Today’s Plan

1. Jena inference support
2. Using the built-in reasoners
3. Using an external reasoner
4. Simple reasoner configuration
5. Introduction to OWL
Outline

1. Jena inference support
2. Using the built-in reasoners
3. Using an external reasoner
4. Simple reasoner configuration
5. Introduction to OWL
The Jena inference system

- Designed for plug-and-play compatibility with different reasoners.
The Jena inference system

- Designed for plug-and-play compatibility with different reasoners.
- Different reasoners implement different axioms and rules, e.g.
The Jena inference system

- Designed for plug-and-play compatibility with different reasoners.
- Different reasoners implement different axioms and rules, e.g.
  - Simple taxonomic reasoning,
The Jena inference system

- Designed for plug-and-play compatibility with different reasoners.
- Different reasoners implement different axioms and rules, e.g.
  - Simple taxonomic reasoning,
  - RDFS,
The Jena inference system

- Designed for plug-and-play compatibility with different reasoners.
- Different reasoners implement different axioms and rules, e.g.
  - Simple taxonomic reasoning,
  - RDFS,
  - OWL,
The Jena inference system

- Designed for plug-and-play compatibility with different reasoners.

- Different reasoners implement different axioms and rules, e.g.
  - Simple taxonomic reasoning,
  - RDFS,
  - OWL,
  - Rule languages (SWRL, Jena rules. Covered in a later lecture).
The Jena inference system

- Designed for plug-and-play compatibility with different reasoners.
- Different reasoners implement different axioms and rules, e.g.
  - Simple taxonomic reasoning,
  - RDFS,
  - OWL,
  - Rule languages (SWRL, Jena rules. Covered in a later lecture).
- Three different types of reasoners:
The Jena inference system

- Designed for plug-and-play compatibility with different reasoners.
- Different reasoners implement different axioms and rules, e.g.
  - Simple taxonomic reasoning,
  - RDFS,
  - OWL,
  - Rule languages (SWRL, Jena rules. Covered in a later lecture).
- Three different types of reasoners:
  - Built-in reasoners,
The Jena inference system

- Designed for plug-and-play compatibility with different reasoners.

- Different reasoners implement different axioms and rules, e.g.
  - Simple taxonomic reasoning,
  - RDFS,
  - OWL,
  - Rule languages (SWRL, Jena rules. Covered in a later lecture).

- Three different types of reasoners:
  - Built-in reasoners,
  - External reasoners (Pellet, Fact++, a. o.)
The Jena inference system

- Designed for plug-and-play compatibility with different reasoners.
- Different reasoners implement different axioms and rules, e.g.
  - Simple taxonomic reasoning,
  - RDFS,
  - OWL,
  - Rule languages (SWRL, Jena rules. Covered in a later lecture).
- Three different types of reasoners:
  - Built-in reasoners,
  - External reasoners (Pellet, Fact++, a. o.)
  - DIG reasoners,
The Jena inference system

- Designed for plug-and-play compatibility with different reasoners.
- Different reasoners implement different axioms and rules, e.g.
  - Simple taxonomic reasoning,
  - RDFS,
  - OWL,
  - Rule languages (SWRL, Jena rules. Covered in a later lecture).
- Three different types of reasoners:
  - Built-in reasoners,
  - External reasoners (Pellet, Fact++, a. o.)
  - DIG reasoners,
    - XML standard for access to description logic processing via HTTP.
The Jena inference system

- Designed for plug-and-play compatibility with different reasoners.

- Different reasoners implement different axioms and rules, e.g.
  - Simple taxonomic reasoning,
  - RDFS,
  - OWL,
  - Rule languages (SWRL, Jena rules. Covered in a later lecture).

- Three different types of reasoners:
  - Built-in reasoners,
  - External reasoners (Pellet, Fact++, a. o.)
  - DIG reasoners,
    - XML standard for access to description logic processing via HTTP.
    - (not covered here)
Reasoner factories and the reasoner registry

- There is a ReasonerFactory class for each type of reasoner.
Reasoner factories and the reasoner registry

- There is a ReasonerFactory class for each type of reasoner.
- It is used to create instances of the associated reasoner.
Reasoner factories and the reasoner registry

- There is a ReasonerFactory class for each type of reasoner.
- It is used to create instances of the associated reasoner.
- Built-in factories are stored in a global ReasonerRegistry class.

Three principal ways to obtain a stand-alone reasoner:

I. Import and use a known factory class, works for built-in and external reasoners alike
II. use a convenience method on the registry
III. retrieve a reasoner from the registry using the reasoners URI index suitable for built-in reasoners

The reasoner can then be applied to a model, to produce an InfModel, by applying the reasoner to a plain Model, using ModelFactory.createInfModel(reasoner, model)
Reasoner factories and the reasoner registry

- There is a `ReasonerFactory` class for each type of reasoner.
- It is used to create instances of the associated reasoner.
- Built-in factories are stored in a global `ReasonerRegistry` class.
- Three principal ways to obtain a stand-alone reasoner:
Reasoner factories and the reasoner registry

- There is a ReasonerFactory class for each type of reasoner.
- It is used to create instances of the associated reasoner.
- Built-in factories are stored in a global ReasonerRegistry class.
- Three principal ways to obtain a stand-alone reasoner:
  1. Import and use a known factory class,
Reasoner factories and the reasoner registry

- There is a `ReasonerFactory` class for each type of reasoner.
- It is used to create instances of the associated reasoner.
- Built-in factories are stored in a global `ReasonerRegistry` class.
- Three principal ways to obtain a stand-alone reasoner:
  - I. Import and use a known factory class,
    - works for built-in and external reasoners alike
Reasoner factories and the reasoner registry

- There is a `ReasonerFactory` class for each type of reasoner.
- It is used to create instances of the associated reasoner.
- Built-in factories are stored in a global `ReasonerRegistry` class.
- Three principal ways to obtain a stand-alone reasoner:
  1. Import and use a known factory class,
     - works for built-in and external reasoners alike
  2. use a convenience method on the registry
Reasoner factories and the reasoner registry

- There is a ReasonerFactory class for each type of reasoner.
- It is used to create instances of the associated reasoner.
- Built-in factories are stored in a global ReasonerRegistry class.
- Three principal ways to obtain a stand-alone reasoner:
  - I. Import and use a known factory class,
    - works for built-in and external reasoners alike
  - II. use a convenience method on the registry
  - III. retrieve a reasoner from the registry using the reasoners URI index

The reasoner can then be applied to a model, to produce an InfModel, by applying the reasoner to a plain Model, using ModelFactory.createInfModel(reasoner, model)
Reasoner factories and the reasoner registry

- There is a ReasonerFactory class for each type of reasoner.
- It is used to create instances of the associated reasoner.
- Built-in factories are stored in a global ReasonerRegistry class.
- Three principal ways to obtain a stand-alone reasoner:
  1. Import and use a known factory class,
     - works for built-in and external reasoners alike
  2. use a convenience method on the registry
  3. retrieve a reasoner from the registry using the reasoners URI index
     - suitable for built-in reasoners

The reasoner can then be applied to a model, to produce an InfModel, by applying the reasoner to a plain Model, using ModelFactory.createInfModel(reasoner, model)
There is a `ReasonerFactory` class for each type of reasoner. It is used to create instances of the associated reasoner.

Built-in factories are stored in a global `ReasonerRegistry` class.

Three principal ways to obtain a stand-alone reasoner:

1. Import and use a known factory class,
   - works for built-in and external reasoners alike
2. Use a convenience method on the registry
3. Retrieve a reasoner from the registry using the reasoners URI index
   - suitable for built-in reasoners

The reasoner can then be applied to a model,
Reasoner factories and the reasoner registry

- There is a `ReasonerFactory` class for each type of reasoner.
- It is used to create instances of the associated reasoner.
- Built-in factories are stored in a global `ReasonerRegistry` class.
- Three principal ways to obtain a stand-alone reasoner:
  1. Import and use a known factory class,
     - works for built-in and external reasoners alike
  2. use a `convenience method` on the registry
  3. retrieve a reasoner from the registry using the reasoners URI index
    - suitable for built-in reasoners
- The reasoner can then be applied to a model,
  - to produce an `InfModel`,

INF3580 :: Spring 2010  Lecture 7 :: 9th February 5 / 41
Reasoner factories and the reasoner registry

- There is a ReasonerFactory class for each type of reasoner.
- It is used to create instances of the associated reasoner.
- Built-in factories are stored in a global ReasonerRegistry class.
- Three principal ways to obtain a stand-alone reasoner:
  1. Import and use a known factory class,
     - works for built-in and external reasoners alike
  2. use a convenience method on the registry
  3. retrieve a reasoner from the registry using the reasoners URI index
     - suitable for built-in reasoners
- The reasoner can then be applied to a model,
  - to produce an InfModel,
  - by applying the reasoner to a plain Model,
Reasoner factories and the reasoner registry

- There is a `ReasonerFactory` class for each type of reasoner.
- It is used to create instances of the associated reasoner.
- Built-in factories are stored in a global `ReasonerRegistry` class.
- Three principal ways to obtain a stand-alone reasoner:
  - I. Import and use a known factory class,
    - works for built-in and external reasoners alike
  - II. use a `convenience method` on the registry
  - III. retrieve a reasoner from the registry using the reasoners URI index
    - suitable for built-in reasoners
- The reasoner can then be applied to a model,
  - to produce an `InfModel`,
  - by applying the reasoner to a plain `Model`,
  - using `ModelFactory.createInfModel(reasoner, model)`
One can also construct an `InfModel` in one go
One can also construct an `InfModel` in one go
- by using convenience methods on the `ModelFactory` class
One can also construct an InfModel in one go
- by using convenience methods on the ModelFactory class
  - e.g. ModelFactory.createRDFSModel(model).
One can also construct an InfModel in one go

- by using convenience methods on the ModelFactory class
  - e.g. ModelFactory.createRDFSModel(model).
- This is typically very simple,
One can also construct an InfModel in one go

- by using convenience methods on the ModelFactory class
  - e.g. ModelFactory.createRDFSModel(model).
- This is typically very simple,
- but makes it more difficult to configure the reasoner
One can also construct an InfModel in one go

- by using convenience methods on the ModelFactory class
  - e.g. ModelFactory.createRDFSModel(model).
- This is typically very simple,
- but makes it more difficult to configure the reasoner
One can also construct an InfModel in one go
- by using convenience methods on the ModelFactory class
  - e.g. ModelFactory.createRDFSModel(model).
- This is typically very simple,
- but makes it more difficult to configure the reasoner

ModelFactory also has convinience methods that return an OntModel
One can also construct an InfModel in one go

- by using convenience methods on the ModelFactory class
  
  e.g. ModelFactory#createRDFSModel(model).

- This is typically very simple,

- but makes it more difficult to configure the reasoner

ModelFactory also has convience methods that return an OntModel

- the OntModel class is a subclass of InfModel
One can also construct an InfModel in one go
- by using convenience methods on the ModelFactory class
  - e.g. ModelFactory.createRDFSModel(model).
- This is typically very simple,
- but makes it more difficult to configure the reasoner

ModelFactory also has convenience methods that return an OntModel
- the OntModel class is a subclass of InfModel
- has a richer API,
Contd.

One can also construct an `InfModel` in one go
- by using convenience methods on the `ModelFactory` class
  - e.g. `ModelFactory.createRDFSModel(model)`.
- This is typically very simple,
- but makes it more difficult to configure the reasoner.

`ModelFactory` also has convenience methods that return an `OntModel`
- the `OntModel` class is a subclass of `InfModel`
- has a richer API,
- and can be configured with an `OntModelSpec` parameter.
One can also construct an InfModel in one go

- by using convenience methods on the ModelFactory class
  - e.g. ModelFactory.createRDFSModel(model).

- This is typically very simple,
- but makes it more difficult to configure the reasoner

ModelFactory also has convenience methods that return an OntModel

- the OntModel class is a subclass of InfModel
- has a richer API,
- and can be configured with an OntModelSpec parameter
- by calling ModelFactory.createOntologyModel(param, model).
Outline

1. Jena inference support
2. Using the built-in reasoners
3. Using an external reasoner
4. Simple reasoner configuration
5. Introduction to OWL
Included in the Jena distribution are a number of predefined reasoners:
Built-in reasoners

Included in the Jena distribution are a number of predefined reasoners:

**Transitive reasoner:** Provides support for simple taxonomy traversal.
Included in the Jena distribution are a number of predefined reasoners:

**Transitive reasoner:** Provides support for simple taxonomy traversal.
- Implements only the *reflexivity* and *transitivity* of
Included in the Jena distribution are a number of predefined reasoners:

**Transitive reasoner:** Provides support for simple taxonomy traversal.

- Implements only the **reflexivity** and **transitivity** of
  - `rdfs:subPropertyOf`, and
Included in the Jena distribution are a number of predefined reasoners:

**Transitive reasoner:** Provides support for simple taxonomy traversal.
- Implements only the *reflexivity* and *transitivity* of
  - `rdfs:subPropertyOf`, and
  - `rdfs:subClassOf`.

---

INF3580 :: Spring 2010
Lecture 7 :: 9th February
Built-in reasoners

Included in the Jena distribution are a number of predefined reasoners:

**Transitive reasoner**: Provides support for simple taxonomy traversal.
- Implements only the *reflexivity* and *transitivity* of
  - `rdfs:subPropertyOf`, and
  - `rdfs:subClassOf`. 
Included in the Jena distribution are a number of predefined reasoners:

**Transitive reasoner:** Provides support for simple taxonomy traversal.
- Implements only the reflexivity and transitivity of
  - `rdfs:subPropertyOf`, and
  - `rdfs:subClassOf`.

**RDFS rule reasoner:** Supports most of the axioms and inference rules specific to RDFS.
Built-in reasoners

Included in the Jena distribution are a number of predefined reasoners:

**Transitive reasoner:** Provides support for simple taxonomy traversal.
- Implements only the reflexivity and transitivity of
  - rdfs:subPropertyOf, and
  - rdfs:subClassOf.

**RDFS rule reasoner:** Supports most of the axioms and inference rules specific to RDFS.

**OWL, OWL mini/micro reasoners:** Implementations of different subsets of OWL (Lite).
Built-in reasoners

Included in the Jena distribution are a number of predefined reasoners:

**Transitive reasoner:** Provides support for simple taxonomy traversal.
- Implements only the reflexivity and transitivity of
  - `rdfs:subPropertyOf`, and
  - `rdfs:subClassOf`.

**RDFS rule reasoner:** Supports most of the axioms and inference rules specific to RDFS.

**OWL, OWL mini/micro reasoners:** Implementations of different subsets of OWL (Lite).

**Generic rule reasoner:** A rule-based reasoner that supports user defined rules.
Using the built-in reasoners

Creating a simple RDFSModel

```java
Model sche = FileManager.get().LoadModel(aURI);
Model dat = FileManager.get().LoadModel(bURI);
InfModel inferredModel = ModelFactory.createRDFSModel(sche, dat);
```
Using convenience methods on ModelFactory

Creating a simple RDFSModel

Model sche = FileManager.get().LoadModel(aURI);
Model dat = FileManager.get().LoadModel(bURI);
InfModel inferredModel = ModelFactory.createRDFSModel(sche, dat);

- createRDFSModel() returns an InfModel.
Creating a simple RDFSModel

Model sche = FileManager.get().LoadModel(aURI);
Model dat = FileManager.get().LoadModel(bURI);
InfModel inferredModel = ModelFactory.createRDFSModel(sche, dat);

- `createRDFSModel()` returns an `InfModel`.
- An `InfModel` supports access to basic inference capability, such as;
Using the built-in reasoners

Using convenience methods on ModelFactory

Creating a simple RDFSModel

Model sche = FileManager.get().LoadModel(aURI);
Model dat = FileManager.get().LoadModel(bURI);
InfModel inferredModel = ModelFactory.createRDFSModel(sche, dat);

- `createRDFSModel()` returns an `InfModel`.
- An `InfModel` supports access to **basic inference capability**, such as:
  - `getDeductionsModel()` which returns the inferred triples,
Using the built-in reasoners

Using convenience methods on ModelFactory

Creating a simple RDFSModel

Model sche = FileManager.get().LoadModel(aURI);
Model dat = FileManager.get().LoadModel(bURI);
InfModel inferredModel = ModelFactory.createRDFSModel(sche, dat);

- `createRDFSModel()` returns an `InfModel`.
- An `InfModel` supports access to **basic inference capability**, such as:
  - `getDeductionsModel()` which returns the inferred triples,
  - `getRawModel()` which returns the base triples,
createRDFSModel() returns an InfModel.

An InfModel supports access to basic inference capability, such as:

- getDeductionsModel() which returns the inferred triples,
- getRawModel() which returns the base triples,
- getReasoner() which returns the RDFS reasoner,
Using the built-in reasoners

Creating a simple RDFSModel

Model sche = FileManager.get().LoadModel(aURI);
Model dat = FileManager.get().LoadModel(bURI);
InfModel inferredModel = ModelFactory.createRDFSModel(sche, dat);

- createRDFSModel() returns an InfModel.
- An InfModel supports access to basic inference capability, such as:
  - getDeductionsModel() which returns the inferred triples,
  - getRawModel() which returns the base triples,
  - getReasoner() which returns the RDFS reasoner,
  - getDerivation(stmt) which returns the derivation of stmt.
Building an InfModel in two steps

The convenience methods on the previous slide builds an InfModel in one go.
Building an InfModel in two steps

The convenience methods on the previous slide builds an InfModel in one go.

- We may also build it in the following manner:
Building an InfModel in two steps

The convenience methods on the previous slide builds an InfModel in one go.

- We may also build it in the following manner:
  1. Obtain a reasoner first,
Building an InfModel in two steps

The convenience methods on the previous slide builds an InfModel in one go.

- We may also build it in the following manner:
  1. Obtain a reasoner first,
  2. Construct a Model object (that is, an RDF graph)
Building an InfModel in two steps

The convenience methods on the previous slide builds an InfModel in one go.

- We may also build it in the following manner:
  1. Obtain a reasoner first,
  2. Construct a Model object (that is, an RDF graph)
  3. pass the reasoner and the model (possibly more than one) to ModelFactory.createInfModel
Building an InfModel in two steps

The convenience methods on the previous slide builds an InfModel in one go.

- We may also build it in the following manner:
  1. Obtain a reasoner first,
  2. Construct a Model object (that is, an RDF graph)
  3. pass the reasoner and the model (possibly more than one) to ModelFactory.createInfModel
Building an InfModel in two steps

The convenience methods on the previous slide builds an InfModel in one go.

- We may also build it in the following manner:
  1. Obtain a reasoner first,
  2. Construct a Model object (that is, an RDF graph)
  3. pass the reasoner and the model (possibly more than one) to
     ModelFactory.createInfModel

Reasoners are returned by static convenience methods on the registry:
Building an InfModel in two steps

The convenience methods on the previous slide builds an InfModel in one go.

- We may also build it in the following manner:
  1. Obtain a reasoner first,
  2. Construct a Model object (that is, an RDF graph)
  3. Pass the reasoner and the model (possibly more than one) to `ModelFactory.createInfModel`

Reasoners are returned by static convenience methods on the registry:

- `ReasonerRegistry.getOWLMicroReasoner()`,
- `ReasonerRegistry.getOWLMiniReasoner()`,
- `ReasonerRegistry.getOWLReasoner()`,
- `ReasonerRegistry.getRDFSReasoner()`,
- `ReasonerRegistry.getRDFSSimpleReasoner()`,
- `ReasonerRegistry.getTransitiveReasoner()`
Building an InfModel in two steps

The convenience methods on the previous slide builds an InfModel in one go.

- We may also build it in the following manner:
  1. Obtain a reasoner first,
  2. Construct a Model object (that is, an RDF graph)
  3. pass the reasoner and the model (possibly more than one) to ModelFactory.createInfModel

Reasoners are returned by static convenience methods on the registry:

- ReasonerRegistry.getOWLMicroReasoner(),
- ReasonerRegistry.getOWLMiniReasoner(),
- ReasonerRegistry.getOWLReasoner(),
- ReasonerRegistry.getRDFSReasoner(),
- ReasonerRegistry.getRDFSSimpleReasoner(),
- ReasonerRegistry.getTransitiveReasoner()
Using the built-in reasoners

Building an InfModel in two steps

The convenience methods on the previous slide builds an InfModel in one go.

- We may also build it in the following manner:
  1. Obtain a reasoner first,
  2. Construct a Model object (that is, an RDF graph)
  3. pass the reasoner and the model (possibly more than one) to ModelFactory.createInfModel

Reasoners are returned by static convenience methods on the registry:

- ReasonerRegistry.getOWLMicroReasoner()
- ReasonerRegistry.getOWLMiniReasoner()
- ReasonerRegistry.getOWLReasoner()
- ReasonerRegistry.getRDFSReasoner()
- ReasonerRegistry.getRDFSSimpleReasoner()
- ReasonerRegistry.getTransitiveReasoner()
Using the built-in reasoners

Building an InfModel in two steps

The convenience methods on the previous slide builds an InfModel in one go.

- We may also build it in the following manner:
  1. Obtain a reasoner first,
  2. Construct a Model object (that is, an RDF graph)
  3. Pass the reasoner and the model (possibly more than one) to ModelFactory.createInfModel

Reasoners are returned by static convenience methods on the registry:

- ReasonerRegistry.getOWLMicroReasoner(),
- ReasonerRegistry.getOWLMiniReasoner(),
- ReasonerRegistry.getOWLReasoner(),
- ReasonerRegistry.getRDFSReasoner(),
- ReasonerRegistry.getRDFSSimpleReasoner()
Building an InfModel in two steps

The convenience methods on the previous slide builds an InfModel in one go.

- We may also build it in the following manner:
  1. Obtain a reasoner first,
  2. Construct a Model object (that is, an RDF graph)
  3. Pass the reasoner and the model (possibly more than one) to ModelFactory.createInfModel

Reasoners are returned by static convenience methods on the registry:

- ReasonerRegistry.getOWLMicroReasoner()
- ReasonerRegistry.getOWLMiniReasoner()
- ReasonerRegistry.getOWLReasoner()
- ReasonerRegistry.getRDFSReasoner()
- ReasonerRegistry.getRDFSSimpleReasoner()
Building an InfModel in two steps

The convenience methods on the previous slide builds an InfModel in one go.

- We may also build it in the following manner:
  1. Obtain a reasoner first,
  2. Construct a Model object (that is, an RDF graph)
  3. pass the reasoner and the model (possibly more than one) to
     ModelFactory.createInfModel

Reasoners are returned by static convenience methods on the registry:

- ReasonerRegistry.getOWLMicroReasoner(),
- ReasonerRegistry.getOWLMiniReasoner(),
- ReasonerRegistry.getOWLReasoner(),
- ReasonerRegistry.getRDFSReasoner(),
- ReasonerRegistry.getRDFSSimpleReasoner(),
- ReasonerRegistry.getTransitiveReasoner()
Using the built-in reasoners contd.

Model sche = FileManager.get().LoadModel(aURI);
Model dat = FileManager.get().LoadModel(bURI);
Reasoner reas = ReasonerRegistry.getOWLReasoner();
InfModel inf = ModelFactory.createInfModel(reas, sche, dat);

This abstract two-step procedure will be the default, since we retain a reference to the reasoner, that can be used for configuration. And since it is suitable for built-in and external reasoners alike.
using `ModelFactory.createInfModel`

```java
Model sche = FileManager.get().LoadModel(aURI);
Model dat = FileManager.get().LoadModel(bURI);

Reasoner reas = ReasonerRegistry.getOWLReasoner();
InfModel inf = ModelFactory.createInfModel(reas, sche, dat);
```
using ModelFactory.createInfModel

Model sche = FileManager.get().LoadModel(aURI);
Model dat = FileManager.get().LoadModel(bURI);

Reasoner reas = ReasonerRegistry.getOWLReasoner();
InfModel inf = ModelFactory.createInfModel(reas, sche, dat);

This abstract two-step procedure will be the default, since;
using ModelFactory.createInfModel

Model sche = FileManager.get().LoadModel(aURI);
Model dat = FileManager.get().LoadModel(bURI);

Reasoner reas = ReasonerRegistry.getOWLReasoner();
InfModel inf = ModelFactory.createInfModel(reas, sche, dat);

This abstract two-step procedure will be the default, since;

- we retain a reference to the reasoner,
contd.

```java
using ModelFactory.createInfModel

Model sche = FileManager.get().LoadModel(aURI);
Model dat = FileManager.get().LoadModel(bURI);

Reasoner reas = ReasonerRegistry.getOWLReasoner();
InfModel inf = ModelFactory.createInfModel(reas, sche, dat);
```

This abstract two-step procedure will be the default, since:

- we retain a reference to the reasoner,
- that can be used for configuration.
using ModelFactory.createInfModel

Model sche = FileManager.get().LoadModel(aURI);
Model dat = FileManager.get().LoadModel(bURI);

Reasoner reas = ReasonerRegistry.getOWLReasoner();
InfModel inf = ModelFactory.createInfModel(reas, sche, dat);

This abstract two-step procedure will be the default, since:

- we retain a reference to the reasoner,
- that can be used for configuration.
- And since it is suitable for built-in and external reasoners alike
There are other built-in reasoners than those that are accessible through
There are other built-in reasoners than those that are accessible through

- the convenience methods on ModelFactory,

All reasoners can be looked up in the registry. The `ReasonerRegistry` stores factory instances indexed by URIs. Reasoners can be retrieved using these indexes, by `registry.create(reasonerURI, param)` where `param` is a configuration parameter, of type `Resource`, but it doesn't do much, and is usually replaced with `null`. 
There are other built-in reasoners than those that are accessible through
  - the convenience methods on ModelFactory,
  - and on ReasonerRegistry,
There are other built-in reasoners than those that are accessible through

- the convenience methods on ModelFactory,
- and on ReasonerRegistry,
- for instance the GenericRuleReasoner.
There are other built-in reasoners than those that are accessible through the convenience methods on ModelFactory, and on ReasonerRegistry, for instance the GenericRuleReasoner.

All reasoners can be looked up in the registry.
Using the built-in reasoners

Accessing all built-in reasoners

- There are other built-in reasoners than those that are accessible through
  - the convenience methods on ModelFactory,
  - and on ReasonerRegistry,
  - for instance the GenericRuleReasoner.
- All reasoners can be looked up in the registry.
- The ReasonerRegistry stores factory instances indexed by URIs.
There are other built-in reasoners than those that are accessible through
the convenience methods on ModelFactory, and on ReasonerRegistry,
for instance the GenericRuleReasoner.

- All reasoners can be looked up in the registry.
- The ReasonerRegistry stores factory instances indexed by URIs.
- Reasoners can be retrieved using these indexes,
There are other built-in reasoners than those that are accessible through
  - the convenience methods on ModelFactory,
  - and on ReasonerRegistry,
  - for instance the GenericRuleReasoner.

All reasoners can be looked up in the registry.
The ReasonerRegistry stores factory instances indexed by URIs.
Reasoners can be retrieved using these indexes,
by registry.create(reasonerURI, param)
There are other built-in reasoners than those that are accessible through
- the convenience methods on ModelFactory,
- and on ReasonerRegistry,
- for instance the GenericRuleReasoner.

All reasoners can be looked up in the registry.

The ReasonerRegistry stores factory instances indexed by URIs.

Reasoners can be retrieved using these indexes,
by `registry.create(reasonerURI, param)`
- where `param` is a configuration parameter,
There are other built-in reasoners than those that are accessible through
- the convenience methods on ModelFactory,
- and on ReasonerRegistry,
- for instance the GenericRuleReasoner.

All reasoners can be looked up in the registry.
The ReasonerRegistry stores factory instances indexed by URIs.
Reasoners can be retrieved using these indexes,
by registry.create(reasonerURI, param)
  - where param is a configuration parameter,
  - of type Resource,
There are other built-in reasoners than those that are accessible through
- the convenience methods on ModelFactory,
- and on ReasonerRegistry,
- for instance the GenericRuleReasoner.

All reasoners can be looked up in the registry.
The ReasonerRegistry stores factory instances indexed by URIs.
Reasoners can be retrieved using these indexes,
by registry.create(reasonerURI, param)
  - where param is a configuration parameter,
  - of type Resource,
  - but it doesn’t do much,
There are other built-in reasoners than those that are accessible through
  - the convenience methods on ModelFactory,
  - and on ReasonerRegistry,
  - for instance the GenericRuleReasoner.

All reasoners can be looked up in the registry.
The ReasonerRegistry stores factory instances indexed by URIs.
Reasoners can be retrieved using these indexes,
by registry.create(reasonerURI, param)
  - where param is a configuration parameter,
  - of type Resource,
  - but it doesn’t do much,
  - and is usually replaced with null.
Using the built-in reasoners

Inspecting the registry

Obtaining an inventory

Get the single global instance of the registry:

```java
ReasonerRegistry reg = ReasonerRegistry.theRegistry();
```

Return a description of all reasoners in the form of an RDF graph:

```java
Model m = reg.getAllDescriptions();
```

Querying the inventory

```sql
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?reasoner ?desc WHERE {
  ?reasoner rdf:type jr:ReasonerClass .
}
```
Using the built-in reasoners

Inspecting the registry

**Obtaining an inventory**

Get the single global instance of the registry:

```java
ReasonerRegistry reg = ReasonerRegistry.theRegistry();
Model m = reg.getAllDescriptions();
```
Using the built-in reasoners

Inspecting the registry

Obtaining an inventory

Get the single global instance of the registry:

```java
ReasonerRegistry reg = ReasonerRegistry.theRegistry();
```
Using the built-in reasoners

Inspecting the registry

Obtaining an inventory

Get the single global instance of the registry:

   ReasonerRegistry reg = ReasonerRegistry.theRegistry();

Return a description of all reasoners in the form of an RDF graph:

PREXRDF: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?reasoner ?desc WHERE {
    ?reasoner rdf:type jr:ReasonerClass .
}
Using the built-in reasoners

Inspecting the registry

Obtaining an inventory

Get the single global instance of the registry:

```java
ReasonerRegistry reg = ReasonerRegistry.theRegistry();
```

Return a description of all reasoners in the form of an RDF graph:

```java
Model m = reg.getAllDescriptions();
```
Using the built-in reasoners

Inspecting the registry

**Obtaining an inventory**

Get the single global instance of the registry:

```java
ReasonerRegistry reg = ReasonerRegistry.theRegistry();
```

Return a description of all reasoners in the form of an RDF graph:

```java
Model m = reg.getAllDescriptions();
```

**Querying the inventory**

```sql
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?reasoner ?desc WHERE {
    ?reasoner rdf:type jr:ReasonerClass .
}
```
**InfModels by lookup**

### Reasoners and descriptions

<table>
<thead>
<tr>
<th>reasoner</th>
<th>desc</th>
</tr>
</thead>
<tbody>
<tr>
<td>jr:DIGReasoner</td>
<td>&quot;Adapter for external (i.e. non-Jena) DIG reasoner&quot;</td>
</tr>
<tr>
<td>jr:GenericRuleReasoner</td>
<td>&quot;Generic rule reasoner, configurable&quot;</td>
</tr>
<tr>
<td>jr:OWLFBRuleReasoner</td>
<td>&quot;Experimental OWL reasoner. Can separate tbox ...&quot;</td>
</tr>
<tr>
<td>jr:OWLMiniFBRuleReasoner</td>
<td>&quot;Experimental mini OWL reasoner. Can separate tbox ..&quot;</td>
</tr>
<tr>
<td>jr:OWLMicroFBRuleReasoner</td>
<td>&quot;Experimental mini OWL reasoner. Can separate ..&quot;</td>
</tr>
<tr>
<td>jr:TransitiveReasoner</td>
<td>&quot;Provides reflexive-transitive closure of subClassOf ...&quot;</td>
</tr>
<tr>
<td>jr:RDFSExptRuleReasoner</td>
<td>&quot;Complete RDFS implementation supporting ...&quot;</td>
</tr>
<tr>
<td>jr:DAMLMicroReasonerFactory</td>
<td>&quot;RDFS rule set with small extensions to support DAML&quot;</td>
</tr>
</tbody>
</table>

### Retrieveing a reasoner by URI

```java
ReasonerRegistry reg = ReasonerRegistry.theRegistry();
Reasoner r = reg.create("jr:OWLFBRuleReasoner", null);
InfModel inf = ModelFactory.createInfModel(r, sche, dat);
```
Using the built-in reasoners

InfModels by lookup

Reasoners and descriptions

<table>
<thead>
<tr>
<th>reasoner</th>
<th>desc</th>
</tr>
</thead>
<tbody>
<tr>
<td>jr::DIGReasoner</td>
<td>&quot;Adapter for external (i.e. non-Jena) DIG reasoner&quot;</td>
</tr>
<tr>
<td>jr::GenericRuleReasoner</td>
<td>&quot;Generic rule reasoner, configurable&quot;</td>
</tr>
<tr>
<td>jr::OWLFBRuleReasoner</td>
<td>&quot;Experimental OWL reasoner. Can separate tbox ...&quot;</td>
</tr>
<tr>
<td>jr::OWLMiniFBRuleReasoner</td>
<td>&quot;Experimental mini OWL reasoner. Can separate tbox ..&quot;</td>
</tr>
<tr>
<td>jr::OWLMicroFBRuleReasoner</td>
<td>&quot;Experimental mini OWL reasoner. Can separate ..&quot;</td>
</tr>
<tr>
<td>jr::TransitiveReasoner</td>
<td>&quot;Provides reflexive-transitive closure of subClassOf ...&quot;</td>
</tr>
<tr>
<td>jr::RDFSExptRuleReasoner</td>
<td>&quot;Complete RDFS implementation supporting ...&quot;</td>
</tr>
<tr>
<td>jr::DAMLMicroReasonerFactory</td>
<td>&quot;RDFS rule set with small extensions to support DAML&quot;</td>
</tr>
</tbody>
</table>

Retrieveing a reasoner by URI

ReasonerRegistry reg = ReasonerRegistry.theRegistry();
Reasoner r = reg.create("jr:OWLFBRuleReasoner", null);
InfModel inf = ModelFactory.createInfModel(r, sche, dat);
Richer models with \textit{OntModel}

- InfModels do not enhance the Model API \textit{as such},
InfModels do not enhance the Model API as such,
they only provide basic functionality associated with the reasoner.
Richer models with *OntModel*

- InfModels do not enhance the Model API as such,
- they only provide basic functionality associated with the reasoner.
Richer models with \textit{OntModel}

- InfModels do not enhance the Model API \textbf{as such},
- they only provide basic functionality associated with the reasoner.

An OntModel on the other hand
Richer models with *OntModel*

- InfModels do not enhance the Model API as such,
- they only provide basic functionality associated with the reasoner.

An OntModel on the other hand
- Provides a better view of a Model known to contain ontology data.
Richer models with \textit{OntModel}

- InfModels do not enhance the Model API as such,
- they only provide basic functionality associated with the reasoner.

An OntModel on the other hand
- Provides a better view of a Model known to contain ontology data.
- It supplies methods such as
Richer models with *OntModel*

- InfModels do not enhance the Model API as such,
- they only provide basic functionality associated with the reasoner.

An *OntModel* on the other hand
- Provides a better view of a Model known to contain ontology data.
- It supplies methods such as
  - createCardinalityRestriction,
Richer models with *OntModel*

- InfModels do not enhance the Model API *as such,*
- they only provide basic functionality associated with the reasoner.

An OntModel on the other hand
- Provides a better view of a Model known to contain ontology data.
- It supplies methods such as
  - createCardinalityRestriction,
  - createSymmetricProperty,
Richer models with *OntModel*

- InfModels do not enhance the Model API as such,
- they only provide basic functionality associated with the reasoner.

An OntModel on the other hand

- Provides a better view of a Model known to contain ontology data.
- It supplies methods such as
  - `createCardinalityRestriction`,
  - `createSymmetricProperty`,
  - `createRestriction`
Richer models with *OntModel*

- InfModels do not enhance the Model API as such,
- they only provide basic functionality associated with the reasoner.

An *OntModel* on the other hand

- Provides a better view of a Model known to contain ontology data.
- It supplies methods such as
  - `createCardinalityRestriction`,
  - `createSymmetricProperty`,
  - `createRestriction`

- Correspond to language constructs in OWL.
Richer models with *OntModel*

- InfModels do not enhance the Model API as such,
- they only provide basic functionality associated with the reasoner.

An *OntModel* on the other hand

- Provides a better view of a Model known to contain ontology data.
- It supplies methods such as
  - `createCardinalityRestriction`,
  - `createSymmetricProperty`,
  - `createRestriction`
- Correspond to language constructs in OWL.
- Required for manipulation of ontologies.
contd.

An *OntModel* does not by itself compute a deductive extension

- It is just an API.
A **OntModel** does not by itself compute a deductive extension

- It is just an API.
- However, it may obviously be hooked up with a reasoner.
Using the built-in reasoners

An *OntModel* does not by itself compute a deductive extension

- It is just an API.
- However, it may obviously be hooked up with a reasoner.
- Again we pass a message to ModelFactory,
An *OntModel* does not by itself compute a deductive extension

- It is just an API.
- However, it may obviously be hooked up with a reasoner.
- Again we pass a message to `ModelFactory`,
- only this time we do not supply a reasoner as an argument,
An *OntModel* does not by itself compute a deductive extension

- It is just an API.
- However, it may obviously be hooked up with a reasoner.
- Again we pass a message to ModelFactory,
- only this time we do not supply a reasoner as an argument,
- rather we supply a *model specification*,

contd.
An *OntModel* does not by itself compute a deductive extension

- It is just an API.
- However, it may obviously be hooked up with a reasoner.
- Again we pass a message to ModelFactory,
- only this time we do not supply a reasoner as an argument,
- rather we supply a *model specification*,
- which is an *OntModelSpec* object,
An *OntModel* does not by itself compute a deductive extension

- It is just an API.
- However, it may obviously be hooked up with a reasoner.
- Again we pass a message to `ModelFactory`,
- only this time we do not supply a reasoner as an argument,
- rather we supply a *model specification*,
- which is an `OntModelSpec` object,
- that encapsulates a description of `OntModel` components;
A *OntModel* does not by itself compute a deductive extension

- It is just an API.
- However, it may obviously be hooked up with a reasoner.
- Again we pass a message to ModelFactory,
- only this time we do not supply a reasoner as an argument,
- rather we supply a *model specification*,
- which is an *OntModelSpec* object,
- that encapsulates a description of *OntModel* components;
  - the storage scheme,
An *OntModel* does not by itself compute a deductive extension

- It is just an API.
- However, it may obviously be hooked up with a reasoner.
- Again we pass a message to `ModelFactory`,
- only this time we do not supply a reasoner as an argument,
- rather we supply a *model specification*,
- which is an `OntModelSpec` object,
- that encapsulates a description of `OntModel` components;
  - the storage scheme,
  - language profile,
An *OntModel* does not by itself compute a deductive extension

- It is just an API.
- However, it may obviously be hooked up with a reasoner.
- Again we pass a message to ModelFactory,
- only this time we do not supply a reasoner as an argument,
- rather we supply a *model specification*,
- which is an OntModelSpec object,
- that encapsulates a description of OntModel components;
  - the storage scheme,
  - language profile,
  - and the reasoner
An *OntModel* does not by itself compute a deductive extension

- It is just an API.
- However, it may obviously be hooked up with a reasoner.
- Again we pass a message to `ModelFactory`,
- only this time we do not supply a reasoner as an argument,
- rather we supply a *model specification*,
- which is an `OntModelSpec` object,
- that encapsulates a description of `OntModel` components;
  - the storage scheme,
  - language profile,
  - and the reasoner
- It is thus quite flexible and extensible.
Some specs from OntModelSpec

- **OWL DL MEM**: A specification for OWL DL models that are stored in memory and use the RDFS inferencer for additional entailments.
- **OWL LITE MEM**: A specification for OWL Lite models that are stored in memory and do no entailment additional reasoning.
- **OWL MEM MICRO RULE INF**: A specification for OWL models that are stored in memory and use the micro OWL rules inference engine for additional entailments.
- **OWL DL MEM**: A specification for OWL DL models that are stored in memory and do no additional entailment reasoning.
Using the built-in reasoners

Some specs from OntModelSpec

The class OntModelSpec contains static descriptive fields:
Some specs from OntModelSpec

The class OntModelSpec contains static descriptive fields:

**OWL_DL_MEM_RDFS_INF**: A specification for OWL DL models that are stored in memory and use the RDFS inferencer for additional entailments.
Some specs from OntModelSpec

The class OntModelSpec contains static descriptive fields:

**OWL_DL_MEM_RDFS_INF**: A specification for OWL DL models that are stored in memory and use the RDFS inferencer for additional entailments.

**OWL_LITE_MEM**: A specification for OWL Lite models that are stored in memory and do no entailment additional reasoning.
Some specs from OntModelSpec

The class OntModelSpec contains static descriptive fields:

**OWL_DL_MEM_RDFS_INF**: A specification for OWL DL models that are stored in memory and use the RDFS inferencer for additional entailments.

**OWL_LITE_MEM**: A specification for OWL Lite models that are stored in memory and do no entailment additional reasoning.

**OWL_MEM_MICRO_RULE_INF**: A specification for OWL models that are stored in memory and use the micro OWL rules inference engine for additional entailments.
Some specs from OntModelSpec

The class OntModelSpec contains static descriptive fields:

**OWL_DL_MEM_RDFS_INF**: A specification for OWL DL models that are stored in memory and use the RDFS inferencer for additional entailments.

**OWL_LITE_MEM**: A specification for OWL Lite models that are stored in memory and do no entailment additional reasoning.

**OWL_MEM_MICRO_RULE_INF**: A specification for OWL models that are stored in memory and use the micro OWL rules inference engine for additional entailments.

**OWL_DL_MEM**: A specification for OWL DL models that are stored in memory and do no additional entailment reasoning.
Creating OntModels with ModelFactory

Specifying an OntModel

```java
OntModelSpec spec = new OntModelSpec(OntModelSpec.OWL_DL_MEM);
OntModel model = ModelFactory.createOntologyModel(spec, model);
```
Creating OntModels with ModelFactory

Specifying an OntModel

```java
OntModelSpec spec = new OntModelSpec(OntModelSpec.OWL_DL_MEM);
OntModel model = ModelFactory.createOntologyModel(spec, model);
```

Note:
Using the built-in reasoners

Creating OntModels with ModelFactory

Specifying an OntModel

OntModelSpec spec = new OntModelSpec(OntModelSpec.OWL_DL_MEM);
OntModel model = ModelFactory.createOntologyModel(spec, model);

Note:

- Jena currently lags behind a bit, as there is no spec. for OWL 2.
Using the built-in reasoners

Creating OntModels with ModelFactory

Specifying an OntModel

```java
OntModelSpec spec = new OntModelSpec(OntModelSpec.OWL_DL_MEM);
OntModel model = ModelFactory.createOntologyModel(spec, model);
```

Note:

- Jena currently lags behind a bit, as there is no spec. for OWL 2.
- or any of its profiles
Using the built-in reasoners

Creating OntModels with ModelFactory

Specifying an OntModel

```java
OntModelSpec spec = new OntModelSpec(OntModelSpec.OWL_DL_MEM);
OntModel model = ModelFactory.createOntologyModel(spec, model);
```

Note:

- Jena currently lags behind a bit, as there is no spec. for OWL 2.
  - or any of its profiles
- Does not mean that one cannot use OWL 2 ontologies with Jena.
Using the built-in reasoners

Creating OntModels with ModelFactory

Specifying an OntModel

```java
OntModelSpec spec = new OntModelSpec(OntModelSpec.OWL_DL_MEM);
OntModel model = ModelFactory.createOntologyModel(spec, model);
```

Note:

- Jena currently lags behind a bit, as there is no spec. for OWL 2.
- or any of its profiles
- Does not mean that one cannot use OWL 2 ontologies with Jena.
  - If the reasoner handles OWL 2 (as e.g. Pellet does),
Creating OntModels with ModelFactory

Specifying an OntModel

```java
OntModelSpec spec = new OntModelSpec(OntModelSpec.OWL_DL_MEM);
OntModel model = ModelFactory.createOntologyModel(spec, model);
```

Note:

- Jena currently lags behind a bit, as there is no spec. for OWL 2.
  - or any of its profiles
- Does not mean that one cannot use OWL 2 ontologies with Jena.
  - If the reasoner handles OWL 2 (as e.g. Pellet does),
  - then Jena can reason with it (that is, with OWL 2 ontologies),
Creating OntModels with ModelFactory

Specifying an OntModel

OntModelSpec spec = new OntModelSpec(OntModelSpec.OWL_DL_MEM);
OntModel model = ModelFactory.createOntologyModel(spec, model);

Note:

- Jena currently lags behind a bit, as there is no spec. for OWL 2.
  - or any of its profiles
- Does not mean that one cannot use OWL 2 ontologies with Jena.
  - If the reasoner handles OWL 2 (as e.g. Pellet does),
  - then Jena can reason with it (that is, with OWL 2 ontologies),
  - but there may not be support in the API for all language constructs,
Creating OntModels with ModelFactory

Specifying an OntModel

```java
OntModelSpec spec = new OntModelSpec(OntModelSpec.OWL_DL_MEM);
OntModel model = ModelFactory.createOntologyModel(spec, model);
```

Note:

- Jena currently lags behind a bit, as there is no spec. for OWL 2.
  - or any of its profiles
- Does not mean that one cannot use OWL 2 ontologies with Jena.
  - If the reasoner handles OWL 2 (as e.g. Pellet does),
  - then Jena can reason with it (that is, with OWL 2 ontologies),
  - but there may not be support in the API for all language constructs,
  - parts of the ontology may not be directly accessible from the code.
Using the built-in reasoners

Creating OntModels with ModelFactory

Specifying an OntModel

OntModelSpec spec = new OntModelSpec(OntModelSpec.OWL_DL_MEM);
OntModel model = ModelFactory.createOntologyModel(spec, model);

Note:

- Jena currently lags behind a bit, as there is no spec. for OWL 2.
  - or any of its profiles
- Does not mean that one cannot use OWL 2 ontologies with Jena.
  - If the reasoner handles OWL 2 (as e.g. Pellet does),
  - then Jena can reason with it (that is, with OWL 2 ontologies),
  - but there may not be support in the API for all language constructs,
  - parts of the ontology may not be directly accessible from the code.
- Likely to change with new releases of Jena.
Outline

1. Jena inference support
2. Using the built-in reasoners
3. Using an external reasoner
4. Simple reasoner configuration
5. Introduction to OWL
External reasoners are best manipulated directly, that is

- One goes directly to the FactoryClass,
External reasoners are best manipulated directly, that is

- One goes directly to the FactoryClass,
- calls the static `theInstance()` to get the factory instance,
External reasoners are best manipulated directly, that is

- One goes directly to the FactoryClass,
- calls the static `theInstance()` to get the factory instance,
- calls the instance’s `create()` method,
External reasoners are best manipulated directly, that is

- One goes directly to the FactoryClass,
- calls the static `theInstance()` to get the factory instance,
- calls the instance’s `create()` method,
- and gets the associated reasoner in return.
External reasoners are best manipulated directly, that is:

- One goes directly to the FactoryClass,
- calls the static `theInstance()` to get the factory instance,
- calls the instance’s `create()` method,
- and gets the associated reasoner in return.
External reasoners are best manipulated directly, that is:

- One goes directly to the FactoryClass,
- calls the static `theInstance()` to get the factory instance,
- calls the instance’s `create()` method,
- and gets the associated reasoner in return.

External reasoners can be combined with InfModels and OntModels alike.
In the former case, things are very simple:
In the former case, things are very simple:

Using Pellet with an **InfModel**

```java
Reasoner reas = PelletReasonerFactory.theInstance().create();
InfModel inf = ModelFactory.createInfModel(reas, sche, dat);
```
In the former case, things are very simple:

Using Pellet with an InfModel

```java
Reasoner reas = PelletReasonerFactory.theInstance().create();
InfModel inf = ModelFactory.createInfModel(reas, sche, dat);
```

The latter case requires a little more tweaking:
contd.

In the former case, things are very simple:

**Using Pellet with an InfModel**

```java
Reasoner reas = PelletReasonerFactory.theInstance().create();
InfModel inf = ModelFactory.createInfModel(reas, sche, dat);
```

The latter case requires a little more tweaking:

**Using Pellet with an OntModel**

```java
Reasoner r = PelletReasonerFactory.theInstance().create();
InfModel mod = ModelFactory.createInfModel(r, s, d);
OntModelSpec spec = new OntModelSpec(OntModelSpec.OWL_DL_MEM);
OntModel ont = ModelFactory.createOntologyModel(spec, mod);
```
Outline

1. Jena inference support
2. Using the built-in reasoners
3. Using an external reasoner
4. Simple reasoner configuration
5. Introduction to OWL
Reasoners can be configured in many ways:
Reasoners can be configured in many ways:

- Some can be configured to reason in different directions, that is
Reasoners can be configured in many ways:

- Some can be configured to reason in different directions, that is
  - from conclusions to premises (so-called backwards chaining),
Reasoners can be configured in many ways:

- Some can be configured to reason in different directions, that is
  - from conclusions to premises (so-called backwards chaining),
  - from premises to conclusion (so-called forwards chaining),
Configuration in general

Reasoners can be configured in many ways:

- Some can be configured to reason in different directions, that is
  - from conclusions to premises (so-called backwards chaining),
  - from premises to conclusion (so-called forwards chaining),
  - or a mix (so-called hybrid reasoning)
Reasoners can be configured in many ways:

- Some can be configured to reason in different directions, that is
  - from conclusions to premises (so-called backwards chaining),
  - from premises to conclusion (so-called forwards chaining),
  - or a mix (so-called hybrid reasoning)
- or to turn transitivity off for properties such as `subClassOf`,
Reasoners can be configured in many ways:

- Some can be configured to reason in different directions, that is
  - from conclusions to premises (so-called backwards chaining),
  - from premises to conclusion (so-called forwards chaining),
  - or a mix (so-called hybrid reasoning)
- or to turn transitivity off for properties such as subClassOf,
- or to log derivations.
Reasoners can be configured in many ways:

- Some can be configured to reason in different directions, that is
  - from conclusions to premises (so-called backwards chaining),
  - from premises to conclusion (so-called forwards chaining),
  - or a mix (so-called hybrid reasoning)
- or to turn transitivity off for properties such as `subClassOf`,
- or to log derivations.
Reasoners can be configured in many ways:

- Some can be configured to reason in different directions, that is
  - from conclusions to premises (so-called backwards chaining),
  - from premises to conclusion (so-called forwards chaining),
  - or a mix (so-called hybrid reasoning)

- or to turn transitivity off for properties such as subClassOf,
- or to log derivations.

In every case you will need a reference to the reasoner, whence
Reasoners can be configured in many ways:

- Some can be configured to reason in different directions, that is
  - from conclusions to premises (so-called backwards chaining),
  - from premises to conclusion (so-called forwards chaining),
  - or a mix (so-called hybrid reasoning)
- or to turn transitivity off for properties such as subClassOf,
- or to log derivations.

In every case you will need a reference to the reasoner, whence

- it is no longer convenient to use the convenience methods in ModelFactory.
Specializing the reasoner

The simplest way to configure a reasoner is to specialize it:

```java
Reasoner r = PelletReasonerFactory.theInstance().create();
Reasoner custom = r.bindSchema(schema);
InfModel inf = ModelFactory.createInfModel(custom, data);
```
The simplest way to configure a reasoner is to specialize it:

- that is, to bind it to a particular ontology.
Specializing the reasoner

The simplest way to configure a reasoner is to **specialize it**:

- that is, to **bind** it to a particular ontology.
The simplest way to configure a reasoner is to specialize it:

- that is, to bind it to a particular ontology.

This is suitable for situations where,
The simplest way to configure a reasoner is to **specialize it**: that is, to **bind** it to a particular ontology.

This is suitable for situations where,

- you want to apply the same schema to several data sets,
Specializing the reasoner

The simplest way to configure a reasoner is to specialize it:

- that is, to **bind** it to a particular ontology.

This is suitable for situations where,

- you want to apply the same schema to several data sets,
- without redoing too many intermediate deductions.
The simplest way to configure a reasoner is to specialize it:

- that is, to bind it to a particular ontology.

This is suitable for situations where,

- you want to apply the same schema to several data sets,
- without redoing too many intermediate deductions.

```java
Reasoner r = PelletReasonerFactory.theInstance().create();
Reasoner custom = r.bindSchema(schema);
InfModel inf = ModelFactory.createInfModel(custom, data);
```
Simple reasoner configuration

Specializing the reasoner

The simplest way to configure a reasoner is to specialize it:

- that is, to **bind** it to a particular ontology.

This is suitable for situations where,

- you want to apply the same schema to several data sets,
- without redoing too many intermediate deductions

**Binding Pellet to schema**

```java
Reasoner r = PelletReasonerFactory.theInstance().create();
Reasoner custom = r.bindSchema(schema);
InfModel inf = ModelFactory.createInfModel(custom, data);
```
A very simple taxonomy

Consider again the RDFS ontology given by:

\[ \text{ex:KillerWhale a rdfs:Class} . \]
\[ \text{ex:Mammal a rdfs:Class} . \]
\[ \text{ex:Vertebrate a rdfs:Class} . \]

\[ \text{ex:KillerWhale rdfs:subClassOf ex:Mammal} . \]
\[ \text{ex:Mammal rdfs:subClassOf ex:Vertebrate} . \]

And suppose we assert:

\[ \text{ex:Keiko a ex:KillerWhale} . \]

Tracing the derivations could be useful for

- debugging,
- automatic explanation.
Logging derivations

Telling the reasoner to log derivations

```java
Reasoner r = ReasonerRegistry.getRDFSReasoner();
r.setDerivationLogging(true);
```

Printing derivations

```java
PrintWriter out = new PrintWriter(System.out);
StmtIterator it = inf.listStatements();

while(it.hasNext()){
    Statement stat = (Statement) it.next();
    for(Iterator id = inf.getDerivation(stat);id.hasNext();)
    {
        Derivation deriv = (Derivation) id.next();
        deriv.printTrace(out, true);
    }
}
```
Rule rdfs9-alt concluded (ex:Keiko rdf:type ex:Vertebrate) <-
Fact (ex:KillerWhale rdfs:subClassOf ex:Vertebrate)
Rule rdfs9-alt concluded (ex:Keiko rdf:type ex:KillerWhale) <-
Fact (ex:KillerWhale rdfs:subClassOf ex:KillerWhale)
Known (ex:Keiko rdf:type ex:KillerWhale) - already shown
Outline

1. Jena inference support
2. Using the built-in reasoners
3. Using an external reasoner
4. Simple reasoner configuration
5. Introduction to OWL
Quick facts

OWL:

- Acronym for *The Web Ontology Language*.
Quick facts

OWL:

- Acronym for *The Web Ontology Language*.
- Became a W3C recommendation in 2004.
Quick facts

OWL:
- Acronym for *The Web Ontology Language*.
- Became a W3C recommendation in 2004.
- Enables **boolean** reasoning over classes and relationships.
Quick facts

OWL:

- Acronym for *The Web Ontology Language*.
- Became a W3C recommendation in 2004.
- Enables *boolean* reasoning over classes and relationships.
- Superseded by OWL 2;
Quick facts

OWL:
- Acronym for *The Web Ontology Language.*
- Became a W3C recommendation in 2004.
- Enables **boolean** reasoning over classes and relationships.
- Superseded by OWL 2;
  - a backwards compatible extension that adds new capabilities.
Quick facts

OWL:
- Acronym for *The Web Ontology Language*.
- Became a W3C recommendation in 2004.
- Enables *boolean* reasoning over classes and relationships.
- Superseded by OWL 2;
  - a backwards compatible extension that adds new capabilities.
- The OWL family of languages are based on *Description Logics*.
Quick facts

OWL:

- Acronym for *The Web Ontology Language*.
- Became a W3C recommendation in 2004.
- Enables *boolean* reasoning over classes and relationships.
- Superseded by OWL 2;
  - a backwards compatible extension that adds new capabilities.
- The OWL family of languages are based on *Description Logics*.
- DLs have well-understood and attractive computational properties.
Glimpse ahead: OWL profiles

- OWL has various profiles that correspond to different DLs.
Glimpse ahead: OWL profiles

- OWL has various profiles that correspond to different DLs.
- These profiles are tailored for specific ends, e.g.

  - OWL 2 QL: Specifically designed for efficient database integration.
  - OWL 2 EL: A lightweight language with polynomial time reasoning. Much used in medical informatics (e.g. the GALEN ontology).
  - OWL 2 RL: Designed for compatibility with rule-based inference tools.
OWL has various profiles that correspond to different DLs. These profiles are tailored for specific ends, e.g.

- **OWL 2 QL:** Specifically designed for efficient database integration.
- **OWL 2 EL:** A lightweight language with polynomial time reasoning. Much used in medical informatics (e.g. the GALEN ontology).
- **OWL 2 RL:** Designed for compatibility with rule-based inference tools.
Glimpse ahead: OWL profiles

- OWL has various profiles that correspond to different DLs.
- These profiles are tailored for specific ends, e.g.
  - OWL 2 QL:
    - Specifically designed for efficient database integration.
OWL has various profiles that correspond to different DLs.

These profiles are tailored for specific ends, e.g.

- **OWL 2 QL:**
  - Specifically designed for efficient database integration.

- **OWL 2 EL:**
OWL has various \textit{profiles} that correspond to different DLs. These profiles are tailored for specific ends, e.g.

- **OWL 2 QL:**
  - Specifically designed for efficient database integration.

- **OWL 2 EL:**
  - A lightweight language with polynomial time reasoning.
Glimpse ahead: OWL profiles

- OWL has various profiles that correspond to different DLs.
- These profiles are tailored for specific ends, e.g.
  - **OWL 2 QL:**
    - Specifically designed for efficient database integration.
  - **OWL 2 EL:**
    - A lightweight language with polynomial time reasoning.
    - Much used in medical informatics (e.g. the GALEN ontology).
• OWL has various profiles that correspond to different DLs.
• These profiles are tailored for specific ends, e.g.
  • **OWL 2 QL:**
    • Specifically designed for efficient database integration.
  • **OWL 2 EL:**
    • A lightweight language with polynomial time reasoning.
    • Much used in medical informatics (e.g. the GALEN ontology).
  • **OWL 2 RL:**
OWL has various profiles that correspond to different DLs.

These profiles are tailored for specific ends, e.g.

- **OWL 2 QL**:
  - Specifically designed for efficient database integration.

- **OWL 2 EL**:
  - A lightweight language with polynomial time reasoning.
  - Much used in medical informatics (e.g. the GALEN ontology).

- **OWL 2 RL**:
  - Designed for compatibility with rule-based inference tools.
### The ALEC fragment of OWL

#### ALEC In DL-notation

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C, D \rightarrow A$</td>
<td>(atomic concept)</td>
</tr>
<tr>
<td>$\top$</td>
<td>(universal concept)</td>
</tr>
<tr>
<td>$\bot$</td>
<td>(bottom concept)</td>
</tr>
<tr>
<td>$\neg C$</td>
<td>(atomic negation)</td>
</tr>
<tr>
<td>$C \sqcap D$</td>
<td>(intersection)</td>
</tr>
<tr>
<td>$\forall R.C$</td>
<td>(value restriction)</td>
</tr>
<tr>
<td>$\exists R.C$</td>
<td>(existential restriction)</td>
</tr>
</tbody>
</table>
**ALCE in DL-notation**

\[ \top^I = \Delta^I \]
\[ \bot^I = \emptyset \]
\[ (\neg C)^I = \Delta^I \setminus C^I \]
\[ (C \sqcap D)^I = C^I \cap D^I \]
\[ (\forall R.C)^I = \{ a \in \Delta^I \mid \forall b(a, b) \in R^I \rightarrow b \in C^I \} \]
\[ (\exists R.C)^I = \{ a \in \Delta^I \mid \exists b(a, b) \in R^I \land b \in C^I \} \]

**OWL ontologies in DL-notation**

Cystic_Fibrosis \equiv Fibrosis \sqcap \exists locatedIn.Pancreas

Genetic_Fibrosis \sqsubseteq Genetic_Disorder

Fibrosis \sqcap \exists locatedIn.Pancreas \sqsubseteq Genetic_Fibrosis
Some differences from RDFS

1. Complex classes can be expressed:

- $C \cap D$ corresponds to logical conjunction,
- $C \cup D$ to logical disjunction, and
- $\neg C$ to logical negation.

Unlike RDFS, OWL is therefore a boolean language. That is, it has a propositional logic as a fragment.

Full propositional negation facilitates consistency checking.
Some differences from RDFS

1. Complex classes can be expressed:
   - $C \sqcap D$ corresponds to logical conjunction,
Some differences from RDFS

1. Complex classes can be expressed:
   - $C \cap D$ corresponds to logical conjunction,
   - $C \cup D$ to logical disjunction, and
Some differences from RDFS

Complex classes can be expressed:
- $C \sqcap D$ corresponds to logical conjunction,
- $C \sqcup D$ to logical disjunction, and
- $\neg C$ to logical negation
Some differences from RDFS

1. Complex classes can be expressed:
   - $C \sqcap D$ corresponds to logical conjunction,
   - $C \sqcup D$ to logical disjunction, and
   - $\neg C$ to logical negation

2. Unlike RDFS, OWL is therefore a boolean language.
Some differences from RDFS

1. **Complex classes can be expressed:**
   - $C \sqcap D$ corresponds to logical **conjunction**,
   - $C \sqcup D$ to logical **disjunction**, and
   - $\neg C$ to logical **negation**

2. **Unlike RDFS, OWL is therefore a **boolean language.**
   - That is, it has a propositional logic as a fragment.
Some differences from RDFS

1. Complex classes can be expressed:
   - $C \cap D$ corresponds to logical conjunction,
   - $C \cup D$ to logical disjunction, and
   - $\neg C$ to logical negation

2. Unlike RDFS, OWL is therefore a boolean language.
   - That is, it has a propositional logic as a fragment.

3. Full propositional negation facilitates consistency checking.
Existential restrictions

- Allow us to describe classes in terms of each other.
Existential restrictions

- Allow us to describe classes in terms of each other.

\[ Cystic\_Fibrosis \equiv Fibrosis \cap \exists \text{locatedIn}.\text{Pancreas} \]
Existential restrictions

- Allow us to describe classes in terms of each other.

  \[ \text{Cystic Fibrosis} \equiv \text{Fibrosis} \sqcap \exists \text{locatedIn}.\text{Pancreas} \]

- Or, more mundanely
Existential restrictions

- Allow us to describe classes in terms of each other.

\[ \text{Cystic\_Fibrosis} \equiv \text{Fibrosis} \cap \exists \text{locatedIn}\cdot \text{Pancreas} \]

- or, more mundanely

\[ \text{ProudMother} \equiv \text{Woman} \cap \exists \text{hasChild}\cdot \text{Lawyer} \]
Existential restrictions

- Allow us to describe classes in terms of each other.
  
  \[ \text{Cystic Fibrosis} \equiv \text{Fibrosis} \cap \exists \text{locatedIn.Pancreas} \]

- or, more mundanely

  \[ \text{ProudMother} \equiv \text{Woman} \cap \exists \text{hasChild.Lawyer} \]

- \text{hasChild.Lawyer} = \text{the set of things that have at least one lawyer child.}
Existential restrictions

- Allow us to describe classes in terms of each other.
  \[ \text{Cystic F} \equiv \text{Fibrosis} \cap \exists \text{locatedIn.Pancreas} \]

- or, more mundanely
  \[ \text{ProudMother} \equiv \text{Woman} \cap \exists \text{hasChild.Lawyer} \]

- \text{hasChild.Lawyer} = \text{the set of things that have at least one lawyer child.}
  - If a thing has a lawyer child,
Existential restrictions

- Allow us to describe classes in terms of each other.

\[ Cystic\_Fibrosis \equiv Fibrosis \sqcap \exists locatedIn.\text{Pancreas} \]

- or, more mundanely

\[ ProudMother \equiv Woman \sqcap \exists hasChild.\text{Lawyer} \]

- \text{hasChild.Lawyer} \equiv \text{the set of things that have at least one lawyer child.}
  - If a thing has a lawyer child,
  - and that thing is a woman,
Existential restrictions

- Allow us to describe classes in terms of each other.

\[
Cystic\_Fibrosis \equiv Fibrosis \cap \exists locatedIn.Pancreas
\]

- or, more mundanely

\[
ProudMother \equiv Woman \cap \exists hasChild.Lawyer
\]

- \text{hasChild.Lawyer} = \text{the set of things that have at least one lawyer child.}
  - If a thing has a lawyer child,
  - and that thing is a woman,
  - then that thing is a proud mother
Existential restrictions in Turtle syntax

Lawyer children

[a owl:Restriction;
   owl:onProperty :hasChild:
   owl:somValuesFrom :Lawyer] .

- owl:Restriction signals a class description,
Existential restrictions in Turtle syntax

Lawyer children

```
[a owl:Restriction;
 owl:onProperty :hasChild:
 owl:somValuesFrom :Lawyer] .
```

- `owl:Restriction` signals a class description,
- `owl:somValuesFrom`; an existential restriction on a property,
Existential restrictions in Turtle syntax

**Lawyer children**

```
[a owl:Restriction;
  owl:onProperty :hasChild:
  owl:somValuesFrom :Lawyer] .
```

- `owl:Restriction` signals a class description,
- `owl:somValuesFrom`; an existential restriction on a property,
- `owl:onProperty` gives the property
Existential restrictions in Turtle syntax

Lawyer children

```
[a owl:Restriction;
 owl:onProperty :hasChild:
 owl:somValuesFrom :Lawyer] .
```

- `owl:Restriction` signals a class description,
- `owl:somValuesFrom`; an existential restriction on a property,
- `owl:onProperty` gives the property
- The description is a blank node, since it has no name.
Existential restrictions illustrated

Figure: Existential restrictions. From Julian Seidenberg "Web Ontology Segmentation: Extraction, Transformation, Evaluation"
Horisontal relations between classes

Figure: Existential restrictions relate classes (from Julian Seidenberg "Web Ontology Segmentation: Extraction, Transformation, Evaluation")
Returning to an example

Suppose we assert:
And we say that
2. Orchestra ≡ ∃conductor. ⊤ ⊓ ∃hasInstrument. ⊤

Then from [1.] we may infer that
3. :OsloPhilharmonic a Orchestra.
4. :OsloPhilharmonic :hasInstrument :x.
Suppose we assert:


And we say that

2. Orchestra ≡ ∃conductor.⊤ ∩ ∃hasInstrument.⊤

Then from [1.] we may infer that

3. :OsloPhilharmonic a :Orchestra .
4. :OsloPhilharmonic :hasInstrument _:_x .
A comparison with rdfs:domain

- Recall that ex:conductor rdfs:domain ex:Orchestra says that only orchestras have conductors.
A comparison with rdfs:domain

- Recall that ex:conductor rdfs:domain ex:Orchestra says that only orchestras have conductors.
- We can express this with existential restrictions:
A comparison with rdfs:domain

- Recall that ex:conductor rdfs:domain ex:Orchestra says that only orchestras have conductors.
- We can express this with existential restrictions:
  $$\exists\text{conductor}.\top \equiv \text{Orchestra}$$
A comparison with rdfs:domain

- Recall that ex:conductor rdfs:domain ex:Orchestra says that only orchestras have conductors.
- We can express this with existential restrictions:
  \[ \exists \text{conductor}. \top \equiv \text{Orchestra} \]
- But we can also express a number finer relationships:
A comparison with rdfs:domain

- Recall that ex:conductor rdfs:domain ex:Orchestra says that only orchestras have conductors.
- We can express this with existential restrictions:
  \[ \exists \text{conductor}. \top \equiv \text{Orchestra} \]
- But we can also express a number finer relationships:
  \[ \text{Choir} \sqsubseteq \exists \text{conductor.} \top \]
A comparison with rdfs:domain

- Recall that ex:conductor rdfs:domain ex:Orchestra says that only orchestras have conductors.
- We can express this with existential restrictions:
  \[ \exists \text{conductor}. \top \equiv \text{Orchestra} \]
- But we can also express a number finer relationships:
  \[ \text{Choir} \sqsubseteq \exists \text{conductor}. \top \]
  \[ \exists \text{conductor}. \text{Cantor} \sqsubseteq \text{ChurchEnsemble} \]
A comparison with rdfs:domain

- Recall that `ex:conductor rdfs:domain ex:Orchestra` says that only orchestras have conductors.
- We can express this with existential restrictions:
  \[ \exists \text{conductor}. \top \equiv \text{Orchestra} \]
- But we can also express a number finer relationships:
  \[ \text{Choir} \sqsubseteq \exists \text{conductor}. \top \]
  \[ \exists \text{conductor}. \text{Cantor} \sqsubseteq \text{ChurchEnsemble} \]
- each time we are relating classes to each other,
A comparison with rdfs:domain

- Recall that ex:conductor rdfs:domain ex:Orchestra says that only orchestras have conductors.
- We can express this with existential restrictions:
  \[ \exists \text{conductor}. \top \equiv \text{Orchestra} \]
- But we can also express a number finer relationships:
  \[ \text{Choir} \subseteq \exists \text{conductor}. \top \]
  \[ \exists \text{conductor}. \text{Cantor} \subseteq \text{ChurchEnsemble} \]
- each time we are relating classes to each other,
- weaving together a fabric of formalized knowledge,
A comparison with rdfs:domain

- Recall that `ex:conductor rdfs:domain ex:Orchestra` says that only orchestras have conductors.
- We can express this with existential restrictions:
  \[ \exists \text{conductor}. \top \equiv \text{Orchestra} \]
- But we can also express a number finer relationships:
  \[ \text{Choir} \sqsubseteq \exists \text{conductor}. \top \]
  \[ \exists \text{conductor}. \text{Cantor} \sqsubseteq \text{ChurchEnsemble} \]
- each time we are relating classes to each other,
- weaving together a fabric of formalized knowledge,
- which stores inferences like a battery stores energy.
A comparison with rdfs:domain

- Recall that ex:conductor rdfs:domain ex:Orchestra says that only orchestras have conductors.
- We can express this with existential restrictions:
  \[ \exists \text{conductor}. \top \equiv \text{Orchestra} \]
- But we can also express a number finer relationships:
  \[ \text{Choir} \sqsubseteq \exists \text{conductor}. \top \]
  \[ \exists \text{conductor}. \text{Cantor} \sqsubseteq \text{ChurchEnsemble} \]
- each time we are relating classes to each other,
- weaving together a fabric of formalized knowledge,
- which stores inferences like a battery stores energy.
- If we add that
A comparison with rdfs:domain

- Recall that ex:conductor rdfs:domain ex:Orchestra says that only orchestras have conductors.
- We can express this with existential restrictions:
  \[ \exists \text{conductor} . \top \equiv \text{Orchestra} \]
- But we can also express a number finer relationships:
  \[ \text{Choir} \sqsubseteq \exists \text{conductor} . \top \]
  \[ \exists \text{conductor} . \text{Cantor} \sqsubseteq \text{ChurchEnsemble} \]
- each time we are relating classes to each other,
- weaving together a fabric of formalized knowledge,
- which stores inferences like a battery stores energy.
- If we add that
  \[ :\text{MusicaAntiqua} :\text{conductor} :\text{Savall} . \] (not actually the case)
A comparison with rdfs:domain

- Recall that ex:conductor rdfs:domain ex:Orchestra says that only orchestras have conductors.
- We can express this with existential restrictions:
  \[ \exists \text{conductor}. \top \equiv \text{Orchestra} \]
- But we can also express a number finer relationships:
  \[
  \text{Choir} \sqsubseteq \exists \text{conductor}. \top \\
  \exists \text{conductor}. \text{Cantor} \sqsubseteq \text{ChurchEnsemble}
  \]
- each time we are relating classes to each other,
- weaving together a fabric of formalized knowledge,
- which stores inferences like a battery stores energy.
- If we add that
  \[
  \text{:MusicaAntiqua :conductor :Savall} \quad (\text{not actually the case}) \\
  \text{:Savall a :Cantor} \quad (\text{nor is this})
  \]
A comparison with rdfs:domain

- Recall that ex:conductor rdfs:domain ex:Orchestra says that only orchestras have conductors.

- We can express this with existential restrictions:
  \[ \exists \text{conductor}. \top \equiv \text{Orchestra} \]

- But we can also express a number finer relationships:
  \[ \text{Choir} \sqsubseteq \exists \text{conductor}. \top \]
  \[ \exists \text{conductor}. \text{Cantor} \sqsubseteq \text{ChurchEnsemble} \]

- each time we are relating classes to each other,
- weaving together a fabric of formalized knowledge,
- which stores inferences like a battery stores energy.

- If we add that
  \[ :\text{MusicaAntiqua} :\text{conductor} :\text{Savall} . \text{(not actually the case)} \]
  \[ :\text{Savall} \ a :\text{Cantor} \text{ (nor is this)} \]

- then we know that
A comparison with rdfs:domain

- Recall that ex:conductor rdfs:domain ex:Orchestra says that only orchestras have conductors.
- We can express this with existential restrictions:
  \[ \exists \text{conductor} . \top \equiv \text{Orchestra} \]
- But we can also express a number finer relationships:
  \[ \text{Choir} \sqsubseteq \exists \text{conductor}. \top \]
  \[ \exists \text{conductor}. \text{Cantor} \sqsubseteq \text{ChurchEnsemble} \]
- each time we are relating classes to each other,
- weaving together a fabric of formalized knowledge,
- which stores inferences like a battery stores energy.
- If we add that
  \[ :\text{MusicaAntiqua} :\text{conductor} :\text{Savall} . \text{(not actually the case)} \]
  \[ :\text{Savall} \text{ a} :\text{Cantor} \text{(nor is this)} \]
- then we know that
  \[ :\text{MusicaAntiqua} \text{ a} :\text{ChurchEnsemble} . \text{(nope)} \]
Introduction to OWL

Existential restrictions in OntModels

Implementing the example

```java
OntModel m = ModelFactory.createOntologyModel(OntModelSpec.OWL_DL_MEM);
OntClass c = m.createClass("ex:Cantor");
OntClass e = m.createClass("ex:ChurchEnsemble");
ObjectProperty cond = m.createObjectProperty("ex:conductor");

// null denotes the URI in an anonymous restriction
SomeValuesFromRestriction r = m.createSomeValuesFromRestriction(null, cond, c);
Statement stmt = model.createStatement(r,OWL.subClassOf, e);
model.add(stmt);
```

More about this later
Supplementary reading

- The Jena ontology API:
- Jena Inference Engine user manual:
- Using a DIG Description Logic reasoner with Jena:

All available from the Jena website.