation holding the first two characters of the POS tag (the `category` feature).
- Remember to reinitialize the "Exercise 2 Transducer" after editing the JAPE file.
- Test it by running the "Exercise 2" application.

Exercise 2: Solution

A possible solution:

```
Rule: GeneralizePOSTag
({Token}):tok
-->
:tok {
  for(Annotation t : tokAnnots) {
    String pos = (String)t.getFeatures()
                                  .get("category");
    if(pos != null) {
      int gpLen = pos.length();
      if(gpLen > 2) gpLen = 2;
      t.getFeatures().put("generalCategory",
          pos.substring(0, gpLen));
    }
  }
}
```

Common Idioms for Java RHS

Removing matched annotations from the input

```
Rule: Location
({Lookup.majorType = "location"}):loc
-->
:loc.Location = { kind = :loc.Location.minorType,
      rule = "Location"},
:loc {
  inputAS.removeAll(locAnnots);
}
```

This can be useful to stop later phases matching the same annotations again.

Common Idioms for Java RHS

Accessing the string covered by a match

```
Rule: Location
({Lookup.majorType = "location"}):loc
-->
:loc {
  try {
    String str = doc.getContent().getContent(
        locAnnots.firstNode().getOffset(),
        locAnnots.lastNode().getOffset())
      .toString();
  }
  catch(InvalidOffsetException e) {
    // can't happen, but won't compile without the catch
  }
}
```

Utility methods

- `gate.Utils` provides static utility methods to make common tasks easier
  - `http://gate.ac.uk/gate/doc/javadoc/gate/Utils.html`
- Add an **import static** `gate.Utils.*;` to your `Imports:` block to use them.
- Accessing the string becomes `stringFor(doc, locAnnots)`
- This is also useful for division of labour
  - Java programmer writes utility class
  - JAPE expert writes rules, importing utility methods

Example: start and end

To get the start and end offsets of an `Annotation`, `AnnotationSet` or `Document`.

```
Rule: NPTokens
({NounPhrase}):np
-->
:np {
  List<String> posTags = new ArrayList<String>();
  for(Annotation tok : inputAS.get("Token")
      .getContained(start(npAnnots), end(npAnnots))) {
    posTags.add(
        (String)tok.getFeatures().get("category"));
  }
  FeatureMap fm =
    npAnnots.iterator().next().getFeatures();
  fm.put("posTags", posTags);
  fm.put("numTokens", (long)posTags.size());
}
```

Exercise 3: Working with Contained Annotations

- Load `hands-on/jape/exercise3.xgapp`
- As before, this is ANNIE plus an extra transducer, this time loading `hands-on/jape/resources/exercise3-main.jape`.
- This is a multiphase grammar containing the `general-pos.jape` from exercise 2 plus `num-nouns.jape`.
- Modify the Java RHS block in `num-nouns.jape` to count the number of nouns in the matched `Sentence` and add this count as a feature on the sentence annotation.
- Remember to reinitialize the "Exercise 3 Transducer" after editing the JAPE file.
- Test it by running the "Exercise 3" application.

Exercise 3: Solution

A possible solution:

```
Imports: { import static gate.Utils.*; }
Phase: NumNouns
Input: Sentence
Options: control = appelt

Rule: CountNouns
({Sentence}):sent
-->
```

Exercise 3: Solution (continued)

```
:sent {
  AnnotationSet tokens = inputAS.get("Token")
    .getContained(start(sentAnnots), end(sentAnnots));
  long numNouns = 0;
  for(Annotation t : tokens) {
    if("NN".equals(t.getFeatures()
                    .get("generalCategory"))) {
      numNouns++;
    }
  }
  sentAnnots.iterator().next().getFeatures()
    .put("numNouns", numNouns);
}
```

Passing state between rules

To pass state between rules, use document features:

```
Rule: Section
({SectionHeading}):sect
-->
:sect {
  doc.getFeatures().put("currentSection",
      stringFor(doc, sectAnnots));
}

Rule: Entity
({Entity}):ent
-->
:ent {
  entAnnots.iterator().next().getFeatures()
    .put("inSection",
         doc.getFeatures().get("currentSection"));
}
```

Passing state between rules

- Remember from yesterday - a `FeatureMap` can hold any Java object.
- So can pass complex structures between rules, not limited to simple strings.

Outline

Ontologies

A 5 minute introduction

- A set of concepts and relationships between them.
- GATE uses the *OWL* formalism for ontologies
- Classes, subclasses, instances, relationships
- Multiple inheritance
  - a class can have many superclasses
  - an instance can belong to many classes

Why Ontologies?

- Semantic annotation: rather than just annotating the word "Sheffield" as a location, link it to an ontology instance
  - Sheffield, UK rather than Sheffield, Massachusetts or Sheffield, Tasmania, etc.
- Reasoning
  - Ontology tells us that this particular Sheffield is part of the country called the United Kingdom, which is part of the continent Europe.
  - So we can infer that this document mentions a city in Europe.
- Relation extraction: match patterns in text and use them to add new information to the ontology.

Ontologies

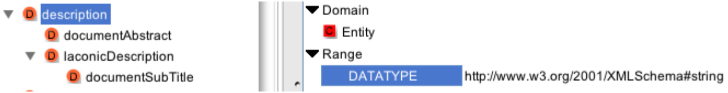
Properties

- *Properties* represent relationships between, and data about, instances.
- Properties can have hierarchy.

- *Object* properties relate one instance to another (DCS *partOf* University of Sheffield) — domain and range specify which classes the instances must belong to
- Can be symmetric, transitive

Ontologies

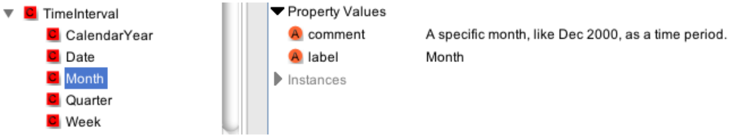
Datatype Properties



- *Datatype* properties attach simple data (*literals*) to instances.
- Available data types are taken from XML Schema.

Ontologies

Annotation Properties



- *Annotation* properties used to annotate classes, instances and other properties (collectively known as *resources*, confusingly).
- Similar to datatype properties, but those can only be attached to instances, not classes.
- e.g. RDFS defines properties like *comment* and *label* (a human-readable name for an ontology resource, as opposed to formal name of the resource which is a URI).

Outline

Ontologies in GATE Embedded

- GATE represents ontologies using abstract data model defined by interfaces in `gate.creole.ontology` package in `gate.jar`
- `Ontology` interface represents an ontology, `OClass`, `OInstance`, `OURI` etc. represent ontology components.
- Implementation provided by `Ontology` plugin, based on OWLIM version 3.
  - Alternative OWLIM 2-based implementation in `Ontology_OWLIM2` plugin for backwards compatibility only
  - Not possible to load both plugins at the same time.
- You need to load the plugin in order to create an `Ontology` object, but code should only interact with the interfaces.
  - `http://gate.ac.uk/gate/doc/javadoc/?gate/creole/ontology/package-summary.html`

Creating an empty ontology

```
Gate.init();
// load the Ontology plugin
Gate.getCreoleRegister().registerDirectories(
    new File(Gate.getPluginsHome(), "Ontology")
      .toURI().toURL());

Ontology emptyOnto = (Ontology)Factory.createResource(
    "gate.creole.ontology.impl.sesame.OWLIMOntology");
```

Loading an existing OWL file

More useful is to load an existing ontology. `OWLIMOntology` can load RDF-XML, N3, ntriples or turtle format.

```
// init GATE and load plugin as before...

URL owl = new File("ontology.owl").toURI().toURL();
FeatureMap params = Factory.newFeatureMap();
params.put("rdfXmlURL", owl);

Ontology theOntology = (Ontology)Factory.createResource(
    "gate.creole.ontology.impl.sesame.OWLIMOntology",
    params);
```

Under the Covers: Sesame

- The Ontology plugin implementation is built on OpenRDF Sesame version 2.
- `OWLIMOntology` LR creates a Sesame repository using a particular configuration of OWLIM as the underlying SAIL (Storage And Inference Layer)
- Other configurations or SAIL implementations can be used via alternative LRs: `CreateSesameOntology` (to create a new repository) and `ConnectSesameOntology` (to open an existing one).
    - though some parts of the GATE ontology API depend on the reasoning provided by OWLIM, so other SAILs may not behave exactly the same.

Persistent Repositories

- When loading an `OWLIMOntology` LR from RDF/ntriples, etc. OWLIM parses the source file and builds internal representation
- Can set `persistent` parameter to `true` and specify a `dataDirectoryURL` to store this internal representation on disk as a Sesame repository.
- `ConnectSesameOntology` PR can use the existing repository — much faster to init, particularly for large ontologies (e.g. 12k instances, 10 seconds to load from RDF, $<$ 0.2s to open repository).

Exploring the ontology

```
// get all the 'top' classes
Set<OClass> tops = ontology.getOClasses(true);

// list them along with their labels
for(OClass c : tops) {
  System.out.println(c.getONodeID() +
       " (" + c.getLabels() + ")");
}

// find a class by URI
OURI uri = ontology.createOURIForName("Person");
OClass personClass = ontology.getOClass(uri);
```

Exploring the ontology

```
// get direct instances of a class
Set<OInstance> people = ontology.getOInstances(
    personClass, OConstants.Closure.DIRECT_CLOSURE);

// get instances of a class or any of its subclasses
Set<OInstance> allPeople = ontology.getOInstances(
    personClass, OConstants.Closure.TRANSITIVE_CLOSURE);
```

Exploring the ontology

```
// get a datatype property
OURI namePropURI = ontology.createOURI(
    "http://example.org/stuff/1.0/hasName");
DatatypeProperty nameProp = ontology
    .getDatatypeProperty(namePropURI);

// find property values for an instance
for(OInstance person : allPeople) {
  List<Literal> names =
      ontology.getDatatypePropertyValues(nameProp);
  for(Literal name : names) {
    System.out.println("Person " + person.getONodeID()
        + " hasName " + name.toTurtle());
  }
}
```

Exploring the ontology

```
// University of Sheffield instance
OURI uosURI = ontology.createOURIForName(
    "UniversityOfSheffield");
OInstance uosInstance = ontology.getOInstance(uosURI);

// worksFor property
OURI worksForURI = ontology.createOURIForName(
    "worksFor");
ObjectProperty worksFor = ontology.getObjectProperty(
    worksForURI);

// find all the people who work for the University of Sheffield
List<OResource> uniEmployees =
    ontology.getOResourcesWith(worksFor, uosInstance);
```

A note about URIs

- Ontology resources are identified by URIs.
- URI is treated as a *namespace* (everything up to and including the last #, / or :, in that order) and a *resource name* (the rest)
- Ontology LR provides factory methods to create `OURI` objects:
  - `createOURI` takes a complete URI string
  - `createOURIForName` takes the resource name and prepends the ontology LR's *default namespace*
  - `generateOURI` takes a resource name, prepends the default NS and adds a unique suffix.
- Only ASCII letters, numbers and certain symbols are permitted in URIs, other characters (including spaces) must be escaped.
  - `OUtils` defines common escaping methods.

Extending the ontology

```
OURI personURI = ontology.createOURIForName("Person");
OClass personClass = ontology.getOClass(personURI);

// create a new class as a subclass of an existing class
OURI empURI = ontology.createOURIForName("Employee");
OClass empClass = ontology.addOClass(empURI);
personClass.addSubClass(empClass);

// create an instance
OURI fredURI = ontology.createOURIForName("FredSmith");
OInstance fred = ontology.addOInstance(fredURI,
                                       empClass);

// Fred works for the University of Sheffield
fred.addObjectPropertyValue(worksFor, uosInstance);
```

Exporting the ontology

```
OutputStream out = ....
ontology.writeOntologyData(out,
    OConstants.OntologyFormat.RDFXML, false);
```

- **`false`** means don't include `OResource`s that came from an import (**`true`** would embed the imported data in the exported ontology).
- Other formats are `TURTLE`, `N3` and `NTRIPLES`.

Ontology API in JAPE

- Recall that JAPE RHS blocks have access to an `ontology` parameter.
- Can use JAPE rules for ontology *population* or *enrichment*
- Create new instances or property values in an ontology based on patterns found in the text.

Exercise 1: Basic Ontology API

- Start GATE Developer.
- Load `hands-on/ontology/exercise1.xgapp`
- This xgapp loads two controllers. “Exercise 1 application” is a “trick” application containing a JAPE grammar `exercise1.jape` with a single rule that is guaranteed to fire exactly once when the application is run.
- The application loads `hands-on/ontology/demo.owl` and configures the JAPE transducer with that ontology.
- We treat the RHS of the rule as a “scratch pad” to test Java code that uses the ontology API.
- Also loads “Reset ontology” application you can use to reset the ontology to its original state.

Exercise 1: Basic Ontology API

- The initial JAPE file contains comments giving some suggested tasks.
- See how many of these ideas you can implement.
- Each time you modify the JAPE file you will need to re-init the "Exercise 1 transducer" then run the "Exercise 1 application".
- Open the ontology viewer to see the result of your changes.
- You will need to close and re-open the viewer each time.
- Use the reset application as necessary.

Remember: ontology API JavaDocs at
`http://gate.ac.uk/gate/doc/javadoc/?gate/creole/ontology/package-summary.html`

Exercise 1: Solutions

Possible solutions (exception handling omitted):

```
// Create an instance of the City class representing Sheffield
OURI cityURI = ontology.createOURIForName("City");
OClass cityClass = ontology.getOClass(cityURI);
OURI sheffieldURI = ontology.generateOURI("Sheffield");
OInstance sheffield = ontology.addOInstance(sheffieldURI,
                                            cityClass);

// Create a new class named "University" as a subclass of Organization
OURI orgURI = ontology.createOURIForName("Organization");
OURI uniURI = ontology.createOURIForName("University");
OClass orgClass = ontology.getOClass(orgURI);
OClass uniClass = ontology.addOClass(uniURI);
orgClass.addSubClass(uniClass);
```

Exercise 1: Solutions (continued)

```
// Create an instance of the University class representing the University of Sheffield
OURI unishefURI = ontology.generateOURI(
    OUtils.toResourceName("University of Sheffield"));
OInstance unishef = ontology.addOInstance(unishefURI,
                                          uniClass);

// Create an object property basedAt with domain Organization and range Location
OURI locationURI = ontology.createOURIForName("Location");
OClass locationClass = ontology.getOClass(locationURI);
OURI basedAtURI = ontology.createOURIForName("basedAt");
ObjectProperty basedAt = ontology.addObjectProperty(
    basedAtURI, Collections.singleton(orgClass),
    Collections.singleton(locationClass));

// Specify that the University of Sheffield is basedAt Sheffield
unishef.addObjectPropertyValue(basedAt, sheffield);
```

Ontology-aware JAPE

- When supplied with an ontology parameter, JAPE can do ontology-aware matching.
- In this mode the feature named “class” on an annotation is special: it is assumed to be an ontology class URI, and will match any subclass.
- If the class feature is not a complete URI, it has the ontology’s default namespace prepended.
  - e.g. `{Lookup.class == "Location"}` with our demo ontology would match Lookup annotations with any subclass of `http://www.owl-ontologies.com/unnamed.owl#Location`, in the class feature, including “City”, “Country”, etc.
- When an ontology parameter is *not* specified, class is treated the same as any other feature (not the case prior to GATE 5.2).

Ontology Population

- Ontology population is the process of adding instances to an ontology based on information found in text.
- We will explore a very simple example, real-world ontology population tasks are complex and domain-specific.

Ontology population example

- The demo ontology from exercise 1 contains a “Location” class with subclasses “City”, “Country”, “Province” and “Region”.
- These correspond to subsets of the ANNIE named entities.
- We want to populate our ontology with instances for each location in a document.
- Very simple assumption – if two Location annotations have the same text, they refer to the same location.
  - Typically you would need to disambiguate, e.g. with coreference information.

Exercise 2: Ontology population

- Start GATE Developer
- Load `hands-on/ontology/exercise2.xgapp`
- This xgapp again loads the demo ontology and defines the ontology reset controller.
- Second controller in this case is a normal ANNIE with two additional JAPE grammars.

ANNIE `locType` to Ontology Class

- ANNIE creates `Location` annotations with a `locType` feature, and `Organization` annotations with an `orgType` feature.
  - e.g. locType = region
- The first of the two additional grammars (“NEs to Mentions”) creates annotations of type Mention with a “class” feature derived from the `locType` or `orgType`.
- Location (or Organization) annotations without a locType (or orgType) are mapped to the top-level Location (Organization) class.

Populating the ontology

- Given these Mention annotations, we can now populate the ontology.
- We want to create one instance for each distinct entity.
- Use the RDFS “label” annotation property to associate the instance with its text.
- So for each Mention of a Location, we need to:
    - determine which ontology class it is a mention of
    - see if there is already an instance of this class with a matching label, and if not, create one, and
    - store the URI of the relevant ontology instance on the Mention annotation.

Exercise 2: Ontology population

Over to you!

- Fill in `hands-on/ontology/exercise2.jape` to implement this algorithm.
- As before, you need to re-init the Exercise 2 transducer each time you edit the JAPE file.
- Use the "Reset ontology" application to clean up the ontology between runs (though if you do it right it won't create extra instances if you run again without cleaning).

Exercise 2: Solution

A possible solution

```
// Create some useful objects - rdfs:label property and a Literal for the covered text.
AnnotationProperty rdfsLabel = ontology.getAnnotationProperty(
    ontology.createOURI(OConstants.RDFS.LABEL));
Literal text = new Literal(stringFor(doc, locAnnots));

for(Annotation mention : locAnnots) {
  // determine the right class
  OURI classUri = ontology.createOURI(
      (String)m.getFeatures().get("class"));
  OClass clazz = ontology.getOClass(classUri);

  // get all existing instances of that class
  Set<OInstance> instances = ontology.getOInstances(clazz,
      OConstants.DIRECT_CLOSURE);
```

Exercise 2: Solution (continued)

```
// see if any of them have the right label – if so, we assume they're the same
OInstance inst = null;
for(OInstance candidate : instances) {
  if(candidate.getAnnotationPropertyValues(
        rdfsLabel).contains(text)) {
    // found it!
    inst = candidate;
    break;
  }
}
```

Exercise 2: Solution (continued)

```
  if(inst == null) {
    // not found an existing instance, create one with a generated name
    String instName = OUtils.toResourceName(text.getValue());
    OURI instURI = ontology.generateOURI(instName + "_");
    inst = ontology.addOInstance(instURI, clazz);
    // and label it with the covered text
    inst.addAnnotationPropertyValue(rdfsLabel, text);
  }

  // finally, store the URI of the (new or existing) instance on the annotation
  mention.getFeatures().put("inst",
      inst.getONodeID().toString());
}
```

Conclusions and further reading

- This is a good example of a case where utility classes are useful.
- We have used this technique in other projects, e.g. `gate.ac.uk/sale/icsd09/sprat.pdf`
- Lots of tutorial materials on ontologies, OWL, etc. available online.
- For GATE, best references are the user guide and javadocs.