INFO-H-509 XML TECHNOLOGIES

Lecture 8 OWL: Web Ontology Language

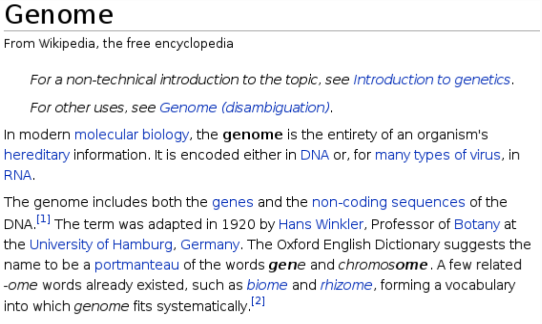
Stijn Vansummeren
April 2, 2015

LECTURE OUTLINE

1. Our story so far
2. Web Ontology Language—OWL
3. Reasoning with OWL

Part I: Our story so far

OUR STORY SO FAR ...



Genome

From Wikipedia, the free encyclopedia

For a non-technical introduction to the topic, see Introduction to genetics.

For other uses, see Genome (disambiguation).

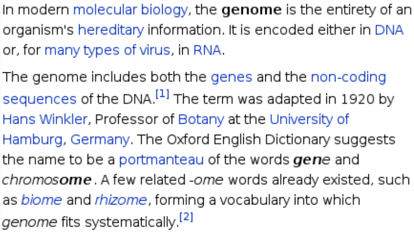
In modern molecular biology, the **genome** is the entirety of an organism's hereditary information. It is encoded either in DNA or, for many types of virus, in RNA.

The genome includes both the genes and the non-coding sequences of the DNA.[1] The term was adapted in 1920 by Hans Winkler, Professor of Botany at the University of Hamburg, Germany. The Oxford English Dictionary suggests the name to be a portmanteau of the words ***gen**e* and *chromos**ome***. A few related *-ome* words already existed, such as *biome* and *rhizome*, forming a vocabulary into which *genome* fits systematically.[2]

- Natural language
- No structure
- Difficult to process automatically

OUR STORY FAR ...

Recent past – no structure



In modern molecular biology, the **genome** is the entirety of an organism's hereditary information. It is encoded either in DNA or, for many types of virus, in RNA.

The genome includes both the genes and the non-coding sequences of the DNA.[1] The term was adapted in 1920 by Hans Winkler, Professor of Botany at the University of Hamburg, Germany. The Oxford English Dictionary suggests the name to be a portmanteau of the words ***gen*e** and *chromos***ome*** . A few related *-ome* words already existed, such as *biome* and *rhizome*, forming a vocabulary into which *genome* fits systematically.[2]

Current/Future structured by RDF (subject, predicate, object)

```
b:genome    b:field        b:molecular-bio   .
b:DNA       b:encode       b:genes           .
b:DNA       b:encode       b:non-coding-seq .
b:genome    b:include      b:non-coding-seq .
b:genome    b:include      b:gene            .
b:genome    b:related-to   b:rhizome         .
```

- RDF asserts knowledge (statements) about entities (resources)
- By convention is clear what the subject, predicate, and object are
- Easier to process automatically, but a computer still does not know their meaning ...
- How do we add some semantics to the statements?

WHAT DO YOU MEAN: SEMANTICS?

Input

```
@prefix prod: <http://www.example.org/products/> .
@prefix terms: <http://www.example.org/terms/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
prod:cam1     rdf:type       terms:digital-camera     .
prod:cam1     terms:price    150                      .
prod:nb1      rdf:type       terms:netbook            .
prod:nb1      terms:price    300                      .
prod:book1    rdf:type       terms:book               .
prod:book1    terms:price    2.50                     .
```

How do we find all products that are digital devices?

WHAT DO YOU MEAN: SEMANTICS?

Input

```
@prefix prod: <http://www.example.org/products/> .
@prefix terms: <http://www.example.org/terms/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
prod:cam1     rdf:type       terms:digital-camera     .
prod:cam1     terms:price    150                      .
prod:nb1      rdf:type       terms:netbook            .
prod:nb1      terms:price    300                      .
prod:book1    rdf:type       terms:book               .
prod:book1    terms:price    2.50                     .
```

How do we find all products that are digital devices?

Hmm, digital cameras are digital devices

select all *x* such that
x, `rdf:type`, `terms:digital-camera` .

WHAT DO YOU MEAN: SEMANTICS?

Input

```
@prefix prod: <http://www.example.org/products/> .
@prefix terms: <http://www.example.org/terms/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
prod:cam1     rdf:type       terms:digital-camera     .
prod:cam1     terms:price    150                      .
prod:nb1      rdf:type       terms:netbook            .
prod:nb1      terms:price    300                      .
prod:book1    rdf:type       terms:book               .
prod:book1    terms:price    2.50                     .
```

How do we find all products that are digital devices?

Hmm, digital cameras are digital devices
… so are netbooks

```
select all x such that
  x, rdf:type, terms:digital-camera .
OR
  x, rdf:type, terms:netbook .
```

WHAT DO YOU MEAN: SEMANTICS?

Input

```
@prefix prod: <http://www.example.org/products/> .
@prefix terms: <http://www.example.org/terms/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
prod:cam1     rdf:type       terms:digital-camera     .
prod:cam1     terms:price    150                      .
prod:nb1      rdf:type       terms:netbook            .
prod:nb1      terms:price    300                      .
prod:book1    rdf:type       terms:book               .
prod:book1    terms:price    2.50                     .
```

- The computer has no “knowledge of the world” stating that cameras and netbooks are digital devices
- So we have to manually encode this “knowledge of the world” in the query
- This solution is inadequate: error-prone and difficult to maintain
- It would be better if we could tell the computer our “knowledge of the world” and let him do the reasoning!

REASONING BY MEANS OF INFERENCE: THE GENERAL IDEA

Explicitly Asserted Knowledge

I am a man
John is a man
Jane is a woman

REASONING BY MEANS OF INFERENCE: THE GENERAL IDEA

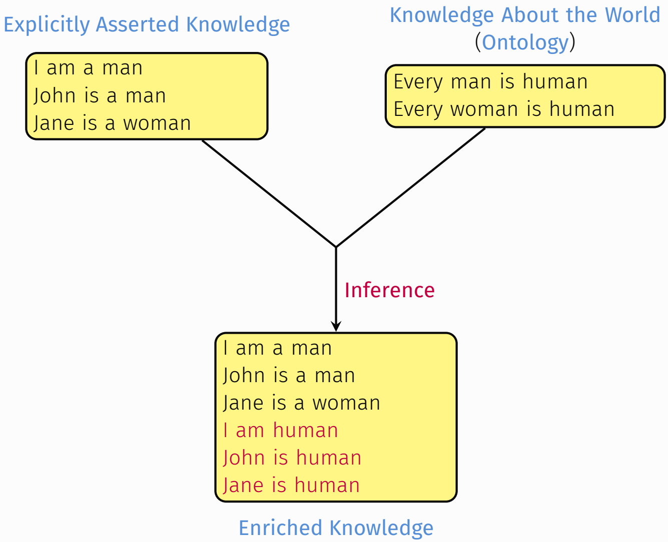
Explicitly Asserted Knowledge

I am a man
John is a man
Jane is a woman

Knowledge About the World
(Ontology)

Every man is human
Every woman is human

REASONING BY MEANS OF INFERENCE: THE GENERAL IDEA



Explicitly Asserted Knowledge

I am a man
John is a man
Jane is a woman

Knowledge About the World
(Ontology)

Every man is human
Every woman is human

Inference

I am a man
John is a man
Jane is a woman
I am human
John is human
Jane is human

Enriched Knowledge

TOWARDS A SMARTER WEB

"Knowledge about the world" for our example:

- Every camera is a digital device
- Every netbook is a digital device
- Every computer is a digital device
- Every book is human-readable

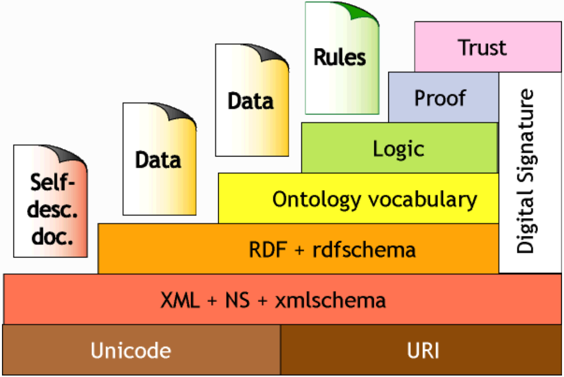
Such knowledge is set-based (also called class-based)

- The set of cameras is a subset of the set of digital devices
- The set of netbooks is a subset of the set of digital devices
- The set of books is a subset of the set of human-readable objects

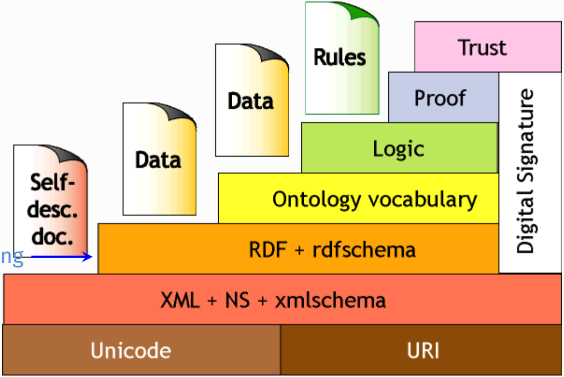
Ontologies

Ontologies provide formal specifications of the classes of objects that inhabit "the world", the relationships between individual and classes, and their properties.

KNOWLEDGE REPRESENTATION ON THE SEMANTIC WEB (2005 VISION)

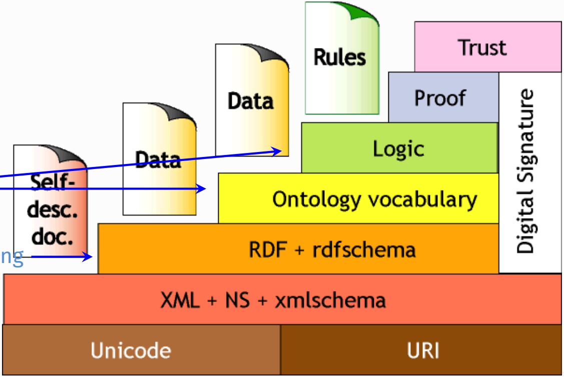


KNOWLEDGE REPRESENTATION ON THE SEMANTIC WEB (2005 VISION)



data exchange/simple reasoning
(previous lecture)

KNOWLEDGE REPRESENTATION ON THE SEMANTIC WEB (2005 VISION)



advanced reasoning
(this lecture)

data exchange/simple reasoning
(previous lecture)

RECALL FROM LAST LECTURE

- RDF = data model: make assertions about resources using triples
- RDF Schema is a standard vocabulary for expressing simple ontologies
  1. Classes, Properties
  2. type, subClassOf, subPropertyOf
  3. range, domain
  4. a number of axiomatic triples describing meta-information about RDFS.

Example

```
ex:vegetableThaiCurry    ex:thaiDishBasedOn    ex:coconutMilk                     .
ex:sebastian             rdf:type              ex:AllergicToNuts                  .
ex:sebastian             ex:eats               ex:vegetableThaiCurry              .

ex:AllergicToNuts        rdfs:subClassOf       ex:Pitiable                        .
ex:thaiDishBasedOn       rdfs:domain           ex:Thai                            .
ex:thaiDishBasedOn       rdfs:range            ex:Nutty                           .
ex:thaiDishBasedOn       rdfs:subPropertyOf    ex:hasIngredient                   .
ex:hasIngredient         rdf:type              rdfs:containerMembershipProperty   .
```

SOME STRANGE THINGS IN RDF SCHEMA

The RDFS meta model has some strange axioms.

- `rdfs:Resource` is the superclass of everything. But, it is itself an instance of its subclass `rdfs:Class`.
- `rdfs:Class` is an instance of itself

SOME STRANGE THINGS IN RDF SCHEMA

The RDFS meta model has some strange axioms.

- `rdfs:Resource` is the superclass of everything. But, it is itself an instance of its subclass `rdfs:Class`.
- `rdfs:Class` is an instance of itself

It is known from logic research that allowing classes to be themselves classes (known as non-wellfoundedness) causes problems when you add more expressive features.

LIMITATIONS OF RDF SCHEMA

RDF Schema allows us to represent some ontological knowledge:

- Typed hierarchies using classes and subclasses, properties and subproperties
- Domain and range restrictions
- Describing instances of classes (through subclasses and `rdf:type`)

Sometimes we want more:

- Local scope of properties Using `rdfs:range` and `rdfs:domain` we can't state that cows only eat plants while other animals may eat meat too.
- Disjointness of classes. We can't state, for example, that `terms:male` and `terms:female` do not have any members in common.
- Special characteristics of properties. Sometimes it is convenient to be able to say that a property is transitive (like "greater than"), unique (like "father of"), or the inverse of another property (like "father of" and "child of").
- Cardinality restrictions like "a person has exactly 2 parents"

OWL: ONTOLOGY WEB LANGUAGE

The Ontology Web Language (OWL) allows us to talk about such things (among others)

OWL: ONTOLOGY WEB LANGUAGE

The Ontology Web Language (OWL) allows us to talk about such things (among others)

There is always a trade-off between expressiveness and efficient reasoning support:

- The more expressive a language …
- The more inefficient the inferincing becomes …
- …it may even become undecidable!

Part II: Web Ontology Language—OWL

OWL: WEB ONTOLOGY LANGUAGE

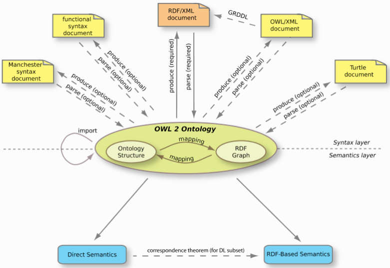
OWL = a Vocabulary like RDF Schema.

- OWL extends the RDFS vocabulary, and adds axioms to express more complex relations between classes (like disjointness, cardinality restrictions, ...) and properties (datatype ranges, functional properties, etc).
- It uses the same data model as RDF schema (namely: RDF)

OWL versions

- OWL version 1.0 was standardized as a recommendation in 2004.
- OWL version 2.0 (second edition) proposes a backwards-compatible update to OWL 1.0. It features several extensions to OWL version 1.0
- We will focus mostly on the OWL 1.0 features in this lecture.

OWL: WEB ONTOLOGY LANGUAGE



OWL has a number of syntaxes.

- Every OWL-compliant tool **must** support the RDF/XML based syntax; others are optional (but sometimes more readable).
- As such, OWL ontologies are usually written in RDF/XML.
- Therefore, the book/handouts also use RDF/XML.
- RDF/XML is very verbose, however, and we will therefore use a Turtle syntax in these slides. (This is of course equivalent.)

THINGS TO REMEMBER ABOUT TURTLE

Turtle has some convenient abbreviations:

- Blank nodes can be described by nesting Turtle statements in [ ]
- Collections can be described by resources between parenthesis (…)

Example:

```
@prefix staff:  <http://www.example.org/staff id/> .
@prefix :  <http://www.example.org/terms/> .

staff:85740      :address        _:addr
_:addr           :city           ”Bedford”^^xsd:string      ;
                 :street         ”1501 Grant Avenue”        ;
                 :state          ”Massachusetts”            ;
                 :postalcode     ”0713”                     .
staff:85740      a               :employee                  .
```

THINGS TO REMEMBER ABOUT TURTLE

Turtle has some convenient abbreviations:

- Blank nodes can be described by nesting Turtle statements in **[ ]**
- Collections can be described by resources between parenthesis **(...)**

Example:

```
@prefix staff: <http://www.example.org/staff id/> .
@prefix : <http://www.example.org/terms/> .

staff:85740    :address    [ :city          ”Bedford”^^xsd:string ;
                             :street        ”1501 Grant Avenue” ;
                             :state         ”Massachusetts” ;
                             :postalcode    ”0713”                  ] .
staff:85740    a           :employee                                  .
```

THINGS TO REMEMBER ABOUT TURTLE

Turtle has some convenient abbreviations:

- Blank nodes can be described by nesting Turtle statements in `[ ]`
- Collections can be described by resources between parenthesis `(...)`

Example:

```
@prefix courses: <http://ulb.be/courses/> .
@prefix terms: <http://ulb.be/terms/> .
@prefix : <http://ulb.be/students/> .

courses:509     terms:students     _:a          .
_:a             rdf:first          :amy         .
_:a             rdf:rest           _:b          .
_:b             rdf:first          :mohamed     .
_:b             rdf:rest           _:c          .
_:b             rdf:first          :john        .
_:b             rdf:rest           rdf:nil      .
```

THINGS TO REMEMBER ABOUT TURTLE

Turtle has some convenient abbreviations:

- Blank nodes can be described by nesting Turtle statements in `[ ]`
- Collections can be described by resources between parenthesis `(...)`

Example:

```
@prefix courses: <http://ulb.be/courses/> .
@prefix terms: <http://ulb.be/terms/> .
@prefix : <http://ulb.be/students/> .

courses:509    terms:students    ( :amy :mohamed :john ) .
```

OWL SYNTACTIC STRUCTURE

- An OWL document (in Turtle, or in RDF/XML) typically starts with declaring namespaces for the `rdf`, `rdfs`, and `owl` prefixes.
- The default namespace is often re-defined to hold the terms of the vocabulary that is being described by the OWL document.

OWL document header:

```
@prefix :     <http://www.example.org>
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
@prefix owl:  <http://www.w3.org/2002/07/owl#>
```

OWL declarations

CLASSES, ROLES, AND INDIVIDUALS

- As in RDF and RDF Schema, the basic building blocks of OWL are classes, properties, and individuals.
- In OWL, properties are also called roles or slots.

Example

CLASSES, ROLES, AND INDIVIDUALS

- As in RDF and RDF Schema, the basic building blocks of OWL are classes, properties, and individuals.
- In OWL, properties are also called roles or slots.
- OWL has its own term to declare classes: `owl:Class` (which is distinct from `rdfs:Class`)

Example

```
:Professor    rdf:type owl:Class .
:Person       rdf:type owl:Class .
:Organization rdf:type owl:Class .
```

CLASSES, ROLES, AND INDIVIDUALS

- As in RDF and RDF Schema, the basic building blocks of OWL are classes, properties, and individuals.
- In OWL, properties are also called roles or slots.
- OWL has its own term to declare classes: `owl:Class` (which is distinct from `rdfs:Class`)
- Individuals are declared with `rdf:type`, as in RDF.

Example

```
:Professor    rdf:type owl:Class .
:Person       rdf:type owl:Class .
:Organization rdf:type owl:Class .

:John         rdf:type owl:Professor .
```

CLASSES, ROLES, AND INDIVIDUALS

- As in RDF and RDF Schema, the basic building blocks of OWL are classes, properties, and individuals.
- In OWL, properties are also called roles or slots.
- OWL has its own term to declare classes: `owl:Class` (which is distinct from `rdfs:Class`)
- Individuals are declared with `rdf:type`, as in RDF.
- OWL supports distinct kinds of properties.
- `owl:ObjectProperty` defines abstract properties (abstract roles), that connect individuals with individuals.
- `owl:DataTypeProperty` defines concrete properties (concrete roles), that connect individuals with data values (i.e., with elements of datatypes).

Example

```
:Professor    rdf:type owl:Class .
:Person       rdf:type owl:Class .
:Organization rdf:type owl:Class .

:John         rdf:type owl:Professor .

:affiliation  rdf:type owl:ObjectProperty ;



:firstName    rdf:type owl:DataTypeProperty;
```

CLASSES, ROLES, AND INDIVIDUALS

- As in RDF and RDF Schema, the basic building blocks of OWL are classes, properties, and individuals.
- In OWL, properties are also called roles or slots.
- OWL has its own term to declare classes: `owl:Class` (which is distinct from `rdfs:Class`)
- Individuals are declared with `rdf:type`, as in RDF.
- OWL supports distinct kinds of properties.
- `owl:ObjectProperty` defines abstract properties (abstract roles), that connect individuals with individuals.
- `owl:DataTypeProperty` defines concrete properties (concrete roles), that connect individuals with data values (i.e., with elements of datatypes).
- `rdf:type`, `rdfs:domain`, `rdfs:range`, `rdfs:subClassOf`, `rdfs:subPropertyOf` are used as before

Example

```
:Professor    rdf:type owl:Class .
:Person       rdf:type owl:Class .
:Organization rdf:type owl:Class .

:John         rdf:type owl:Professor .

:affiliation  rdf:type owl:ObjectProperty ;
              rdfs:domain :Person ;
              rdfs:range :Organization .

:firstName    rdf:type owl:DataTypeProperty;
              rdfs:domain :Person;
              rdfs:range xsd:string.

:rudi         :affiliation :aifb, :ontoprise;
              :firstName ”Rudi”^^xsd:string .
```

CLASSES, ROLES, AND INDIVIDUALS

- As in RDF and RDF Schema, the basic building blocks of OWL are classes, properties, and individuals.
- In OWL, properties are also called roles or slots.
- OWL has its own term to declare classes: `owl:Class` (which is distinct from `rdfs:Class`)
- Individuals are declared with `rdf:type`, as in RDF.
- OWL supports distinct kinds of properties.
- `owl:ObjectProperty` defines abstract properties (abstract roles), that connect individuals with individuals.
- `owl:DataTypeProperty` defines concrete properties (concrete roles), that connect individuals with data values (i.e., with elements of datatypes).
- `rdf:type`, `rdfs:domain`, `rdfs:range`, `rdfs:subClassOf`, `rdfs:subPropertyOf` are used as before

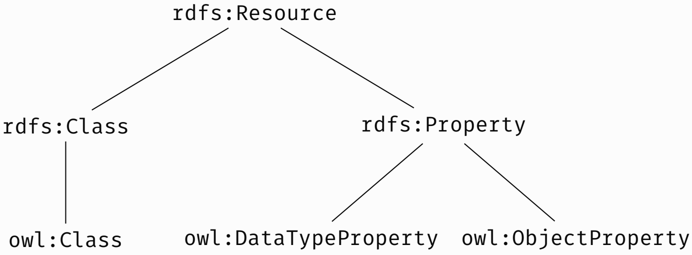
Example

```
:Professor    rdf:type owl:Class .
:Person       rdf:type owl:Class .
:Organization rdf:type owl:Class .

:John         rdf:type owl:Professor .

:affiliation  rdf:type owl:ObjectProperty ;
              rdfs:domain :Person ;
              rdfs:range :Organization .

:firstName    rdf:type owl:DataTypeProperty;
              rdfs:domain :Person;
              rdfs:range xsd:string.

:rudi         :affiliation :aifb, :ontoprise;
              :firstName ”Rudi”^^xsd:string .
```

Valid deductions:

```
:rudi         rdf:type :Person
:aifb         rdf:type :Organization
:ontoprise    rdf:type :Organization
```

CLASSES AND ROLES: RELATIONSHIP WITH RDFS



`rdfs:Resource`

`rdfs:Class`

`rdfs:Property`

`owl:Class`

`owl:DataTypeProperty`

`owl:ObjectProperty`

CONCRETE PROPERTIES: SUPPORTED DATATYPES

- The range of `owl:DataTypeProperty` can in principle refer to any of the XML Schema built-in simple types.
- Tools are not required to support all of these datatypes, however (and typically support only a few).

Type	sample values
`xsd:string`	any Unicode string
`xsd:boolean`	true, false, 1, 0
`xsd:decimal`	3.1415
`xsd:float`	6.02214199E23
`xsd:double`	42E970
`xsd:dateTime`	2004-09-26T16:29:00-05:00
`xsd:time`	16:29:00-05:00
`xsd:date`	2004-09-26
`xsd:hexBinary`	48656c6c6f0a
`xsd:base64Binary`	SGVsbG8K
`xsd:anyURI`	http://www.brics.dk/ixwt/
`xsd:QName`	rcp:recipe, recipe
...	

NOTATION

In what follows:

- We range over arbitrary URIs by P, R and S (i.e., anything admissible for the predicate position of a triple)
- u, v, w, C, D, and E refer to arbitrary URIs or blank node IDs by (i.e., anything admissible for the subject position of a triple)
- x and y can be used for arbitrary URIs, blank node IDs or literals

PROPERTY CHARACTERISTICS: INVERSES

- *P* `owl:inverseOf` *R* is used to specify that property *P* is the inverse of property *R* (and vice versa)

Deduction Rule

If	*P* `owl:inverseOf` *R* **.**
And	*u P v* **.**
Then add	*v R u* **.**

If	*P* `owl:inverseOf` *R* **.**
And	*u R v* **.**
Then add	*v P u* **.**

Example

```
:fatherOf     owl:inverseOf     :childOf .
:Jake         :fatherOf         :John    .
```

PROPERTY CHARACTERISTICS: INVERSES

- *P* `owl:inverseOf` *R* is used to specify that property *P* is the inverse of property *R* (and vice versa)

Deduction Rule

If	*P* `owl:inverseOf` *R* **.**
And	*u P v* **.**
Then add	*v R u* **.**
If	*P* `owl:inverseOf` *R* **.**
And	*u R v* **.**
Then add	*v P u* **.**

Example

:fatherOf	owl:inverseOf	:childOf	.
:Jake	:fatherOf	:John	.
:John	:childOf	:Jake	.

PROPERTY CHARACTERISTICS: SYMMETRY

- P `rdf:type` `owl:SymmetricProperty` is used to specify that property P is a symmetric property

Deduction Rule

If	P `rdf:type` `owl:SymmetricProperty .`
And	$u\ P\ v$ `.`
Then add	$v\ P\ u$ `.`

Example

```
:marriedTo    rdf:type      owl:SymmetricProperty .
:Jake         :marriedTo    :Eve                  .
```

PROPERTY CHARACTERISTICS: SYMMETRY

- *P* `rdf:type` `owl:SymmetricProperty` is used to specify that property *P* is a symmetric property

Deduction Rule

If	*P* `rdf:type owl:SymmetricProperty .`
And	*u P v* `.`
Then add	*v P u* `.`

Example

:marriedTo	rdf:type	owl:SymmetricProperty	.
:Jake	:marriedTo	:Eve	.
:Eve	:marriedTo	:Jake	.

PROPERTY CHARACTERISTICS: TRANSITIVITY

- *P* `rdf:type` `owl:TransitiveProperty` is used to specify that property *P* is a Transitive property

Deduction Rule

If *P* `rdf:type` `owl:TransitiveProperty` .
And *u P v* .
And *v P w* .
Then add *u P w* .

Example

```
:ancestor   rdf:type    owl:TransitiveProperty .
:Jake       :ancestor   :John                  .
:Jill       :ancestor   :Jake                  .
```

PROPERTY CHARACTERISTICS: TRANSITIVITY

- *P* `rdf:type` `owl:TransitiveProperty` is used to specify that property *P* is a Transitive property

Deduction Rule

If	*P* `rdf:type` `owl:TransitiveProperty` .
And	*u P v* .
And	*v P w* .
Then add	*u P w* .

Example

:ancestor	rdf:type	owl:TransitiveProperty	.
:Jake	:ancestor	:John	.
:Jill	:ancestor	:Jake	.
:Jill	:ancestor	:John	.

ASSERTING EQUIVALENCE OF CLASSES

- It is always possible that we have used a specific URI to identify a particular concept while someone else has used a different URI for the same concept
- *v* `owl:equivalentClass` *w* is used to specify that every member of class *v* is a member of class *w*, and vice versa

Semantics is given by

```
owl:equivalentClass rdf:type owl:SymmetricProperty .
owl:equivalentClass rdf:type owl:TransitiveProperty .
owl:equivalentClass rdfs:subPropertyOf rdfs:subClassOf .
```

Example

```
:Man    owl:equivalentClass    :Homme .
:Jake   rdf:type               :Man   .
```

ASSERTING EQUIVALENCE OF CLASSES

- It is always possible that we have used a specific URI to identify a particular concept while someone else has used a different URI for the same concept
- *v* `owl:equivalentClass` *w* is used to specify that every member of class *v* is a member of class *w*, and vice versa

Semantics is given by

```
owl:equivalentClass rdf:type owl:SymmetricProperty .
owl:equivalentClass rdf:type owl:TransitiveProperty .
owl:equivalentClass rdfs:subPropertyOf rdfs:subClassOf .
```

Example

```
:Man      owl:equivalentClass   :Homme .
:Jake     rdf:type              :Man   .
:Homme    owl:equivalentclass   :Man   .
:Man      rdfs:subClassOf       :Homme .
:Homme    rdfs:subClassOf       :Man   .
:Jake     rdf:type              :Homme .
```

ASSERTING EQUIVALENCE OF PROPERTIES

- It is always possible that we have used a specific URI to identify a particular concept while someone else has used a different URI for the same concept
- *P* `owl:equivalentProperty` *R* is used to specify that properties *P* and *R* are equivalent

Semantics is given by

```
owl:equivalentProperty rdf:type owl:SymmetricProperty .
owl:equivalentProperty rdf:type owl:TransitiveProperty .
owl:equivalentProperty rdfs:subPropertyOf rdfs:subPropertyOf .
```

```
:fatherOf    owl:equivalentProperty    :père .
:Jake        :père                     :John .
```

ASSERTING EQUIVALENCE OF PROPERTIES

- It is always possible that we have used a specific URI to identify a particular concept while someone else has used a different URI for the same concept
- *P* `owl:equivalentProperty` *R* is used to specify that properties *P* and *R* are equivalent

Semantics is given by

```
owl:equivalentProperty rdf:type owl:SymmetricProperty .
owl:equivalentProperty rdf:type owl:TransitiveProperty .
owl:equivalentProperty rdfs:subPropertyOf rdfs:subPropertyOf .
```

```
:fatherOf   owl:equivalentProperty   :père     .
:Jake       :père                    :John     .
:père       owl:equivalentProperty   :fatherOf .
:père       rdfs:subPropertyOf       :fatherOf .
:fatherOf   rdfs:subPropertyOf       :père     .
:Jake       :fatherOf                :John     .
```

ASSERTING EQUIVALENCE OF INDIVIDUALS

- It is always possible that we have used a specific URI to identify a particular concept while someone else has used a different URI for the same concept
- *v* `owl:sameAs` *w* is used to specify that *v* and *w* are the same individuals

Semantics is given by

`owl:sameAs rdf:type owl:SymmetricProperty .`

If *u* `owl:sameAs` *v* .
And *u P x* .
Then add *v P x* .

If *u* `owl:sameAs` *v* .
And *w P u* .
Then add *w P v* .

MORE PROPERTY CHARACTERISTICS: FUNCTIONALITY

- *P* `rdf:type` `owl:FunctionalProperty` is used to specify that *P* can only take one object for a particular subject

Inference Rule:

If *P* `rdf:type` `owl:FunctionalProperty .`
And *u P v* **.**
And *u P w* **.**
Then add *v* `owl:sameAs` *w* **.**

Example

:hasFather	rdf:type	owl:FunctionalProperty	.
:John	:hasFather	:Jake	.
:John	:hasFather	ex:Jake-J	.

MORE PROPERTY CHARACTERISTICS: FUNCTIONALITY

- *P* `rdf:type owl:FunctionalProperty` is used to specify that *P* can only take one object for a particular subject

Inference Rule:

If *P* `rdf:type owl:FunctionalProperty .`
And *u P v* `.`
And *u P w* `.`
Then add *v* `owl:sameAs` *w* `.`

Example

:hasFather	rdf:type	owl:FunctionalProperty	.
:John	:hasFather	:Jake	.
:John	:hasFather	ex:Jake-J	.
:Jake	owl:sameAs	ex:Jake-J	.

MORE PROPERTY CHARACTERISTICS: INVERSE FUNCTIONALITY

- *P* `rdf:type` `owl:InverseFunctionalProperty` is used to specify that *P* can only take one subject for a particular object

Inference Rule:

If *P* `rdf:type` `owl:InverseFunctionalProperty .`
And *v* *P* *u* `.`
And *w* *P* *u* `.`
Then add *v* `owl:sameAs` *w* `.`

BOOLEAN CLASS CONSTRUCTORS: INTERSECTION (1/2)

- C `owl:intersectionOf` `(`$D_1 \ldots D_k$`)` is used to indicate that u is an instance of class C if, and only if, it is simultaneously an instance of all classes $D_1, \ldots, D_k$

Inference Rule:

If C `owl:intersectionOf (`$D_1 \ldots D_k$`) .`
And u `rdf:type` C `.`
Then add u `rdf:type` D_i `.` (for every $1 \leq i \leq k$)

If C `owl:intersectionOf (`$D_1 \ldots D_k$`) .`
And u `rdf:type` D_i `.` (for every $1 \leq i \leq k$)
Then add u `rdf:type` C `.`

Example

```
:MaleProfessor    rdf:type              owl:Class               ;
                  owl:intersectionOf    (:Professor :Male)      .
:John             rdf:type              :MaleProfessor          .
```

BOOLEAN CLASS CONSTRUCTORS: INTERSECTION (1/2)

- C `owl:intersectionOf` ($D_1 \ldots D_k$) is used to indicate that u is an instance of class C if, and only if, it is simultaneously an instance of all classes $D_1, \ldots, D_k$

Inference Rule:

If C `owl:intersectionOf` ($D_1 \ldots D_k$) .
And u `rdf:type` C .
Then add u `rdf:type` D_i . (for every $1 \leq i \leq k$)

If C `owl:intersectionOf` ($D_1 \ldots D_k$) .
And u `rdf:type` D_i . (for every $1 \leq i \leq k$)
Then add u `rdf:type` C .

Example

```
:MaleProfessor   rdf:type             owl:Class              ;
                 owl:intersectionOf   (:Professor :Male)     .
:John            rdf:type             :MaleProfessor         .
:John            rdf:type             :Professor             ;
                 rdf:type             :Male                  .
```

BOOLEAN CLASS CONSTRUCTORS: INTERSECTION (1/2)

- C `owl:intersectionOf` `(`$D_1 \ldots D_k$`)` is used to indicate that u is an instance of class C if, and only if, it is simultaneously an instance of all classes $D_1, \ldots, D_k$

Inference Rule:

If C `owl:intersectionOf` `(`$D_1 \ldots D_k$`)` `.`
And u `rdf:type` C `.`
Then add u `rdf:type` D_i `.` (for every $1 \leq i \leq k$)

If C `owl:intersectionOf` `(`$D_1 \ldots D_k$`)` `.`
And u `rdf:type` D_i `.` (for every $1 \leq i \leq k$)
Then add u `rdf:type` C `.`

Another Example

```
:MaleProfessor   rdf:type             owl:Class              ;
                 owl:intersectionOf   (:Professor :Male)     .
:John            rdf:type             :Male                  ;
                 rdf:type             :Professor             .
```

BOOLEAN CLASS CONSTRUCTORS: INTERSECTION (1/2)

- C `owl:intersectionOf (`$D_1 \ldots D_k$`)` is used to indicate that u is an instance of class C if, and only if, it is simultaneously an instance of all classes $D_1, \ldots, D_k$

Inference Rule:

If C `owl:intersectionOf (`$D_1 \ldots D_k$`) .`
And u `rdf:type` C `.`
Then add u `rdf:type` D_i `.` (for every $1 \leq i \leq k$)

If C `owl:intersectionOf (`$D_1 \ldots D_k$`) .`
And u `rdf:type` D_i `.` (for every $1 \leq i \leq k$)
Then add u `rdf:type` C `.`

Another Example

```
:MaleProfessor    rdf:type             owl:Class             ;
                  owl:intersectionOf   (:Professor :Male)    .
:John             rdf:type             :Male                 ;
                  rdf:type             :Professor            .
:John             rdf:type             :MaleProfessor        .
```

BOOLEAN CLASS CONSTRUCTORS: INTERSECTION (2/2)

- C `owl:intersectionOf` ($D_1 \ldots D_k$) is used to indicate that u is an instance of C if, and only if, it is simultaneously an instance of all classes $D_1, \ldots, D_k$
- `owl:intersectionOf` is often used together with blank nodes and `rdfs:subClassOf` to make this an “if” instead of an “if and only if”

Example

```
:MaleProfessor   rdf:type               owl:Class              ;
                 rdfs:subClassOf
                   [ owl:intersectionOf  (:Person :Male) ]     .

:John            rdf:type               :Person                ;
                 rdf:type               :Male                  .
```

All `:MaleProfessor` are `:Male` and `:Person`. Not all `:Male` `:Persons` are :`:MaleProfessor`, however. Hence, we cannot infer `:John rdf:type :MaleProfessor`

BOOLEAN CLASS CONSTRUCTORS: COMPLEMENTOF

- *C* `owl:complementOf` *D* is used to indicate that *u* is an instance of *C* if, and only if, it is not an instance of *D*.

Semantics given by:

`owl:complementOf rdf:type owl:SymmetricProperty .`

If *C* `owl:complementOf` *D* `.`
And *u* `rdf:type` *C* `.`
Then it **cannot hold that** *u* `rdf:type` *D* `.`

Be careful with complementOf! Example.

```
:Male     owl:complementOf :Female  .
:tweety   rdf:type         :Penguin  .
```

BOOLEAN CLASS CONSTRUCTORS: COMPLEMENTOF

- *C* `owl:complementOf` *D* is used to indicate that *u* is an instance of *C* if, and only if, it is not an instance of *D*.

Semantics given by:

`owl:complementOf rdf:type owl:SymmetricProperty .`

If *C* `owl:complementOf` *D* .
And *u* `rdf:type` *C* .
Then it **cannot hold that** *u* `rdf:type` *D* .

Be careful with complementOf! Example.

```
:Male     owl:complementOf :Female  .
:tweety   rdf:type         :Penguin  .
```

Here, we cannot conclude that `:tweety` is Male, nor that it is Female

BOOLEAN CLASS CONSTRUCTORS: COMPLEMENTOF

- *C* `owl:complementOf` *D* is used to indicate that *u* is an instance of *C* if, and only if, it is not an instance of *D*.

Semantics given by:

`owl:complementOf rdf:type owl:SymmetricProperty .`

If *C* `owl:complementOf` *D* `.`
And *u* `rdf:type` *C* `.`
Then it **cannot hold that** *u* `rdf:type` *D* `.`

Be careful with complementOf! Example.

```
:Male     owl:complementOf :Female  .
:tweety   rdf:type         :Penguin  .
```

BOOLEAN CLASS CONSTRUCTORS: COMPLEMENTOF

- C `owl:complementOf` D is used to indicate that u is an instance of C if, and only if, it is not an instance of D.

Semantics given by:

`owl:complementOf rdf:type owl:SymmetricProperty .`

If C `owl:complementOf` D `.`
And u `rdf:type` C `.`
Then it **cannot hold that** u `rdf:type` D `.`

Be careful with complementOf! Example.

```
:Male      owl:complementOf :Female .
:tweety    rdf:type         :Penguin .
:Furniturerdfs:subClassOf   [ owl:complementOf :Female ] .
:desk      rdf:type         :Furniture .
```

BOOLEAN CLASS CONSTRUCTORS: COMPLEMENTOF

- *C* `owl:complementOf` *D* is used to indicate that *u* is an instance of *C* if, and only if, it is not an instance of *D*.

Semantics given by:

`owl:complementOf rdf:type owl:SymmetricProperty .`

If *C* `owl:complementOf` *D* `.`
And *u* `rdf:type` *C* `.`
Then it **cannot hold that** *u* `rdf:type` *D* `.`

Be careful with complementOf! Example.

```
:Male       owl:complementOf :Female .
:tweety     rdf:type         :Penguin .
:Furniturerdfs:subClassOf    [ owl:complementOf :Female ] .
:desk       rdf:type         :Furniture .
:desk       rdf:type         [ owl:complementOf :Female ].
```

BOOLEAN CLASS CONSTRUCTORS: COMPLEMENTOF

- C `owl:complementOf` D is used to indicate that u is an instance of C if, and only if, it is not an instance of D.

Semantics given by:

`owl:complementOf rdf:type owl:SymmetricProperty .`

If C `owl:complementOf` D `.`
And u `rdf:type` C `.`
Then it **cannot hold that** u `rdf:type` D `.`

Be careful with complementOf! Example.

```
:Male       owl:complementOf :Female .
:tweety     rdf:type         :Penguin .
:Furniturerdfs:subClassOf    [ owl:complementOf :Female ] .
:desk       rdf:type         :Furniture .
:desk       rdf:type         [ owl:complementOf :Female ].
:desk       rdf:type         :Male .
```

CLASS DISJOINTNESS

- *C* `owl:disjointWith` *D* is used to indicate that no instance of *C* is an instance of *D*, and vice versa.

It is an abbreviation of:

C `rdfs:subClassOf [ owl:complementOf` *D* `]`
D `rdfs:subClassOf [ owl:complementOf` *C* `]`

BOOLEAN CLASS CONSTRUCTORS: UNIONOF

- C `owl:unionOf` $(D_1 \ldots D_k)$ is used to indicate that u is an instance of C if, and only if, it is an instance of at least one the classes $D_1, \ldots, D_k$.

Semantics given by:

If C `owl:unionOf (`$D_1 \ldots D_k$`) .`
And u `rdf:type` D_i `.` (for some $1 \leq i \leq k$)
Then add u `rdf:type` C `.`

If C `owl:unionOf (`$D_1 \ldots D_k$`) .`
And u `rdf:type [owl:complementOf` D_j`] .` for every $j \neq i$
Then add u `rdf:type` D_i `.`

Example

```
:Person    owl:unionOf    (:Male :Female)              .
:john      rdf:type       :Person                      .
:john      rdf:type       [owl:complementOf :Female] .
```

BOOLEAN CLASS CONSTRUCTORS: UNIONOF

- C `owl:unionOf` $(D_1 \ldots D_k)$ is used to indicate that u is an instance of C if, and only if, it is an instance of at least one the classes $D_1, \ldots, D_k$.

Semantics given by:

If C `owl:unionOf` $(D_1 \ldots D_k)$.
And u `rdf:type` D_i . (for some $1 \leq i \leq k$)
Then add u `rdf:type` C .

If C `owl:unionOf` $(D_1 \ldots D_k)$.
And u `rdf:type` `[owl:complementOf` D_j`]` . for every $j \neq i$
Then add u `rdf:type` D_i .

Example

:Person	owl:unionOf	(:Male :Female)	.
:john	rdf:type	:Person	.
:john	rdf:type	[owl:complementOf :Female]	.
:john	`rdf:type`	:Male	.

CLOSED CLASSES: ONE OF

- C `owl:oneOf` $(v_1 \ldots v_k)$ is used to indicate that the only individuals of class C are $v_1, \ldots, v_k$.

OWL BUILTIN CLASSES

- There are two predefined classes in OWL: `owl:Thing` and `owl:Nothing`
- `owl:Thing` is the most general class, it has every individual as an instance
- `owl:Nothing` is the empty class, it does not have any instances

ASSERTING CLASSES DISJOINT AND INDIVIDUALS DISTINCT

- Unlike most other knowledge representation languages, OWL does not assume the Unique Name Assumption (UNA): distinct resources need not represent distinct things.
- As already seen, equivalence between classes can be specified by `owl:equivalentClass`; equivalence between properties by `owl:equivalentProperty`; and between individuals by `owl:sameAs`.
- Individuals can be declared distinct by means of `owl:differentFrom`
- Classes can be declared disjoint by means of `owl:disjointWith`.

PROPERTY RESTRICTIONS: INTRODUCTION

- By means of `rdfs:domain` and `rdfs:range` we can only specify that the domain and range should hold globally for a property
- OWL allows us to make local restriction on properties (e.g. cows eat plants while other animals may also eat meat) by means of so-called property restrictions
- OWL distinguishes between the following two:
  - Value constraints (`owl:someValuesFrom`, `owl:allValuesFrom`, `owl:hasValue`)
  - Cardinality constraints (`owl:minCardinality`, `owl:maxCardinality`)

Property restrictions define (anonymous) classes and have the general syntax:

```
_:1  rdf:type        owl:Restriction;
     owl:onProperty  P;
     value constr    D .
```

```
_:1  rdf:type        owl:Restriction;
     owl:onProperty  P;
     cardinal constr ”i”^^xsd:nonNegativeInteger .
```

PROPERTY RESTRICTIONS: ALLVALUESFROM

- `[rdf:type owl:Restriction; owl:onProperty` *P*`; owl:allValuesFrom` *D*`]` denotes the class consisting of all individuals *u* for which the range of property *P* is class *D*.

Inference Rule:

If *u* `rdf:type[rdf:type owl:Restriction;`
`owl:onProperty` *P*`;`
`owl:allValuesFrom` *D*`] .`
And *u P v* `.`
Then add *v* `rdf:type` *D*

If for all *v* s.t. *u P v* `.`
it holds that *v* `rdf:type` *D* `.`
Then add *u* `rdf:type[rdf:type owl:Restriction;`
`owl:onProperty` *P*`;`
`owl:allValuesFrom` *D*`] .`

PROPERTY RESTRICTIONS: ALLVALUESFROM

- `[rdf:type owl:Restriction; owl:onProperty` *P*`; owl:allValuesFrom` *D*`]` denotes the class consisting of all individuals *u* for which the range of property *P* is class *D*.

Inference Rule:

If	*u* `rdf:type [rdf:type owl:Restriction;`
	`owl:onProperty` *P*`;`
	`owl:allValuesFrom` *D*`] .`

And	*u P v* `.`
Then add	*v* `rdf:type` *D*
If for all *v* s.t.	*u P v* `.`
it holds that	*v* `rdf:type` *D* `.`

Then add	*u* `rdf:type [rdf:type owl:Restriction;`
	`owl:onProperty` *P*`;`
	`owl:allValuesFrom` *D*`] .`

Example:

```
:OnlyDaughters  rdf:type         owl:Class ;
                rdfs:subclassOf  [rdf:type owl:Restriction;
                                  owl:onProperty :hasChild;
                                  owl:allValuesFrom :Female] .
:john           rdf:type         :OnlyDaughters ;
                :hasChild        :mary .
:jane           :hasChild        :john .
```

PROPERTY RESTRICTIONS: ALLVALUESFROM

- `[rdf:type owl:Restriction; owl:onProperty` *P*`; owl:allValuesFrom` *D*`]` denotes the class consisting of all individuals *u* for which the range of property *P* is class *D*.

Inference Rule:

If	*u* `rdf:type[rdf:type owl:Restriction;`
	`owl:onProperty` *P*`;`
	`owl:allValuesFrom` *D*`] .`

And	*u P v* `.`
Then add	*v* `rdf:type` *D*
If for all *v* s.t.	*u P v* `.`
it holds that	*v* `rdf:type` *D* `.`

Then add	*u* `rdf:type[rdf:type owl:Restriction;`
	`owl:onProperty` *P*`;`
	`owl:allValuesFrom` *D*`] .`

Example:

```
:OnlyDaughters  rdf:type         owl:Class ;
                rdfs:subclassOf  [rdf:type owl:Restriction;
                                   owl:onProperty :hasChild;
                                   owl:allValuesFrom :Female] .
:john           rdf:type         :OnlyDaughters ;
                :hasChild        :mary .
:jane           :hasChild        :john .
:mary           rdf:type         :Female .
```

PROPERTY RESTRICTIONS: SOMEVALUESFROM

- `[rdf:type owl:Restriction; owl:onProperty` *P*`; owl:someValuesFrom` *D*`]` denotes the class consisting of all individuals *u* that have at least one occurrence of property *P* whose range is class *D*.

Inference Rule:

```
If          u P v .
And         v rdf:type D .
Then add    u rdf:type[rdf:type owl:Restriction;
                       owl:onProperty P;
                       owl:someValuesFrom D] .

If          u rdf:type[rdf:type owl:Restriction;
                       owl:onProperty P;
                       owl:someValuesFrom D] .
There has to exist some  u P v .
with                     v rdf:type D
```

PROPERTY RESTRICTIONS: SOMEVALUESFROM

- `[rdf:type owl:Restriction; owl:onProperty` *P*`; owl:someValuesFrom` *D*`]` denotes the class consisting of all individuals *u* that have at least one occurrence of property *P* whose range is class *D*.

Example: a class for all Mothers

```
:Mother     rdf:type            owl:Class ;
            owl:intersectionOf  (  :Person :Female
                                   [rdf:type owl:Restriction;
                                    owl:onProperty :hasChild;
                                    owl:someValuesFrom :Person]
                                ) .
:jane       rdf:type            :Female, :Person ;
            :hasChild           :john .
:john       rdf:type            :Person .
```

PROPERTY RESTRICTIONS: SOMEVALUESFROM

- `[rdf:type owl:Restriction; owl:onProperty` *P*`; owl:someValuesFrom` *D*`]` denotes the class consisting of all individuals *u* that have at least one occurrence of property *P* whose range is class *D*.

Example: a class for all Mothers

```
:Mother     rdf:type            owl:Class ;
            owl:intersectionOf  (  :Person :Female
                                   [rdf:type owl:Restriction;
                                    owl:onProperty :hasChild;
                                    owl:someValuesFrom :Person]
                                ) .
:jane       rdf:type            :Female, :Person ;
            :hasChild           :john .
:john       rdf:type            :Person .
:jane       rdf:type            :Mother .
```

PROPERTY RESTRICTIONS: HASVALUE

- `[rdf:type owl:Restriction; owl:onProperty` *P*`; owl:hasValue` *v*`]` is a particular form of `owl:someValuesFrom`. It denotes the class consisting of all individuals *u* for which *u P v* holds.

Inference Rule:

```
If          u P v .
Then add    u rdf:type[rdf:type owl:Restriction;
                      owl:onProperty P;
                      owl:hasValue v] .

If          u rdf:type[rdf:type owl:Restriction;
                      owl:onProperty P;
                      owl:hasValue D] .
Then add              u P v .
```

PROPERTY RESTRICTIONS: HASVALUE

- `[rdf:type owl:Restriction; owl:onProperty` *P*`; owl:hasValue` *v*`]` is a particular form of `owl:someValuesFrom`. It denotes the class consisting of all individuals *u* for which *u P v* holds.

Example: a class for all of Mary's children

```
:MarysChildren rdf:type         owl:Class ;
               owl:intersectionOf ( :Person
                                    [rdf:type owl:Restriction;
                                     owl:onProperty :hasParent;
                                     owl:hasValue :mary]
                                  ) .
:john          rdf:type         :Person .
:john          :hasParent       :mary .
```

PROPERTY RESTRICTIONS: HASVALUE

- `[rdf:type owl:Restriction; owl:onProperty` *P*`; owl:hasValue` *v*`]` is a particular form of `owl:someValuesFrom`. It denotes the class consisting of all individuals *u* for which *u P v* holds.

Example: a class for all of Mary's children

```
:MarysChildren rdf:type           owl:Class ;
               owl:intersectionOf ( :Person
                                    [rdf:type owl:Restriction;
                                     owl:onProperty :hasParent;
                                     owl:hasValue :mary]
                                  ) .
:john          rdf:type           :Person .
:john          :hasParent         :mary .
:john          rdf:type           :MarysChildren .
```

CARDINALITY RESTRICTIONS: MINCARDINALITY

- `[rdf:type owl:Restriction; owl:onProperty` P`; owl:minCardinality "`i`"^^xsd:nonNegativeInteger ]` is used to denotes the class consisting of all individuals u for which there are at least i distinct individuals $v_1, v_2, \ldots, v_i$ such that $u\ P\ v_1, v_2, \ldots, v_i$.

CARDINALITY RESTRICTIONS: MINCARDINALITY

- `[rdf:type owl:Restriction; owl:onProperty` P`; owl:minCardinality ”`i`”^^xsd:nonNegativeInteger ]` is used to denotes the class consisting of all individuals u for which there are at least i distinct individuals $v_1, v_2, \ldots, v_i$ such that $u\ P\ v_1, v_2, \ldots, v_i$.

Example: every person has at least two parents

```
:Person  rdf:type        owl:Class ;
         owl:subClassOf  [rdf:type owl:Restriction;
                          owl:onProperty :hasParent;
                          owl:minCardinality ”2”^^xsd:nonNegativeInteger].
```

CARDINALITY RESTRICTIONS: MAXCARDINALITY

- `[rdf:type owl:Restriction; owl:onProperty` P`; owl:maxCardinality` `"`i`"^^xsd:nonNegativeInteger ]` is used to denotes the class consisting of all individuals u for which there are at most i distinct individuals $v_1, v_2, \ldots, v_i$ such that $u\ P\ v_1, v_2, \ldots, v_i$.

CARDINALITY RESTRICTIONS: MAXCARDINALITY

- `[rdf:type owl:Restriction; owl:onProperty` P`; owl:maxCardinality` `”`i`”^^xsd:nonNegativeInteger ]` is used to denotes the class consisting of all individuals u for which there are at most i distinct individuals $v_1, v_2, \ldots, v_i$ such that $u\ P\ v_1, v_2, \ldots, v_i$.

Example: every person has at most two parents

```
:Person  rdf:type          owl:Class ;
         owl:subClassOf    [rdf:type owl:Restriction;
                            owl:onProperty :hasParent;
                            owl:maxCardinality ”2”^^xsd:nonNegativeInteger].
```

CARDINALITY RESTRICTIONS: MAXCARDINALITY

- `[rdf:type owl:Restriction; owl:onProperty` *P*`; owl:cardinality "`*i*`"^^xsd:nonNegativeInteger ]` is used to denotes the class consisting of all individuals u for which there are at exactly i distinct individuals $v_1, v_2, \ldots, v_i$ such that $u\ P\ v_1, v_2, \ldots, v_i$.

CONSISTENCY OF AN OWL DOCUMENT/GRAPH WITH OWL

- An OWL document is consistent (also called satisfiable) if it does not contain any contradictions
- An OWL document is class consistent if none of its classes is equivalent to `owl:Nothing`

Example of an inconsistent document:

```
:Male      rdf:type          owl:Class .
:Female    rdf:type          owl:Class .
:Male      owl:disjointWith  :Female .
:john      rdf:type          :Male .
:john      rdf:type          :Female .
```

CONSISTENCY OF AN OWL DOCUMENT/GRAPH WITH OWL

- An OWL document is consistent (also called satisfiable) if it does not contain any contradictions
- An OWL document is class consistent if none of its classes is equivalent to `owl:Nothing`

Example of a consistent but class inconsistent document:

```
 :Book        rdf:type          owl:Class .
:Publication  rdf:type          owl:Class .
:Book         rdfs:subClassOf   :Publication ;
              owl:disjointWith  :Publication .
```

Part III: Reasoning with OWL

TYPICAL REASONING TASKS

- Checking consistency (also called satisfiability).
- Checking class consistency
- Computing all relationships between classes in documents.
- Computing all instances of a given class.

TYPICAL REASONING TASKS

- Checking consistency (also called satisfiability).
- Checking class consistency
- Computing all relationships between classes in documents.
- Computing all instances of a given class.

- If the OWL vocabulary is carelessly mingled with the RDF Schema vocabulary, then all of these task are undecidable: no algorithm exists that does these tasks and is guaranteed to terminate.
- As such, the OWL 1.0 standard defines several OWL dialects, including dialects for which these tasks are decidable:
  - OWL Full
  - OWL DL
  - OWL Lite

OWL VERSION 1.0 FULL

OWL version 1 Full

- Allows all of the OWL vocabulary
- Allows the combination of these vocabulary terms in arbitrary ways with RDF and RDF schema (including `rdfs:Class`, `rdfs:Resource`, `rdfs:Property`)
- OWL Full is fully upward-compatible with RDF, both syntactically and semantically
- OWL Full is so powerful that it is undecidable
    - Hence, there is no complete or efficient reasoning support
    - Undecidability due, among other reasons, to the fact the we allow classes to be members of themselves.

OWL VERSION 1.0 DL

OWL version 1 DL

- OWL 1 DL = OWL Description Logic
- A sublanguage of OWL Full that restricts how OWL constructors can be used. (See next slide)
- It corresponds to a well-studied description logic
- Description logics are well-known knowledge representation formalisms. They are fragments of first order logic with decidable and often efficient reasoning support.
- As such, OWL 1 DL permits efficient reasoning support
- But we lose full compatibility with RDF and RDF Schema
    - Every legal OWL DL document is a legal RDF document
    - But not every legal RDF document is a legal OWL DL document.

OWL VERSION 1.0 DL (2/2)

OWL version 1 DL conditions

- The only terms from RDF and RDFS that can be used are `rdf:type`, `rdfs:domain`, `rdfs:range`, `rdfs:subClassOf`, `rdfs:subPropertyOf` (in particular, `rdfs:Class` and `rdfs:Property` cannot be used).

OWL VERSION 1.0 DL (2/2)

OWL version 1 DL conditions

- The only terms from RDF and RDFS that can be used are `rdf:type`, `rdfs:domain`, `rdfs:range`, `rdfs:subClassOf`, `rdfs:subPropertyOf` (in particular, `rdfs:Class` and `rdfs:Property` cannot be used).
- Type separation and declaration: an OWL DL document must treat classes, abstract properties, concrete properties, and datatypes as disjoint things. Furthermore, classes and properties must be declared explicitly.

  The following is hence not allowed (on classes we should only use `subClassOf` and the like)

  ```
  :Book     rdf:type       owl:Class .
  :Book     :germanName    ”Buch” .
  :Book     :frenchName    ”Livre” .
  ```

OWL VERSION 1.0 DL (2/2)

OWL version 1 DL conditions

- The only terms from RDF and RDFS that can be used are `rdf:type`, `rdfs:domain`, `rdfs:range`, `rdfs:subClassOf`, `rdfs:subPropertyOf` (in particular, `rdfs:Class` and `rdfs:Property` cannot be used).
- Type separation and declaration: an OWL DL document must treat classes, abstract properties, concrete properties, and datatypes as disjoint things. Furthermore, classes and properties must be declared explicitly.

  The following is hence not allowed (on classes we should only use `subClassOf` and the like)

  ```
  :Book     rdf:type       owl:Class .
  :Book     :germanName    ”Buch” .
  :Book     :frenchName    ”Livre” .
  ```

- Restricted use of concrete roles: `owl:inverseOf`, `owl:TransitiveProperty`, `owl:InverseFunctionalProperty`, and `owl:SymmetricProperty` must not be used for concrete properties.

OWL VERSION 1.0 DL (2/2)

OWL version 1 DL conditions

- The only terms from RDF and RDFS that can be used are `rdf:type`, `rdfs:domain`, `rdfs:range`, `rdfs:subClassOf`, `rdfs:subPropertyOf` (in particular, `rdfs:Class` and `rdfs:Property` cannot be used).
- Type separation and declaration: an OWL DL document must treat classes, abstract properties, concrete properties, and datatypes as disjoint things. Furthermore, classes and properties must be declared explicitly.

  The following is hence not allowed (on classes we should only use `subClassOf` and the like)

  ```
  :Book      rdf:type       owl:Class .
  :Book      :germanName    ”Buch” .
  :Book      :frenchName    ”Livre” .
  ```

- Restricted use of concrete roles: `owl:inverseOf`, `owl:TransitiveProperty`, `owl:InverseFunctionalProperty`, and `owl:SymmetricProperty` must not be used for concrete properties.
- Restricted use of abstract properties: cardinality restrictions via `owl:cardinality`, `owl:minCardinality`, `owl:maxCardinality` must not be used with transitive properties, inverses of transitive properties, or superproperties of transitive properties. (To obtain decidability.)

OWL VERSION 1.0 LITE

OWL version 1 Lite

- Restricts OWL 1 DL further to a subset of the language constructors. (E.g., no disjointness statements, cardinality statements, ...)
- Designed to be easier
  - to understand, for users (ontology builders)
  - implement, for tool builders
- The disadvantage is its restricted expressivity

OWL VERSION 1.0 LITE

OWL version 1 Lite

- Restricts OWL 1 DL further to a subset of the language constructors. (E.g., no disjointness statements, cardinality statements, …)
- Designed to be easier
  - to understand, for users (ontology builders)
  - implement, for tool builders
- The disadvantage is its restricted expressivity
- And actually, it proved as difficult to implement as OWL DL.

OWL VERSION 2.0 DIALECTS

In OWL 2, there are only two dialects

- **OWL Full**: all features, but reasoning is undecidable
- **OWL DL**[a]: expressive, but not entirely compatible with RDF Schema; efficient reasoning support

---

[a] DL=Description Logic

- OWL 1 DL is a strict subset of OWL 2 DL
- OWL Lite was dropped as a dialect in OWL 2 because it proved to be as hard to implement as OWL 1 DL (yet it is significantly less expressive).

OWL VERSION 2 PROFILES

OWL 2 introduces three new fragments of OWL 2 DL, called profiles

- Full OWL 2 DL has requires exponential time for most reasoning tasks.
- OWL 2 EL is designed for applications where very large ontologies are needed, and where expressive power can be traded for performance guarantees.
  Reasoning complexity: polynomial time in the size of the OWL ontology.
- OWL 2 QL is designed for applications where relatively lightweight ontologies are used to organize large numbers of individuals and where it is useful or necessary to access the data directly via relational queries (e.g., SQL).

  Reasoning complexity: polynomial time in the size of the OWL ontology.
- OWL 2 RL is designed for applications where relatively lightweight ontologies are used to organize large numbers of individuals and where it is useful or necessary to operate directly on data in the form of RDF triples.

  Reasoning complexity: polynomial time in the size of the OWL ontology.

OWL VERSION 2.0 EXTRA FEATURES

- OWL version 1.0 was standardized as a recommendation in 2004.
- OWL version 2.0 (second edition) proposes a backwards-compatible update to OWL 1.0. It features several extensions to OWL version 1.0
  - keys;
  - property chains;
  - richer datatypes, data ranges;
  - qualified cardinality restrictions;
  - asymmetric, reflexive, and disjoint properties; and
  - enhanced annotation capabilities

See book/handouts!

EXAMPLES OF OWL ONTOLOGIES (1/3)

FOAF: Friend of a Friend

- vocabulary to link people and information using the Web

> *FOAF is a project devoted to linking people and information using the Web. Regardless of whether information is in people's heads, in physical or digital documents, or in the form of factual data, it can be linked. FOAF integrates three kinds of network: social networks of human collaboration, friendship and association; representational networks that describe a simplified view of a cartoon universe in factual terms, and information networks that use Web-based linking to share independently published descriptions of this inter-connected world. FOAF does not compete with socially-oriented Web sites; rather it provides an approach in which different sites can tell different parts of the larger story, and by which users can retain some control over their information in a non-proprietary format.*

EXAMPLES OF OWL ONTOLOGIES (2/3)

GoodRelations: a vocabulary for e-commerce

- a lightweight ontology for annotating offerings and other aspects of e-commerce on the Web.
- GoodRelations is the only OWL DL ontology officially supported by both Google and Yahoo.

- It provides a standard vocabulary for expressing things like that a particular Web site describes an offer to sell cellphones of a certain make and model at a certain price, that a pianohouse offers maintenance for pianos that weigh less than 150 kg, or that a car rental company leases out cars of a certain make and model from a particular set of branches across the country.
- Also, most if not all commercial and functional details of e-commerce scenarios can be expressed, e.g. eligible countries, payment and delivery options, quantity discounts, opening hours, etc.

EXAMPLES OF OWL ONTOLOGIES (3/3)

SNOMED CT

- an ontology of medical terms.
- used in clinical documentation and reporting
- standard ontology for patient records etc. in Belgium

More specifically, the following sample computer applications use SNOMED CT:

- Electronic Health Record Systems
- Computerized Provider Order Entry CPOE such as E-Prescribing or Laboratory Order Entry
- Catalogues of clinical services; e.g., for Diagnostic Imaging procedures
- Knowledge databases used in clinical decision support systems (CDSS)
- Remote Intensive Care Unit Monitoring
- Laboratory Reporting
- Emergency Room Charting
- Cancer Reporting
- Genetic Databases

REFERENCES

- P. Hitzler, M. Krötzsch, S. Rudolph. *Foundations of Semantic Web technologies*. Chapter 4.
- D. Allemang, J. Hendler. *Semantic Web for the Working Ontologist.*. Chapter 9-11.