

# Ontologies

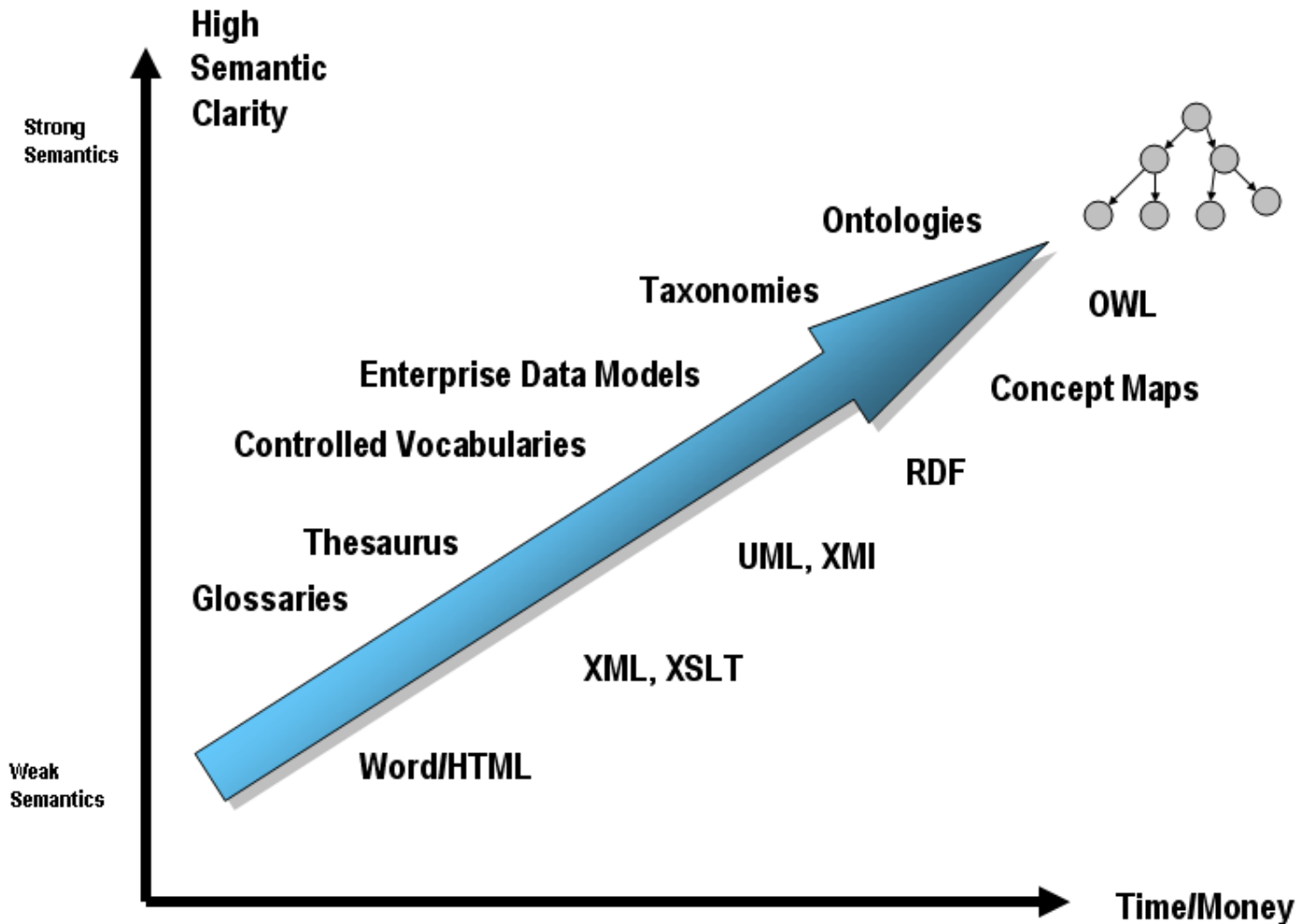
OWL2



POLITECNICO  
DI TORINO

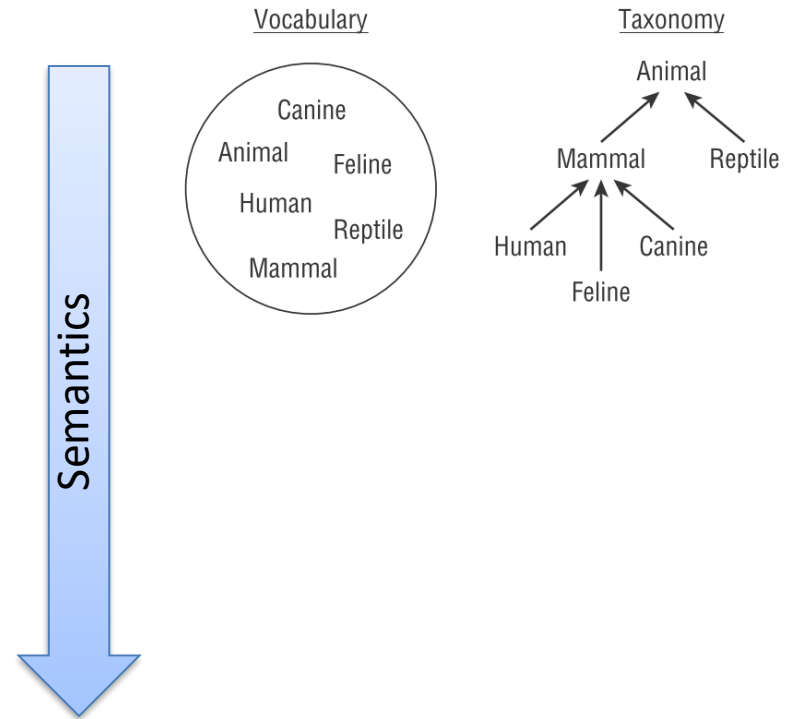


# Why Ontologies?

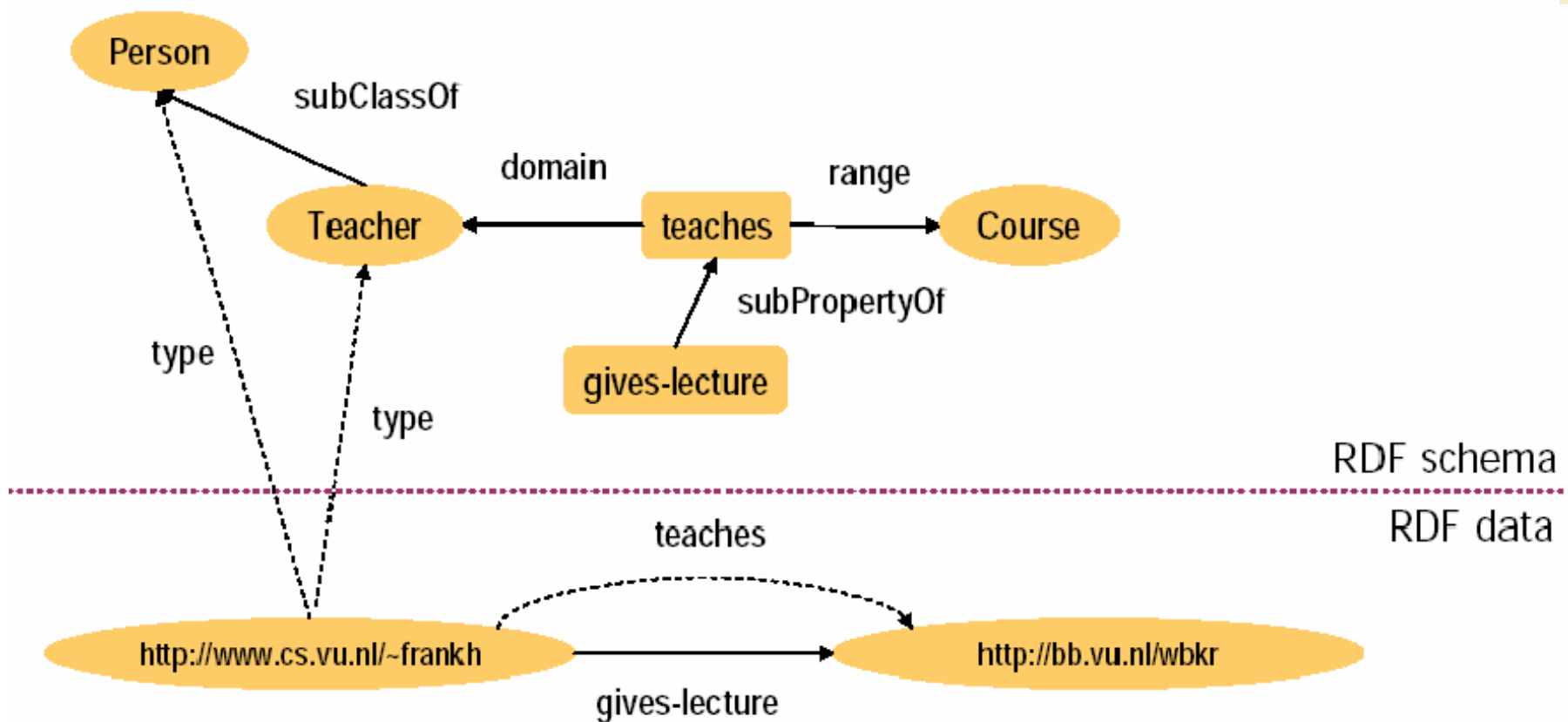


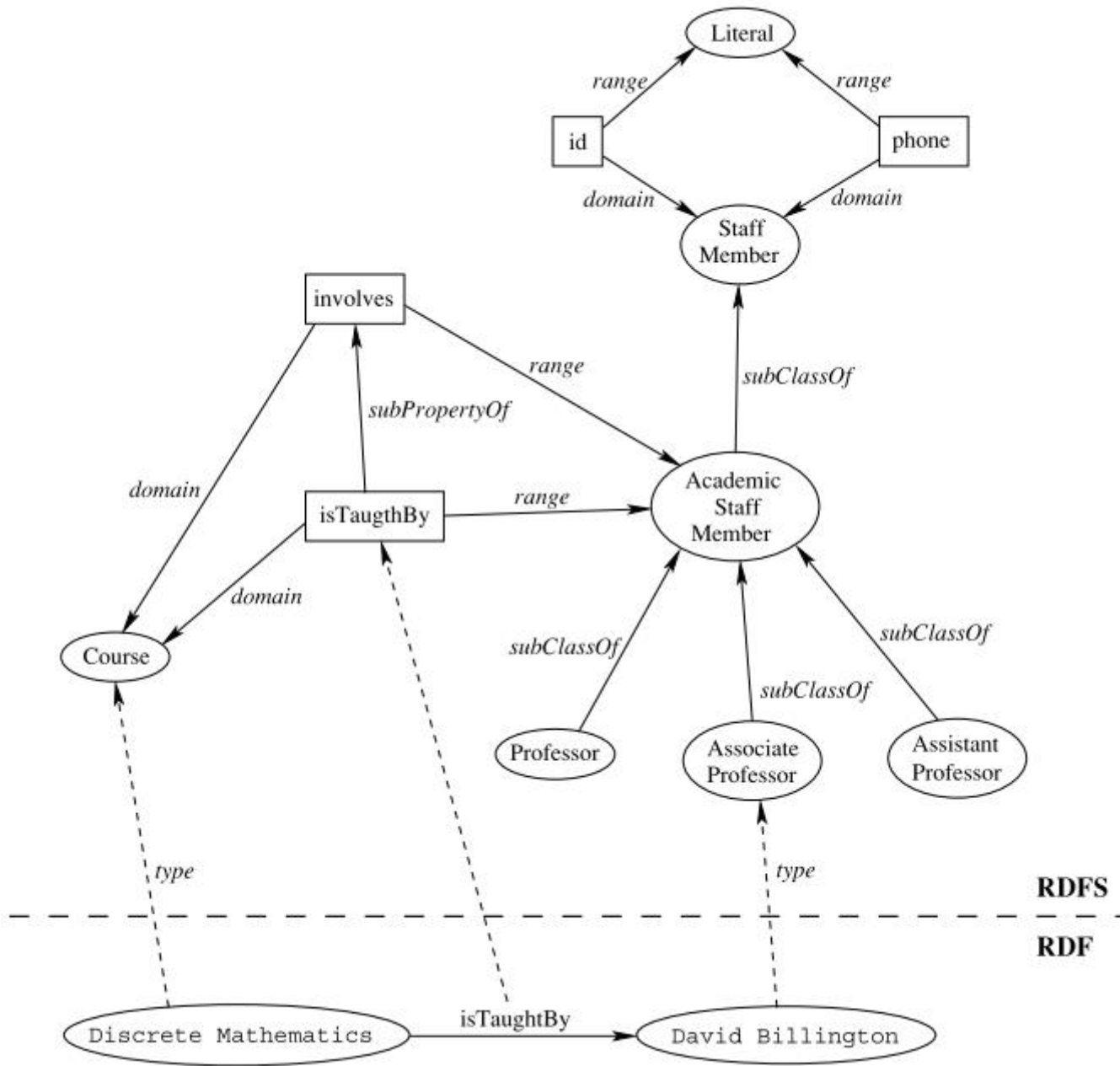
# Knowledge Organization Systems

- Term Lists
  - Authority files
  - Glossaries
  - Dictionaries, Vocabularies
  - Gazetteers
- Classifications and Categories
  - Subject headings
  - Classification schemes
  - Taxonomies
  - Categorization Schemes.
- Relationship Lists
  - Thesauri
  - Semantic networks
  - Ontologies



# RDF / RDF Schema





# RDFS problems

- RDFS is too “weak” to describe resources with a suitable level of details
  - range and domain cannot be localized (e.g. the range of hasChild is a person when applied to a person, elephant when applied to an elephant)
  - no constraints on existence or cardinality (e.g. all instances of persons have one and only one mother which is a person, and have exactly two parents)
  - it is not possible to define transitive, inverse or symmetrical statements (e.g. part of is a transitive property, hasPart is the inverse of isPartOf, touches is symmetrical)
- Reasoning is not well supported
  - Non standard semantics, no native reasoner exists

# Requirements for an ontology language

- Extend existing Web standards
  - XML, RDF, RDFS, ...
- Easy to understand and to use
  - based on well known knowledge representation (KR) languages
- Formally specified
- Adequate expressive power
- Automatic support for reasoning

# Modeling knowledge

- OWL 2 is a knowledge representation language, designed to formulate, exchange and reason with knowledge about a domain of interest
- Basic notions
  - Axioms: the basic statements that an OWL ontology expresses
  - Entities: elements used to refer to real-world objects
  - Expressions: combinations of entities to form complex descriptions from basic ones
- The results of the modeling processes are called ontologies
- Knowledge consists of elementary pieces that are often referred to as statements or propositions
- Statements that are made in an ontology are called axioms in OWL 2



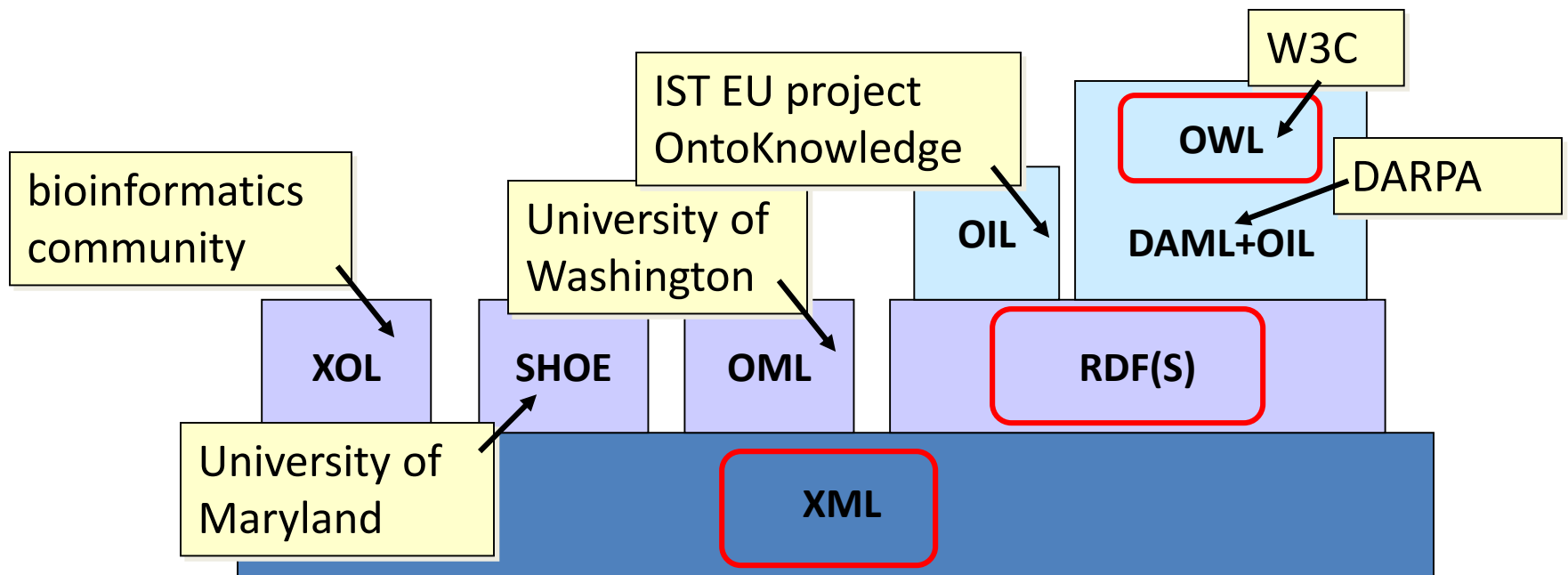
# Modeling knowledge

- When humans think, they draw consequences from their knowledge
- A statement is a consequence of other statements essentially means that this statement is true whenever the other statements are
- In OWL terms: “a set of statements A entails a statement a if in any state of affairs wherein all statements from A are true, also a is true”
- A set of statements may be
  - Consistent, if there is a possible state of affairs in which all the statements in the set are jointly true
  - Inconsistent, if there is no such state of affairs
- The formal semantics of OWL specifies, in essence, for which possible “states of affairs” a particular set of OWL statements is true

# What's in an Ontology?

- Classes
- Instances
- Properties
  - Object Properties
  - DataType Properties
- Restrictions
- Annotations
- Different ontology standards use slightly differing terminology

# “History” of Web languages



# Web Ontology Language (OWL)

- Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and relations between things
- Computational logic-based language such that knowledge expressed in OWL can be reasoned with by computer programs either to verify the consistency of that knowledge or to make implicit knowledge explicit
- OWL documents, known as ontologies, can be published in the World Wide Web and may refer to or be referred from other OWL ontologies
- OWL is not a programming language
  - it is declarative, i.e. it describes a state of affairs in a logical way
- OWL: W3C Recommendation, Feb 10th 2004
- OWL 2: Revised W3C Recommendation, October 27th 2009

# Web Ontology Language (OWL)

- More expressive than RDFS
  - Identity equivalence/difference
  - sameAs, differentFrom, equivalentClass/Property
- More expressive class definitions
  - Class intersection, union, complement, disjointness
  - Cardinality restrictions
- More expressive property definitions
  - Object/Datatype properties
  - Transitive, functional, symmetric, inverse properties
  - Value restrictions

# Web Ontology Language (OWL)

- What can be done with OWL?
  - Consistency checks – are there contradictions in the logical model?
  - Satisfiability checks – are there classes that cannot have any instances?
  - Classification – what is the type of a particular instance?

# OWL basics

- Statements in OWL normally refer to objects of the world and describe them by putting them into categories (like “Mary is female”) or saying something about their relation (“John and Mary are married”)
- All atomic constituents of statements, be they objects (John, Mary), categories (female) or relations (married) are called entities
- In OWL 2
  - objects are called “individuals”
  - categories are called “classes”
  - relations are called “properties”

# OWL basics

- Properties in OWL 2 are further subdivided
  - Object properties relate objects to objects (like a person to their spouse)
  - Datatype properties assign data values to objects (like an age to a person)
  - Annotation properties are used to encode information about (parts of) the ontology itself (like the author and creation date of an axiom) instead of the domain of interest



# OWL basics

- Names of entities can be combined into expressions using so called constructors
  - As a basic example, the atomic classes “female” and “professor” could be combined conjunctively to describe the class of female professors
  - The latter would be described by an OWL class expression, that could be used in statements or in other expressions
- Expressions can be seen as new entities which are defined by their structure
  - In OWL, the constructors for each sort of entity vary greatly
  - The expression language for classes is very rich and sophisticated
  - The expression language for properties is much less so

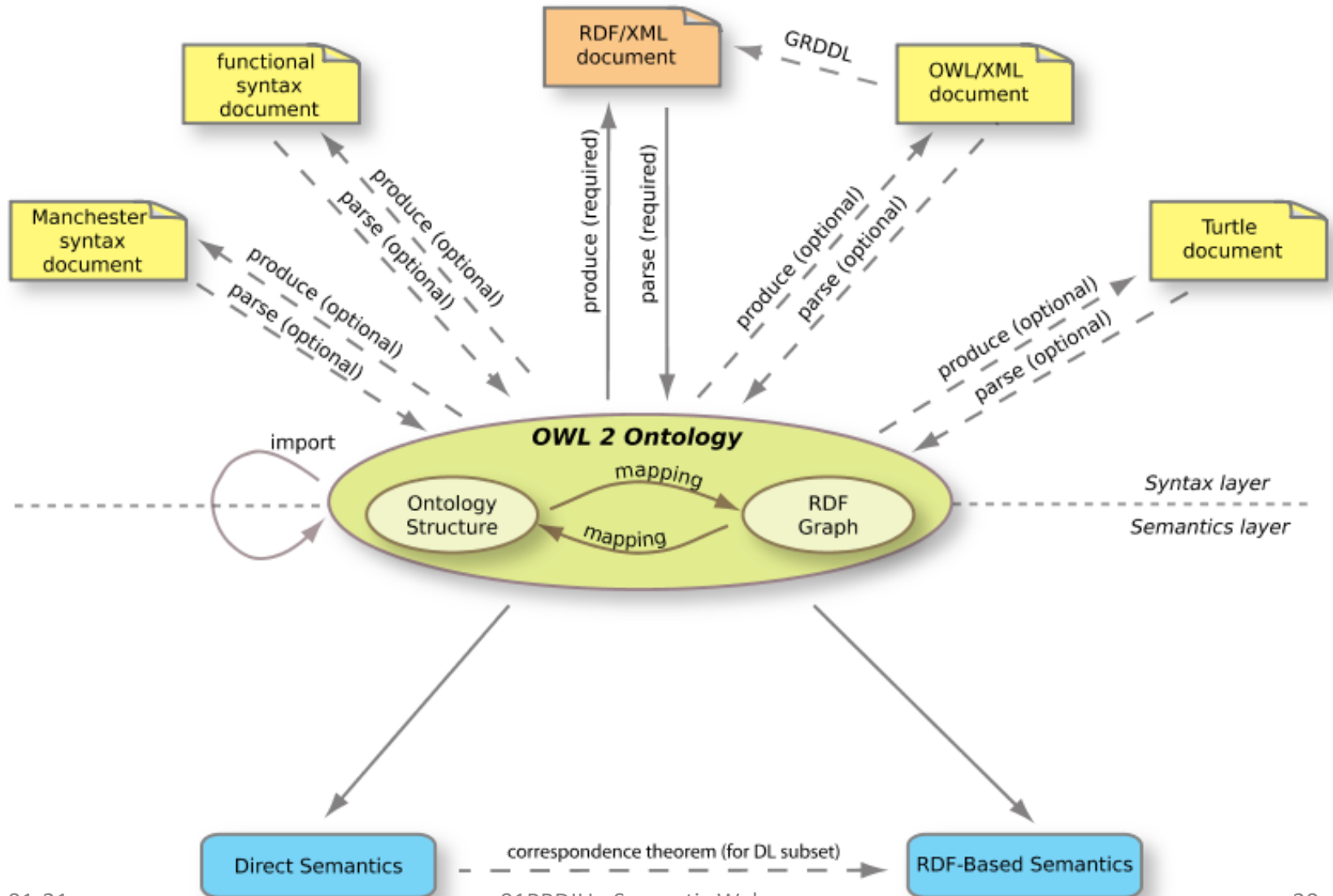
# OWL Main Assumptions

- **Open World Assumption**
- «The truth of a statement is independent of whether it is known»
- Not knowing whether a statement is explicitly true does not imply that the statement is false, it is simply unknown (or not-yet-known)
- New information must always be additive (even if contradictory)

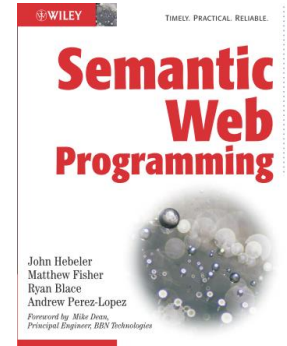
# OWL Main Assumptions

- **No Unique Names Assumption**
- «Unless explicitly stated otherwise, you cannot assume that resources that are identified by different URIs are different»
- Both Classes and Individuals

# OWL2 syntax and semantic



# OWL Building Blocks



## Main

- Class
  - a set of resources
- Individual
  - any resource that is a member of at least one class
- Property
  - used to describe a resource

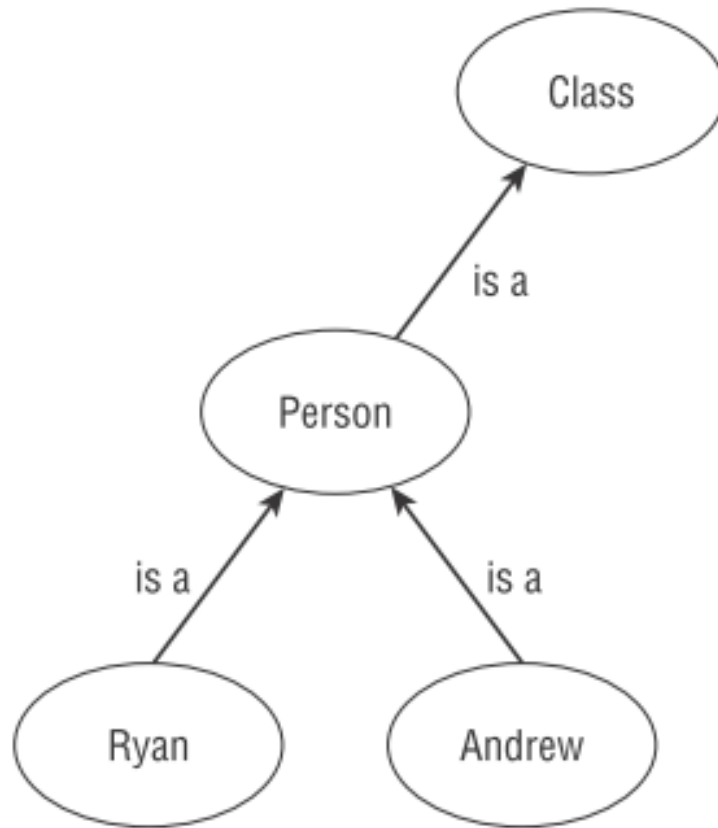
## Other

- Headers
  - define and describe the the ontology itself
- Annotations
  - add nonsemantic descriptive information
- Datatype definitions
  - describe ranges of values

# Classes and Individuals

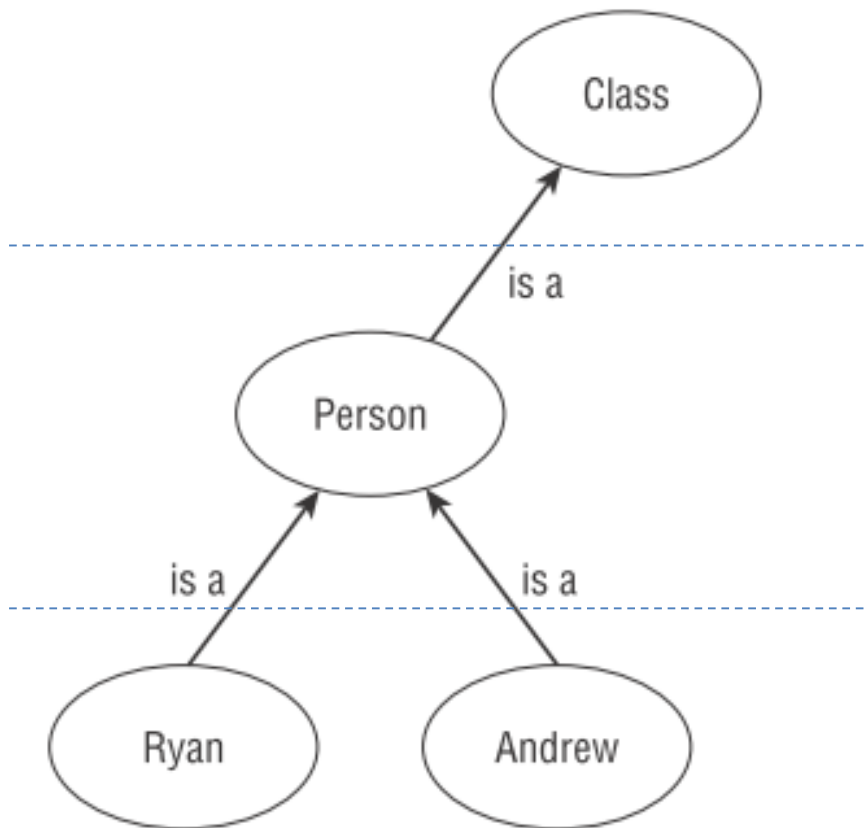
- A **class** is a special kind of resource that represents a set of resources that share common characteristics or are similar in some way
- A resource that is a member of a class is called an **individual** and represents an instance of that class
- Warning: Both are resources, no strict division (depends on the modeling approach)

# Example



- `ex:Person` is a class (of type `owl:Class`)
- `ex:Ryan` and `ex:Andrew` are instances of the class `ex:Person`
- Unless you examine the `rdf:type`, you cannot determine

# Example



- In *Description Logic*:
- Class
  - Metamodeling level
- Person
  - T-Box level
  - terminology box
  - TBox statements describe a conceptualization, a set of **concepts and properties** for these concepts
- Ryan, Andrew
  - A-Box level
  - assertion box
  - a **fact** associated with a terminological vocabulary



# Class Membership

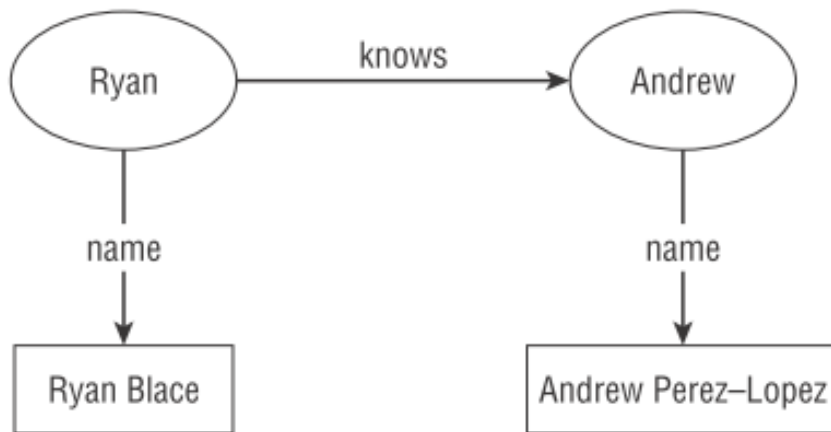
- Individuals can become **members** of classes
  - Directly
    - by **asserting** their membership explicitly
    - `ex:Ryan rdf:type ex:Person .`
  - Indirectly
    - by defining the **membership conditions** for a class such that it can be inferred that a resource is a member of that class
    - more powerful, may lead to discovery of new facts

# Properties

- A property in OWL is a resource that is used as a predicate in statements that describe individuals
- Object properties link individuals to other individuals
- Datatype properties link individuals to literal values

# Example

- `ex:knows`
  - Object Property
- `ex:name`
  - Datatype property



# OWL2 syntaxes

- Various syntaxes available for OWL, to serve various purposes
- Functional-Style syntax
  - Designed to be easier for specification purposes and to provide a foundation for the implementation of OWL 2 tools such as APIs and reasoners
- RDF/XML syntax
  - Just RDF/XML, with a particular translation for the OWL constructs
  - This is the only syntax that is mandatory to be supported by all OWL 2 tools
- Turtle syntax
  - Turtle serializations for the RDF-based syntax

# OWL2 syntaxes

- Manchester syntax
  - Designed to be easier for non-logicians to read
- OWL/XML syntax
  - an XML syntax for OWL defined by an XML schema
- There are tools that can translate between the different syntaxes

# Elements of an Ontology

- Ontology Header
- Annotations
- Basic Classification
  - Classes and Individuals
  - `rdfs:SubClassOf`
  - `owl:Thing` and `owl:Nothing`
- Defining and Using Properties
- Property Domain and Range
- Describing Properties
  - `rdfs:subPropertyOf`
  - Top and Bottom Properties
  - Inverse Properties
  - Disjoint Properties
  - Property Chains
  - Symmetric, Reflexive, and Transitive Properties
  - Functional and Inverse Functional Properties
  - Keys
- Datatypes
  - Data type Restrictions
  - Defining Datatypes in Terms of Other Datatypes
- Negative Property Assertions
- Property Restrictions
  - Value Restrictions
  - Cardinality Restrictions
  - Qualified Cardinality Restrictions
- Advanced Class Description
  - Enumerating Class Membership
  - Set Operators
  - Disjoint Classes
- Equivalence in OWL
  - Equivalence among Individuals
  - Equivalence among Classes and Properties

# Ontology header

- Ontology declaration (XML syntax)

```
<rdf:RDF xmlns:owl =http://www.w3.org/2002/07/owl#"
  xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd ="http://www.w3.org/2001/XMLSchema#">
```

- Ontology metadata (information about the ontology)

```
<owl:Ontology rdf:about="">
  <rdfs:comment>An example OWL ontology</rdfs:comment>
  <owl:priorVersion
    rdf:resource="http://www.mydomain.org/uni-ns-old"/>
  <owl:imports
    rdf:resource="http://www.mydomain.org/persons"/>
  <rdfs:label>University Ontology</rdfs:label>
</owl:Ontology>
```

# Ontology imports

- The property `owl:imports` specifies the set of ontologies that are referred to in the importing ontology.

```
@prefix ex: <http://example.org/ >.
```

```
ex: rdf:type owl:Ontology;  
    rdfs:comment "This is an example ontology";  
    owl:imports <http://example.org/example-import>.
```



# Annotations

- Statements that describe resources using "annotation properties"
- Annotation properties are semantics-**free** properties
- Can describe **any** resource or axiom in an ontology, including the ontology itself
- Annotation properties are primarily used by **tools** and applications to interact with humans

# Annotation properties

PROPERTY	DESCRIPTION OF USE
<code>rdfs:label</code>	A label, or terse description of the subject resource.
<code>rdfs:comment</code>	A comment about the subject resource.
<code>owl:versionInfo</code>	Information about the subject ontology or resource version. Frequently used to embed source control metadata.
<code>rdfs:seeAlso</code>	Used to specify that another resource may hold more information about the subject resource. Not commonly used.
<code>rdfs:isDefinedBy</code>	Used to specify that another resource defines the subject resource. Not commonly used.

# owl:Class

- The resource owl:Class represents the class containing all OWL classes
- Every class in OWL must be a member of owl:Class
- Every resource that has an rdf:type of owl:Class is a class.

# Classes and instances

- Example: “Mary is a person”

- Functional-Style Syntax

```
ClassAssertion( :Person :Mary )
```

- RDF/XML Syntax

```
<Person rdf:about="Mary"/>
```

- Turtle Syntax

```
:Mary rdf:type :Person .
```

- Manchester Syntax

```
Individual: Mary  
Types: Person
```

- OWL/XML Syntax

```
<ClassAssertion>  
  <Class IRI="Person"/>  
  <NamedIndividual IRI="Mary"/>  
</ClassAssertion>
```

# Example

```
@prefix ex: <http://example.org/>.

# Canine and Human are owl classes
ex:Canine rdf:type owl:Class.
ex:Human  rdf:type owl:Class.

# Daisy is an instance of the class Canine
ex:Daisy  rdf:type ex:Canine.

# Ryan is an instance of the class Human
ex:Ryan   rdf:type ex:Human.
```

# Class extension

- The set of individuals that are members of a class is considered its **class extension**
- An individual may belong to the extension of different classes
- An OWL class has intrinsic meaning beyond its class extension: two classes can have exactly the same class extension but still represent unique classes (extension equivalence is not a sufficient condition for class equivalence)

# Class restrictions

- Additional information about the class is given in the form of *restrictions*, constructs that restrict the membership of the class
  - subclass relationships
  - explicit membership enumeration
  - property restrictions
  - class-based set operations

# Class hierarchies

- Example: “Woman is a subclass of Person”

- Functional-Style Syntax

```
SubClassOf ( :Woman :Person )
```

- RDF/XML Syntax

```
<owl:Class rdf:about="Woman">  
  <rdfs:subClassOf rdf:resource="Person"/>  
</owl:Class>
```

- Turtle Syntax

```
:Woman rdfs:subClassOf :Person .
```

- Manchester Syntax

```
Class: Woman  
SubClassOf: Person
```

- OWL/XML Syntax

```
<SubClassOf>  
  <Class IRI="Woman"/>  
  <Class IRI="Person"/>  
</SubClassOf>
```



# Example

```
@prefix ex: <http://example.org/>.
```

```
ex:Mammal rdf:type owl:Class.
```

```
# Canine is a subclass of Mammal
```

```
ex:Canine rdf:type owl:Class;  
          rdfs:subClassOf ex:Mammal.
```

```
# Human is a subclass of Mammal
```

```
ex:Human rdf:type owl:Class;  
         rdfs:subClassOf ex:Mammal.
```

```
# Both Daisy and Ryan are implicitly members of the class Mammal
```

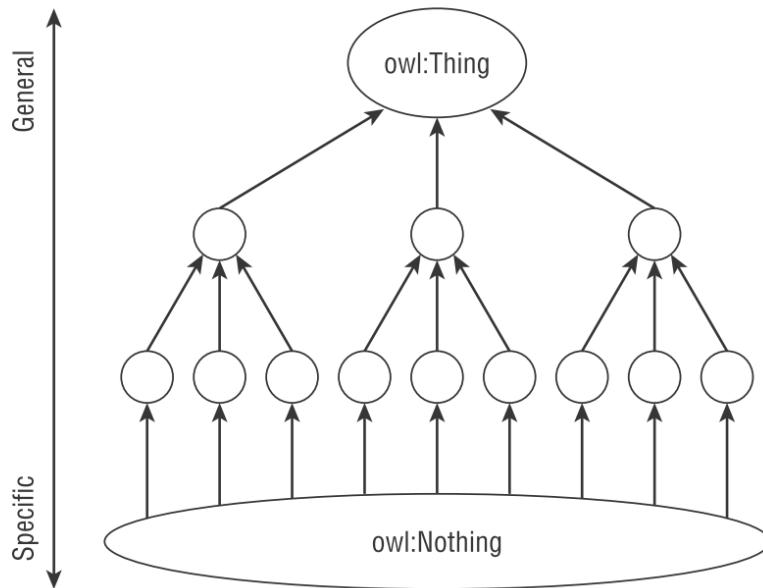
```
ex:Daisy rdf:type ex:Canine.
```

```
ex:Ryan rdf:type ex:Human.
```

# Subclass or Instance?

- Depends on the modeling context
- A subclass represents a **subset** of the members of the parent class
- An instance represents an **individual** member of a class

# Thing and Nothing



- No class can be more general than `owl:Thing` or more specific than `owl:Nothing`
- Every OWL class is implicitly a subclass of `owl:Thing`
- `owl:Nothing` is implicitly a subclass of every OWL class

# Properties

## Object properties

- Domain: Individuals of a (named or implicit) class
- Range: Individuals of a (named or implicit) class
- owl:ObjectProperty

## Datatype properties

- Domain: Individuals of a (named or implicit) class
- Range: Literals (possibly typed)
- owl:DatatypeProperty

# Object properties

- Example: “Mary is John’s wife”
- Functional-Style Syntax

```
ObjectPropertyAssertion( :hasWife :John :Mary )
```

- RDF/XML Syntax

```
<rdf:Description rdf:about="John">  
  <hasWife rdf:resource="Mary"/>  
</rdf:Description>
```

- Turtle Syntax

```
:John :hasWife :Mary .
```

- Manchester Syntax

```
Individual: John  
Facts: hasWife Mary
```

- OWL/XML Syntax

```
<ObjectPropertyAssertion>  
  <ObjectProperty IRI="hasWife"/>  
  <NamedIndividual IRI="John"/>  
  <NamedIndividual IRI="Mary"/>  
</ObjectPropertyAssertion>
```

# Datatypes properties

- Example: “John’s age is 51”
- Functional-Style Syntax

```
DataPropertyAssertion( :hasAge :John "51"^^xsd:integer )
```

- RDF/XML Syntax

```
<Person rdf:about="John">  
  <hasAge rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">51</hasAge>  
</Person>
```

- Turtle Syntax

```
:John :hasAge 51 .
```

- Manchester Syntax

```
Individual: John  
Facts: hasAge "51"^^xsd:integer
```

- OWL/XML Syntax

```
<DataPropertyAssertion>  
  <DataProperty IRI="hasAge"/>  
  <NamedIndividual IRI="John"/>  
  <Literal datatypeIRI="  
    http://www.w3.org/2001/  
   /XMLSchema#integer">51</Literal>  
</DataPropertyAssertion>
```

# Example

```
@prefix ex: <http://example.org/>.
```

```
...
```

```
#name is a datatype property because it refers to literals
```

```
ex:name rdf:type owl:DatatypeProperty.
```

```
#breed is an object property because it refers to an individual
```

```
ex:breed rdf:type owl:ObjectProperty.
```

```
ex:Daisy ex:name "Daisy";
```

```
    ex:breed ex:GoldenRetriever.
```

```
ex:Ryan ex:name "Ryan Blace".
```

# Domain and Range

- Property restrictions:
- **rdfs:domain** — Specifies the type of all individuals who are the subject of statements using the property being described
- **rdfs:range** — Specifies the type of all individuals (or the datatype of all literals) that are the object of statements using the property being described
- Warning: they are globally asserted relationships



# Domain and range restrictions

- Example: “if B is the wife of A, B is a woman and A is a man”

- Functional-Style Syntax

```
ObjectPropertyDomain( :hasWife :Man )  
ObjectPropertyRange( :hasWife :Woman )
```

- RDF/XML Syntax

```
<owl:ObjectProperty rdf:about="hasWife">  
  <rdfs:domain rdf:resource="Man"/>  
  <rdfs:range rdf:resource="Woman"/>  
</owl:ObjectProperty>
```

- Turtle Syntax

```
:hasWife  
  rdfs:domain :Man ;  
  rdfs:range :Woman .
```

# Domain and range restrictions

- Example: “if B is the wife of A, B is a woman and A is a man”

- Manchester Syntax

```
ObjectProperty: hasWife
  Domain: Man
  Range: Woman
```

- OWL/XML Syntax

```
<ObjectPropertyDomain>
  <ObjectProperty IRI="hasWife"/>
  <Class IRI="Man"/>
</ObjectPropertyDomain>
<ObjectPropertyRange>
  <ObjectProperty IRI="hasWife"/>
  <Class IRI="Woman"/>
</ObjectPropertyRange>
```

# Property hierarchies

- Example: “hasWife is a subproperty of hasSpouse”

- Functional-Style Syntax

```
SubObjectPropertyOf( :hasWife :hasSpouse )
```

- RDF/XML Syntax

```
<owl:ObjectProperty rdf:about="hasWife">  
  <rdfs:subPropertyOf rdf:resource="hasSpouse"/>  
</owl:ObjectProperty>
```

- Turtle Syntax

```
:hasWife rdfs:subPropertyOf :hasSpouse .
```

- Manchester Syntax

```
ObjectProperty: hasWife  
  SubPropertyOf: hasSpouse
```

- OWL/XML Syntax

```
<SubObjectPropertyOf>  
  <ObjectProperty IRI="hasWife"/>  
  <ObjectProperty IRI="hasSpouse"/>  
</SubObjectPropertyOf>
```

# Example

```
#registered name is a subproperty of name
ex:registeredName rdf:type owl:DatatypeProperty;
                  rdfs:subPropertyOf ex:name.
```

```
#the subproperty relationship implies that Daisy's registered
#name is also one of her names
ex:Daisy ex:registeredName "Morning Daisy Bathed in Sunshine".
```

# Top & Bottom properties

<b>owl:topObjectProperty</b>	<b>owl:topDataProperty</b>
connects all possible pairs of individuals	connects all possible individuals with all literals
<b>owl:bottomObjectProperty</b>	<b>owl:bottomDataProperty</b>
connects no pairs of individuals	does not connect any individual with a literal

# Inverse properties

- Inverse properties

```
<owl:ObjectProperty rdf:about="hasParent">  
  <owl:inverseOf rdf:resource="hasChild"/>  
</owl:ObjectProperty>
```

```
@prefix ex: <http://example.org/>.  
...  
ex:hasOwner rdf:type owl:ObjectProperty.  
ex:owns rdf:type owl:ObjectProperty.  
  
#has owner is the inverse of owns  
ex:hasOwner owl:inverseOf ex:owns.  
  
ex:Daisy ex:hasOwner ex:Ryan
```

# Disjoint properties

- Disjoint properties
  - Example: parent-child marriages cannot occur

```
<rdf:Description rdf:about="hasParent">  
  <owl:propertyDisjointWith rdf:resource="hasSpouse"/>  
</rdf:Description>
```

```
@prefix ex: <http://example.org/>.
```

```
...
```

```
ex:hasMother rdf:type owl:ObjectProperty.
```

```
ex:hasFather rdf:type owl:ObjectProperty.
```

```
ex:hasMother owl:propertyDisjointWith ex:hasFather.
```

# Disjoint properties

- May declare a group of properties as mutually disjoint with `owl:AllDisjointProperties`

```
@prefix ex: <http://example.org/>.
...
ex:hasMother rdf:type owl:ObjectProperty.
ex:hasFather rdf:type owl:ObjectProperty.

[] rdf:type owl:AllDisjointProperties;
   owl:members (
     ex:hasMother
     ex:hasFather
   ).
```



# Property chains

- Example: definition of the hasGrandparent property
  - We want hasGrandparent to connect all individuals that are linked by a chain of exactly two hasParent properties

```
<rdf:Description rdf:about="hasGrandparent">
  <owl:propertyChainAxiom rdf:parseType="Collection">
    <owl:ObjectProperty rdf:about="hasParent"/>
    <owl:ObjectProperty rdf:about="hasParent"/>
  </owl:propertyChainAxiom>
</rdf:Description>
```

# Property Chains: Example

```
ex:hasUncle rdf:type owl:ObjectProperty.  
ex:hasParent rdf:type owl:ObjectProperty.  
ex:hasBrother rdf:type owl:ObjectProperty.
```

```
# Describe the uncle relationship situation  
ex:Ryan ex:hasParent ex:Jean.  
ex:Jean rdf:type ex:Human;  
         ex:hasBrother ex:Doug.
```

```
# Define that the property chain is a  
# subproperty of the uncle relationship  
[] rdfs:subPropertyOf ex:hasUncle;  
   owl:propertyChain (  
     ex:hasParent  
     ex:hasBrother  
   ).
```

# Property Classes

PROPERTY CLASS	DEFINITION
<code>owl:SymmetricProperty</code>	$(A \text{ p } B)$ implies the statement $(B \text{ p } A)$ .
<code>owl:AsymmetricProperty</code>	$(A \text{ p } B)$ implies there is no statement $(B \text{ p } A)$ .
<code>owl:ReflexiveProperty</code>	Implies the statement $(A \text{ p } A)$ , for all A.
<code>owl:IrreflexiveProperty</code>	Implies there is no statement $(A \text{ p } A)$ , for all A.
<code>owl:TransitiveProperty</code>	$(A \text{ p } B)$ and $(B \text{ p } C)$ implies the statement $(A \text{ p } C)$ .
<code>owl:FunctionalProperty</code>	$(A \text{ p } x)$ and $(C \text{ p } y)$ implies that $x = y$ .
<code>owl:InverseFunctionalProperty</code>	$(A \text{ p } B)$ and $(C \text{ p } B)$ implies that $A = C$ .

# Symmetric, Reflexive, and Transitive Properties

- Symmetric and asymmetric properties

```
<owl:SymmetricProperty rdf:about="hasSpouse"/>  
...  
<owl:AsymmetricProperty rdf:about="hasChild"/>
```

- Reflexive and irreflexive properties

```
<owl:ReflexiveProperty rdf:about="hasRelative"/>  
...  
<owl:IrreflexiveProperty rdf:about="parentOf"/>
```

- Transitive properties

```
<owl:TransitiveProperty rdf:about="hasAncestor"/>
```

# Functional and Inverse Functional Properties

- Functional and inverse functional properties
  - Example: every individual can be linked by the hasHusband property to at most one other individual

```
<owl:FunctionalProperty rdf:about="hasHusband"/>  
...  
<owl:InverseFunctionalProperty rdf:about="hasHusband"/>
```

# Keys

- A collection of (data or object) properties can be assigned as a key to a class expression
  - This means that each named instance of the class expression is uniquely identified by the set of values which these properties attain in relation to the instance
- Example: the identification of a person by her social security number

```
<owl:Class rdf:about="Person">  
  <owl:hasKey rdf:parseType="Collection">  
    <owl:ObjectProperty rdf:about="hasSSN"/>  
  </owl:hasKey>  
</owl:Class>
```

# Multiple keys

```
# has owner and name uniquely identify canines
ex:Canine owl:hasKey (
  ex:name
  ex:hasOwner
).
```

# DataTypes

- Numeric— xsd:integer , xsd:float , xsd:real , xsd:decimal
- String— xsd:string, xsd:token, xsd:language
- Boolean— xsd:Boolean
- URI— xsd:anyUri
- XML— rdf:XMLLiteral
- Time— xsd:dateTime
- ...plus many others



# Data Type restrictions

<b>FACET</b>	<b>DESCRIPTION</b>
<code>xsd:length</code>	$N$ is the exact number of items (or characters) allowed.
<code>xsd:minLength</code>	$N$ is the minimum number of items (or characters) allowed.
<code>xsd:maxLength</code>	$N$ is the maximum number items (or characters) allowed.
<code>xsd:pattern</code>	A regular expression that defines allowed character strings.
<code>xsd:minInclusive</code>	Values must be greater than or equal to $N$ .
<code>xsd:minExclusive</code>	Values must be strictly greater than $N$ .
<code>xsd:maxInclusive</code>	Values must be less than or equal to $N$ .
<code>xsd:maxExclusive</code>	Values must be strictly less than $N$ .
<code>xsd:totalDigits</code>	The number of digits must be equal to $N$ .
<code>xsd:fractionDigits</code>	$N$ is the maximum number of decimal places allowed.

# Data Type combination

- New Data Types may be derived by existing ones through
  - owl:intersectionOf
  - owl:unionOf
  - owl:datatypeComplementOf

# Enumerated DataTypes

- Define a datatype as consisting of an enumeration of (literal) values using
  - owl:oneOf

# Object properties – negative assertion

- Example: “Mary is not Bill’s wife”
- Functional-Style Syntax

```
NegativeObjectPropertyAssertion( :hasWife :Bill :Mary )
```

- RDF/XML Syntax

```
<owl:NegativePropertyAssertion>  
  <owl:sourceIndividual rdf:resource="Bill"/>  
  <owl:assertionProperty rdf:resource="hasWife"/>  
  <owl:targetIndividual rdf:resource="Mary"/>  
</owl:NegativePropertyAssertion>
```

- Turtle Syntax

```
[ ] rdf:type owl:NegativePropertyAssertion ;  
    owl:sourceIndividual :Bill ;  
    owl:assertionProperty :hasWife ;  
    owl:targetIndividual :Mary .
```

# Object properties – negative assertion

- Example: “Mary is not Bill’s wife”
- Manchester Syntax

```
Individual: Bill  
Facts: not hasWife Mary
```

- OWL/XML Syntax

```
<NegativeObjectPropertyAssertion>  
  <ObjectProperty IRI="hasWife"/>  
  <NamedIndividual IRI="Bill"/>  
  <NamedIndividual IRI="Mary"/>  
</NegativeObjectPropertyAssertion>
```

# Examples

## Negative Object Property Assertion

```
@prefix ex: <http://example.org/>.
...
[] rdf:type owl:NegativePropertyAssertion;
   owl:sourceIndividual ex:Daisy;
   owl:assertionProperty ex:hasOwner;
   owl:targetIndividual ex:Amber.
```

## Negative DataType Property Assertion

```
@prefix ex: <http://example.org/>.
...
[] rdf:type owl:NegativePropertyAssertion;
   owl:sourceIndividual ex:Daisy;
   owl:assertionProperty ex:name;
   owl:targetValue "Rudiger".
```

# Property Restrictions

- Describe properties within the context of a specific class
- You can specify how a property is to be used when it is applied to an instance of a particular class
- A property restriction describes the class of **individuals** that meet the specified property-based conditions
- owl:Restriction

# Value Restrictions

<b>RESTRICTION</b>	<b>INTERPRETATION</b>
<code>owl:allValuesFrom</code>	For all instances, if they have the property, it must have the specified range.
<code>owl:someValuesFrom</code>	For all instances, they must have at least one occurrence of the property with the specified range.
<code>owl:hasValue</code>	For all instances, they must have an occurrence of the property with the specified value.



# Universal quantification

- Property restriction used to describe a class of individuals for which **all** related individuals must be instances of a given class
- Example: “somebody is a happy person if all their children are happy persons”
- RDF/XML Syntax

```
<owl:Class rdf:about="HappyPerson"/>
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="hasChild"/>
        <owl:allValuesFrom rdf:resource="HappyPerson"/>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

# Existential quantification

- Property restriction that defines a class as the set of all individuals that are connected via a particular property to another individual which is an instance of a certain class
- Example: “the class of Parents is the class of individuals that are linked to a Person by the hasChild property”
- RDF/XML Syntax

```
<owl:Class rdf:about="Parent">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="hasChild"/>
      <owl:someValuesFrom rdf:resource="Person"/>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

# hasValue restriction

- Limits the individuals that may be the target of the property to those identified by one specific value
- Example: the class of John's children

```
<owl:Class rdf:about="JohnsChildren">  
  <owl:equivalentClass>  
    <owl:Restriction>  
      <owl:onProperty rdf:resource="hasParent"/>  
      <owl:hasValue rdf:resource="John"/>  
    </owl:Restriction>  
  </owl:equivalentClass>  
</owl:Class>
```

# Self-restriction

- Refer to the class of all individuals that are related to themselves using that property
- Example: narcissistic persons

```
<owl:Class rdf:about="NarcisticPerson">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="loves"/>
      <owl:hasSelf rdf:datatype="&xsd:boolean"> true </owl:hasSelf>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

# Cardinality Restrictions

- Specify in very precise terms how many times a property can be used to describe an instance of a class

RESTRICTION	INTERPRETATION
<code>owl:minCardinality</code>	There must be at least $N$ properties.
<code>owl:maxCardinality</code>	There can be at most $N$ properties.
<code>owl:cardinality</code>	There are exactly $N$ properties.

# Cardinality restrictions

- Used to specify the number of individuals involved in the restriction
- Example: “John has at most four children who are

```
<rdf:Description rdf:about="John">
  <rdf:type>
    <owl:Restriction>
      <owl:maxQualifiedCardinality rdf:datatype="&xsd;nonNegativeInteger">
        4
      </owl:maxQualifiedCardinality>
      <owl:onProperty rdf:resource="hasChild"/>
      <owl:onClass rdf:resource="Parent"/>
    </owl:Restriction>
  </rdf:type>
</rdf:Description>
```

# Example

```
@prefix ex: <http://example.org/>.
...
ex:Canine rdfs:subClassOf [
    rdf:type owl:Restriction;
    owl:onProperty ex:registeredName;
    owl:maxCardinality 1
].

ex:Canine rdfs:subClassOf [
    rdf:type owl:Restriction;
    owl:onProperty ex:breed;
    owl:minCardinality 1
].
```

# Complex example

```
@prefix ex: <http://example.org/>.
...
# Large breeds must have average
# weight greater than or equal to 50 lbs
ex:LargeBreed rdf:type owl:Class;
               rdfs:subClassOf ex:Breed;
               rdfs:subClassOf [
                 rdf:type owl:Restriction;
                 owl:minCardinality 1;
                 owl:onProperty ex:averageWeight
               ];
               rdfs:subClassOf [
                 rdf:type owl:Restriction;
                 owl:onProperty ex:averageWeight;
                 owl:allValuesFrom [
                   rdf:type rdfs:Datatype;
                   owl:onDatatype xsd:real;
                   owl:withRestrictions (
                     [
                       xsd:minInclusive 50.0;
                     ]
                   )
                 ]
               ]
               ].
```



# Qualified Cardinality restrictions

- Used to specify the number of individuals involved in the restriction

RESTRICTION	INTERPRETATION
<code>owl:minQualifiedCardinality</code>	There must be at least $N$ properties that each point to an instance of $C$ .
<code>owl:maxQualifiedCardinality</code>	There can be at most $N$ properties that each point to an instance of $C$ .
<code>owl:qualifiedCardinality</code>	There are exactly $N$ properties that point to an instance of $C$ .

# Qualified Cardinality restrictions

- Example: “John has 5 children”

```
<rdf:Description rdf:about="John">
  <rdf:type>
    <owl:Restriction>
      <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
        5
      </owl:cardinality>
      <owl:onProperty rdf:resource="hasChild"/>
    </owl:Restriction>
  </rdf:type>
</rdf:Description>
```

# Enumeration of individuals

- A very straightforward way to describe a class is just to enumerate all its instances
  - “closed classes” or enumerated sets
- No individual that is not listed in the enumeration can become a member of this class.
- An individual that is included in a class membership enumeration is implicitly a member of that class

# Example

- Example: a class of birthday guests

```
<owl:Class rdf:about="MyBirthdayGuests">
  <owl:equivalentClass>
    <owl:Class>
      <owl:oneOf rdf:parseType="Collection">
        <rdf:Description rdf:about="Bill"/>
        <rdf:Description rdf:about="John"/>
        <rdf:Description rdf:about="Mary"/>
      </owl:oneOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

# Example

```
@prefix ex: <http://example.org/>.
...
ex:Daisy rdf:type ex:Canine.
...
ex:Cubby rdf:type ex:Canine.
ex:Amber rdf:type ex:Canine.
ex:London rdf:type ex:Canine.

# Each friend of Daisy's is explicitly included in this class
ex:FriendsOfDaisy rdf:type owl:Class;
    owl:oneOf (
    ex:Cubby
    ex:Amber
    ex:London
    ).
```

# Set Operators

- Set operations can be used to describe the membership of a class in terms of the extensions of other classes

<b>SET OPERATION</b>	<b>INTERPRETATION</b>
<code>owl:intersectionOf</code>	Individuals that are instances of all classes A, B, and C
<code>owl:unionOf</code>	Individuals that are instances of at least one class A, B, or C
<code>owl:complementOf</code>	Individuals that are not instances of class A

# Intersection of two classes

- Example: “Mothers are Women that are also Parents”

- Functional-Style Syntax

```
EquivalentClasses (  
  :Mother  
  ObjectIntersectionOf ( :Woman :Parent )  
)
```

- RDF/XML Syntax

```
<owl:Class rdf:about="Mother">  
  <owl:equivalentClass>  
    <owl:Class>  
      <owl:intersectionOf rdf:parseType="Collection">  
        <owl:Class rdf:about="Woman"/>  
        <owl:Class rdf:about="Parent"/>  
      </owl:intersectionOf>  
    </owl:Class>  
  </owl:equivalentClass>  
</owl:Class>
```

- Turtle Syntax

```
:Mother owl:equivalentClass [  
  rdf:type owl:Class ;  
  owl:intersectionOf ( :Woman :Parent )  
] .
```

# Intersection of two classes

- Example: “Mothers are Women that are also Parents”

- Manchester  
Syntax

```
Class: Mother  
EquivalentTo: Woman and Parent
```

- OWL/XML Syntax

```
<EquivalentClasses>  
  <Class IRI="Mother"/>  
  <ObjectIntersectionOf>  
    <Class IRI="Woman"/>  
    <Class IRI="Parent"/>  
  </ObjectIntersectionOf>  
</EquivalentClasses>
```



# Example - Intersection

```
# Example 1—intersection of
ex:PetsOfRyan rdf:type owl:Class;
               owl:intersectionOf (
                 ex:Mammal
                 [
                   rdf:type owl:Restriction;
                   owl:onProperty ex:hasOwner;
                   owl:hasValue ex:Ryan
                 ]
               ).
```

# Union of two classes

- Example: “Parents are the union of Mothers and Fathers”
- RDF/XML Syntax

```
<owl:Class rdf:about="Parent">
  <owl:equivalentClass>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="Mother"/>
        <owl:Class rdf:about="Father"/>
      </owl:unionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

# Example – Union

```
# Example 2-union of
ex:isFriendsWith rdf:type owl:ObjectProperty.

ex:FriendsOfRyan rdf:type ex:Class;
  owl:unionOf (
    [
      rdf:type owl:Class;
      owl:oneOf (
        ex:Daisy
      )
    ]
    ex:FriendsOfDaisy
  [
    rdf:type owl:Restriction;
    owl:onProperty ex:isFriendsWith;
    owl:hasValue ex:Ryan
  ]
  ).
```

# Complement of a class

- Example: “A ChildlessPerson is a Person that is not a Parent”
- RDF/XML Syntax

```
<owl:Class rdf:about="ChildlessPerson">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="Person"/>
        <owl:Class>
          <owl:complementOf rdf:resource="Parent"/>
        </owl:Class>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

# Use of complex classes in assertions

- Example: “Jack is a Person but not a Parent”
- RDF/XML Syntax

```
<rdf:Description rdf:about="Jack">
  <rdf:type>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="Person"/>
        <owl:Class>
          <owl:complementOf rdf:resource="Parent"/>
        </owl:Class>
      </owl:intersectionOf>
    </owl:Class>
  </rdf:type>
</rdf:Description>
```

# Class disjointness

- Example: “Man and women are disjoint classes”

- Functional-Style Syntax

```
DisjointClasses ( :Woman :Man )
```

- RDF/XML Syntax

```
<owl:AllDisjointClasses>  
  <owl:members rdf:parseType="Collection">  
    <owl:Class rdf:about="Woman"/>  
    <owl:Class rdf:about="Man"/>  
  </owl:members>  
</owl:AllDisjointClasses>
```

- Turtle Syntax

```
[ ] rdf:type owl:AllDisjointClasses ;  
    owl:members ( :Woman :Man ) .
```

- Manchester Syntax

```
DisjointClasses: Woman, Man
```

- OWL/XML Syntax

```
<DisjointClasses>  
  <Class IRI="Woman"/>  
  <Class IRI="Man"/>  
</DisjointClasses>
```

# Example (3 different syntaxes)

```
@prefix ex: <http://example.org/>.
...
# canine and human are disjoint classes
ex:Canine owl:disjointWith ex:Human.
```

```
ex:Animal rdf:type owl:Class.
ex:Bird rdf:type owl:Class;
    rdfs:subClassOf ex:Animal.
ex:Lizard rdf:type owl:Class;
    rdfs:subClassOf ex:Animal.
ex:Feline rdf:type owl:Class;
    rdfs:subClassOf ex:Animal.
```

```
# Each of the classes is pair-wise disjoint
_: rdf:type owl:AllDisjointClasses;
owl:members (
    ex:Bird
    ex:Lizard
    ex:Feline
    ex:Canine
).
```

```
# Each of the classes is pair-wise disjoint
# and Animal is the union of those classes
ex:Animal owl:disjointUnionOf (
    ex:Bird
    ex:Lizard
    ex:Feline
    ex:Canine
).
```

# Equivalence in OWL

- For individuals:
  - owl:sameAs
  - owl:differentFrom
    - Remember the No Unique Names Assumption!
- For classes:
  - owl:equivalentClass
- For properties
  - owl:equivalentProperty



# Equality of individuals

- Example: “James and Jim are the same individual”

- Functional-Style Syntax

```
SameIndividual( :James :Jim )
```

- RDF/XML Syntax

```
<rdf:Description rdf:about="James">  
  <owl:sameAs rdf:resource="Jim"/>  
</rdf:Description>
```

- Turtle Syntax

```
:James owl:sameAs :Jim .
```

- Manchester Syntax

```
Individual: James  
SameAs: Jim
```

- OWL/XML Syntax

```
<SameIndividual>  
  <NamedIndividual IRI="James"/>  
  <NamedIndividual IRI="Jim"/>  
</SameIndividual>
```

# Inequality of individuals

- Example: “John and Bill are not the same individual”
  - Lack of the “unique names assumption”

- Functional-Style Syntax `DifferentIndividuals( :John :Bill )`

- RDF/XML Syntax 

```
<rdf:Description rdf:about="John">
  <owl:differentFrom rdf:resource="Bill"/>
</rdf:Description>
```

- Turtle Syntax `:John owl:differentFrom :Bill .`

- Manchester Syntax `Individual: John
DifferentFrom: Bill`

- OWL/XML Syntax 

```
<DifferentIndividuals>
  <NamedIndividual IRI="John"/>
  <NamedIndividual IRI="Bill"/>
</DifferentIndividuals>
```

# Example (AllDifferent syntax)

```
@prefix ex: <http://example.org/>.
...
ex:Daisy rdf:type ex:Canine.
ex:Cubby rdf:type ex:Canine.
ex:Amber rdf:type ex:Canine.
ex:London rdf:type ex:Canine.
...
[] rdf:type owl:AllDifferent;
   owl:distinctMembers (
     ex:Daisy
     ex:Cubby
     ex:Amber
     ex:London
   ).
```

# Class equivalence

- Example: “Person and Human are semantically equivalent”

- Functional-Style Syntax

```
EquivalentClasses( :Person :Human )
```

- RDF/XML Syntax

```
<owl:Class rdf:about="Person">  
  <owl:equivalentClass rdf:resource="Human"/>  
</owl:Class>
```

- Turtle Syntax

```
:Person owl:equivalentClass :Human .
```

- Manchester Syntax

```
Class: Person  
EquivalentTo: Human
```

- OWL/XML Syntax

```
<EquivalentClasses>  
  <Class IRI="Person"/>  
  <Class IRI="Human"/>  
</EquivalentClasses>
```




# Class and Property Equivalence in OWL

- When you assert that two **classes** are equivalent, the two classes are treated as a single resource from then on
- All class **restrictions** and the class **extensions** are shared between the two classes.
- This implies that **all individuals** who are members of either class will **implicitly** become members of the other class as well.
  
- When you assert that two properties are equivalent, the property descriptions are combined.
- Every statement that uses one of the properties as a predicate implicitly exists with the other equivalent property as a predicate as well

# Tools for OWL

- Editors (<http://semanticweb.org/wiki/Editors>)
  - Most common editor: Protégé 4
  - Other tools: TopBraid Composer (\$), NeOn toolkit
  - Special purpose apps, esp. for light-weight ontologies (e.g. FOAF editors)
- Reasoners (<http://semanticweb.org/wiki/Reasoners>)
  - OWL DL: Pellet, Hermit, FaCT++, RacerPro (\$)
  - OWL EL: CEL, SHER, snorocket (\$), ELLY (extension of IRIS)
  - OWL RL: OWLIM, Jena, Oracle OWL Reasoner (\$)
  - OWL QL: Owlgres, QuOnto, Quill

# License

- This work is licensed under the Creative Commons “Attribution-NonCommercial-ShareAlike Unported (CC BY-NC-SA 3,0)” License.
- You are free:
  - to Share - to copy, distribute and transmit the work
  - to Remix - to adapt the work
- Under the following conditions:
  -  – Attribution - You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
  -  – Noncommercial - You may not use this work for commercial purposes.
  -  – Share Alike - If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.
- To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/>