

Structure of OWL Ontologies



- OWL must allow for information to be gathered from distributed sources
- This is partly done by allowing ontologies to be related, including explicitly importing information from other ontologies
- OWL makes an *open world* assumption
 - descriptions of resources are not confined to a single file or scope
 - While class *C1* may be defined originally in ontology *O1*, it can be extended in other ontologies.
 - The consequences of these additional propositions about *C1* are *monotonic*
 - New information
 - cannot retract previous information
 - can be contradictory
 - facts and entailments can only be *added*, never *deleted*

Namespaces



- XML namespace declarations enclosed in an opening `rdf:RDF` tag
 - These provide a means to unambiguously interpret identifiers and make the rest of the ontology presentation much more readable
 - A typical OWL ontology begins with a namespace declaration similar to the following

Namespace declaration (1)

```
<rdf:RDF
  xmlns      =http://www.w3.org/TR/2003/PR-owl-guide-20031215/wine#
  xml:base   =http://www.w3.org/TR/2003/PR-owl-guide-20031215/wine#
  xmlns:vin  =http://www.w3.org/TR/2003/PR-owl-guide-20031215/wine#
  xmlns:food="http://www.w3.org/TR/2003/PR-owl-guide-20031215/food#"
  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#">
```

- The first two declarations identify the namespace associated with this ontology
- They declare the *default* namespace, stating that unprefix qualified names refer to the current ontology
- The third identifies the namespace of the current ontology with the prefix `vin:`

Namespace declaration (2)

```
<rdf:RDF
  xmlns      =http://www.w3.org/TR/2003/PR-owl-guide-20031215/wine#
  xml:base   =http://www.w3.org/TR/2003/PR-owl-guide-20031215/wine#
  xmlns:vin  =http://www.w3.org/TR/2003/PR-owl-guide-20031215/wine#
  xmlns:food="http://www.w3.org/TR/2003/PR-owl-guide-20031215/food#"
  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#">
```

- The fourth identifies the namespace of the supporting food ontology with the prefix `food`:
- The fifth namespace declaration says that in this document, elements prefixed with `owl`: should be understood as referring to things drawn from the namespace called **`http://www.w3.org/2002/07/owl#`**
 - This is a conventional OWL declaration, used to introduce the OWL vocabulary

Namespace declaration (3)

```
<rdf:RDF
  xmlns      =http://www.w3.org/TR/2003/PR-owl-guide-20031215/wine#
  xml:base   =http://www.w3.org/TR/2003/PR-owl-guide-20031215/wine#
  xmlns:vin  =http://www.w3.org/TR/2003/PR-owl-guide-20031215/wine#
  xmlns:food="http://www.w3.org/TR/2003/PR-owl-guide-20031215/food#"
  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#">
```

- OWL depends on constructs defined by RDF, RDFS, and XML Schema datatypes. In this document, the `rdf:` prefix refers to things drawn from the namespace called **`http://www.w3.org/1999/02/22-rdf-syntax-ns#`**
- The next two namespace declarations make similar statements about the RDF Schema (`rdfs:`) and XML Schema datatype (`xsd:`) namespaces

Use of Entity Declarations (1)

- As an aid to writing lengthy URLs it can often be useful to provide a set of entity definitions in a document type declaration (DOCTYPE) that precedes the ontology definitions
- The names defined by the namespace declarations only have significance as parts of XML tags
- Attribute values are *not* namespace sensitive

Use of Entity Declarations (2)

- But in OWL we frequently reference ontology identifiers using attribute values. They can be written down in their fully expanded form, for example
"http://www.w3.org/TR/2003/PR-owl-guide-20031215/wine#merlot"
- Alternatively, abbreviations can be defined using an ENTITY definition, for example:

```
<!DOCTYPE rdf:RDF [  
  <ENTITY vin "http://www.w3.org/TR/2003/PR-owl-guide-20031215/wine#" >  
  <ENTITY food "http://www.w3.org/TR/2003/PR-owl-guide-20031215/food#" >  
>
```

- After this pair of ENTITY declarations, we could write the value
 - **"&vin;merlot"** and it would expand to
 - **"http://www.w3.org/TR/2003/PR-owl-guide-20031215/wine#merlot"**

Ontology Headers



```
<owl:Ontology rdf:about="">
  <rdfs:comment>An example OWL ontology</rdfs:comment>
  <owl:priorVersion
    rdf:resource="http://www.w3.org/TR/2003/CR-owl-guide-1a/wine"/>
  <owl:imports
    rdf:resource="http://www.w3.org/TR/2003/PR-owl-guide-1b/food"/>
  <rdfs:label>Wine Ontology</rdfs:label>
  ...
</owl:Ontology>
```

- owl:Ontology collects meta-data
- owl:imports includes referenced ontology
 - import might fail

Classes



- OWL classes correspond to DL concepts
 - describe common characteristics of individuals
- A few simple examples:
 - `<owl:Class rdf:ID="Winery"/>`
 - `<owl:Class rdf:ID="Region"/>`
 - `<owl:Class rdf:ID="ConsumableThing"/>`
- 'rdf:ID' defines the class name
- Within this document, "Region" can be referenced as
 - `rdf:resource="#Region"`
- `rdf:about="#Winery"/>` can be used to extend class "Winery"

Subclasses



- Subclasses introduce a subsumed-by (\sqsubseteq) relationship
- `<owl:Class rdf:ID="PotableLiquid">`
 `<rdfs:subClassOf rdf:resource="#ConsumableThing"/>`
 `...`
 `</owl:Class>`
- Expressed in DL notation as
 - `PotableLiquid \sqsubseteq ConsumableThing`
- Use of
 - `<owl:Class rdf:about="Pasta">`
 `<rdfs:subClassOf rdf:resource="#EdibleThing"/>`
 `...`
 `</owl:Class>`
 - `Pasta \sqsubseteq EdibleThing`

Individuals



- Describe members of classes
 - aka instances of classes / concepts
- Declare an individual with name "CentralCoastRegion" as a member of class "Region"
 - `<Region rdf:ID="CentralCoastRegion"/>`
- This is equivalent to
 - `<owl:Thing rdf:about="#CentralCoastRegion">
 <rdf:type rdf:resource="#Region"/>
</owl:Thing>`
- `rdf:type` is an RDF property that ties an individual to a class of which it is a member

Class vs. Individual



■ Class

- simply a name and collection of properties that describe a set of individuals

■ Individual

- member of classes

■ Subclass vs. instance

- The president of a country C is a natural candidate for an individual (no other similar individuals, unique)
- However, the president of country C can be also considered as class (representing the role, characteristics) of presidents of country C

■ Individual vs. class

- Color **red** can be an instance of class color
- Color **red** can be a class describing all red colors

Properties

- Two types of properties
 - *datatype properties*, relations between instances of classes and RDF literals and XML Schema datatypes
 - *object properties*, relations between instances of two classes
- Two object property declarations
 - sequence of OWL elements represents implicit conjunction
 - `<owl:ObjectProperty rdf:ID="madeFromGrape">`
 - `<rdfs:domain rdf:resource="#Wine"/>`
 - `<rdfs:range rdf:resource="#WineGrape"/>``</owl:ObjectProperty>`
 - $\exists \text{ madeFromGrape.T} \sqsubseteq \text{Wine}$
 - $T \sqsubseteq \forall \text{ madeFromGrape.WineGrape}$
 - `<owl:ObjectProperty rdf:ID="course">`
 - `<rdfs:domain rdf:resource="#Meal"/>`
 - `<rdfs:range rdf:resource="#MealCourse"/>``</owl:ObjectProperty>`

Nested properties (sub-properties)

- ```
<owl:Class rdf:ID="WineDescriptor"/>
<owl:Class rdf:ID="WineColor">
 <rdfs:subClassOf rdf:resource="#WineDescriptor"/>
 ...
</owl:Class>
```
- ```
<owl:ObjectProperty rdf:ID="hasWineDescriptor">
  <rdfs:domain rdf:resource="#Wine"/>
  <rdfs:range rdf:resource="#WineDescriptor"/>
</owl:ObjectProperty>
```
- ```
<owl:ObjectProperty rdf:ID="hasColor">
 <rdfs:subPropertyOf rdf:resource="#hasWineDescriptor"/>
 <rdfs:range rdf:resource="#WineColor"/>
 ...
</owl:ObjectProperty>
```

  - $\text{hasColor} \sqsubseteq \text{hasWineDescriptor}$
  - $T \sqsubseteq \forall \text{hasColor.WineColor}$

# Restrictions

```
<owl:Class rdf:ID="Wine">
 <rdfs:subClassOf rdf:resource="&food;PotableLiquid"/>
 <rdfs:subClassOf>
 <owl:Restriction>
 <owl:onProperty rdf:resource="#madeFromGrape"/>
 <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">
 1
 </owl:minCardinality>
 </owl:Restriction>
 </rdfs:subClassOf>
 ...
</owl:Class>
```

$Wine \sqsubseteq Food:PotableLiquid \sqcap \exists_{\geq 1} madeFromGrape$

- Highlighted area defines an **unnamed** (or **anonymous**) class
  - represents the set of things with at least one **madeFromGrape** property
- **Wine** class definition body states that things that are wines are also members of this anonymous class
  - every individual wine must participate in at least one **madeFromGrape** relation

# Datatype properties



- Express relationships between instances of classes and RDF literals and XML Schema datatypes
- All OWL reasoners are required to support the `xsd:integer` and `xsd:string` datatypes
- Example
  - ```
<owl:DatatypeProperty rdf:ID="yearValue">  
  <rdfs:domain rdf:resource="#VintageYear"/>  
  <rdfs:range  
    rdf:resource="&xsd;positiveInteger"/>  
</owl:DatatypeProperty>
```
 - `yearValue` relates `owl:Thing` to positive integer values

Properties and individuals

- Two individuals

(SantaCruzMountainsRegion,CaliforniaRegion): locatedIn

- ```
<Region rdf:ID="SantaCruzMountainsRegion">
 <locatedIn rdf:resource="#CaliforniaRegion"/>
</Region>
```

Winery: SantaCruzMountainVineyard

- ```
<Winery rdf:ID="SantaCruzMountainVineyard"/>
```

- Define a wine as an individual

- ```
<CabernetSauvignon
rdf:ID="SantaCruzMountainVineyardCabernetSauvignon">
 <locatedIn rdf:resource="#SantaCruzMountainsRegion"/>
 <hasMaker rdf:resource="#SantaCruzMountainVineyard"/>
</CabernetSauvignon>
```

- ```
<VintageYear rdf:ID="Year1998">
  <yearValue rdf:datatype="&xsd;positiveInteger">
    1998
  </yearValue>
</VintageYear>
```

Property characteristics: Transitivity

- Transitivity: $P(x,y) \wedge P(y,z) \Rightarrow P(x,z)$
 - `<owl:ObjectProperty rdf:ID="locatedIn">`
 `<rdf:type rdf:resource="#owl:TransitiveProperty"/>`
 `</owl:ObjectProperty>`
- Example
 - `<Region rdf:ID="SantaCruzMountainsRegion">`
 `<locatedIn rdf:resource="#CaliforniaRegion"/>`
 `</Region>`
 - `<Region rdf:ID="CaliforniaRegion">`
 `<locatedIn rdf:resource="#USRegion"/>`
 `</Region>`
- SantaCruzMountainsRegion is located in USRegion

Property characteristics: Symmetry

- Symmetry: $P(x,y) \Leftrightarrow P(y,x)$
 - ```
<owl:ObjectProperty rdf:ID="adjacentRegion">
 <rdf:type rdf:resource="#owl:SymmetricProperty"/>
 <rdfs:domain rdf:resource="#Region"/>
 <rdfs:range rdf:resource="#Region"/>
</owl:ObjectProperty>
```
  - ```
<Region rdf:ID="MendocinoRegion">  
  <locatedIn rdf:resource="#CaliforniaRegion"/>  
  <adjacentRegion rdf:resource="#SonomaRegion"/>  
</Region>
```
- MendocinoRegion is adjacent to the SonomaRegion and vice-versa
- MendocinoRegion is located in the CaliforniaRegion but **not vice versa**

Property characteristics: Functional

- Functional property: $P(x,y) \wedge P(x,z) \Rightarrow y=z$
 - `<owl:Class rdf:ID="VintageYear"/>`
 - `<owl:ObjectProperty rdf:ID="hasVintageYear">`
 - `<rdf:type rdf:resource="&owl;FunctionalProperty"/>`
 - `<rdfs:domain rdf:resource="#Vintage"/>`
 - `<rdfs:range rdf:resource="#VintageYear"/>`
 - `</owl:ObjectProperty>`
 - wines have a unique vintage year
- What can be concluded from this?
 - `<owl:ObjectProperty rdf:ID="hasEvenVintageYear">`
 - `<rdfs:subPropertyOf rdf:resource="#hasVintageYear"/>`
 - `</owl:ObjectProperty>`

Property characteristics: Inverse

- Inverse property: $P(x,y) \Leftrightarrow Q(y,x)$
 - P is inverse of Q and vice-versa
 - ```
<owl:ObjectProperty rdf:ID="hasMaker">
 <rdf:type rdf:resource="#owl:FunctionalProperty"/>
</owl:ObjectProperty>
```
  - ```
<owl:ObjectProperty rdf:ID="producesWine">  
  <owl:inverseOf rdf:resource="#hasMaker"/>  
</owl:ObjectProperty>
```
 - Is property `producesWine` functional?
- Can symmetry declared with inverses?

Property characteristics: InverseFunctional

- Inverse functional property: $P(x,y) \wedge P(z,y) \Rightarrow x=z$
 - `<owl:ObjectProperty rdf:ID="hasMaker"/>`
 - `<owl:ObjectProperty rdf:ID="producesWine">`
 - `<rdf:type rdf:resource="&owl;InverseFunctionalProperty"/>`
 - `<owl:inverseOf rdf:resource="#hasMaker"/>`
 - `</owl:ObjectProperty>`
- These declarations have the same effect as the ones on the previous slide

Property restrictions: allValuesFrom

- Previously described property declarations are **global**
- Now we cover **local** restrictions
- Restrict the right-hand side of properties locally
 - ```
<owl:Class rdf:ID="Wine">
 <rdfs:subClassOf rdf:resource="&food;PotableLiquid"/>
 ...
 <rdfs:subClassOf>
 <owl:Restriction>
 <owl:onProperty rdf:resource="#hasMaker"/>
 <owl:allValuesFrom rdf:resource="#Winery"/>
 </owl:Restriction>
 </rdfs:subClassOf>
 ...
</owl:Class>
```
  - $\text{Wine} \sqsubseteq \text{Food:PotableLiquid} \sqcap \forall \text{hasMaker.Winery}$

# Property restrictions: someValuesFrom

- Require an instance for the right-hand side of properties

```
■ <owl:Class rdf:ID="Wine">
 <rdfs:subClassOf rdf:resource="&food;PotableLiquid"/>
 ...
 <rdfs:subClassOf>
 <owl:Restriction>
 <owl:onProperty rdf:resource="#hasMaker"/>
 <owl:someValuesFrom rdf:resource="#Winery"/>
 </owl:Restriction>
 </rdfs:subClassOf>
 ...
</owl:Class>
```

- $Wine \sqsubseteq Food:PotableLiquid \sqcap \exists \text{hasMaker}.Winery$

# Property restrictions: cardinalities

- Minimum cardinality for right-hand side of properties already introduced

- This allows one to specify the exact cardinality

```
<owl:Class rdf:ID="Vintage">
 <rdfs:subClassOf>
 <owl:Restriction>
 <owl:onProperty rdf:resource="#hasVintageYear"/>
 <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
 1
 </owl:cardinality>
 </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>
```

$\text{Vintage} \sqsubseteq \exists_{\geq 1} \text{madeFromGrape} \sqcap \exists_{\leq 1} \text{madeFromGrape}$

- Upper bound: `owl:maxCardinality`
- Lower bound: `owl:minCardinality`

# Property restrictions: hasValue

- allows one to specify classes based on the existence of *particular* property values
  - An individual will be a member of such a class whenever at least *one* of its property values is equal to the hasValue resource
  - ```
<owl:Class rdf:ID="Burgundy">  
  ...  
  <rdfs:subClassOf>  
    <owl:Restriction>  
      <owl:onProperty rdf:resource="#hasSugar"/>  
      <owl:hasValue rdf:resource="#Dry"/>  
    </owl:Restriction>  
  </rdfs:subClassOf>  
</owl:Class>
```

Ontology mapping (axioms)



- Ontologies need to be widely shared
- Minimize intellectual effort involved in developing an ontology by re-use
- In the best of all possible worlds they need to be composed
- For example, you might adopt
 - a date ontology from one source and
 - a physical location ontology from another and then
 - extend the notion of location to include the time period during which it holds
- If you can find an existing ontology that has already undergone extensive use and refinement, it makes sense to adopt it

Equivalence between classes

- Declare classes as equivalent

- named classes become synonyms to one another

- denote the same set of individuals (as instances)

- ```
<owl:Class rdf:ID="Wine">
 <owl:equivalentClass rdf:resource="#Wine"/>
</owl:Class>
```

- class `Wine` from wine ontology is equivalent to class `Wine` from food ontology

- Another example

- ```
<owl:Class rdf:ID="TexasThings">  
  <owl:equivalentClass>  
    <owl:Restriction>  
      <owl:onProperty rdf:resource="#locatedIn"/>  
      <owl:allValuesFrom rdf:resource="#TexasRegion"/>  
    </owl:Restriction>  
  </owl:equivalentClass>  
</owl:Class>
```

Equivalence between properties

- can be used to state that two properties have the same property extension
 - ```
<owl:ObjectProperty rdf:ID="hasCar">
 <owl:equivalentProperty rdf:resource="&#x26;#x26;hasVehicle"/>
</owl:ObjectProperty>
```
- Property equivalence is not the same as property equality
- Equivalent properties have the same "values" (i.e., the same property extension), but may have different intensional meaning (i.e., denote different concepts)

# Identity between individuals

- Declares two individuals as identical

- similar to equivalent classes

- ```
<Wine rdf:ID="MikesFavoriteWine">  
  <owl:sameAs rdf:resource="#StGenevieveTexasWhite"/>  
</Wine>
```

- Now `MikesFavoriteWine` is a synonym to `StGenevieveTexasWhite` and vice versa

- Identity could be implied by other declarations

- ```
<owl:Thing rdf:about="#BancroftChardonnay">
 <hasMaker rdf:resource="#Bancroft"/>
 <hasMaker rdf:resource="#Beringer"/>
</owl:Thing>
```

- if we assume that `hasMaker` is a functional property, it is implied that `Bancroft` must be identical to `Beringer`

# Disjointness between individuals

- Declares two individuals to be different to each other

- ```
<WineSugar rdf:ID="Dry"/>
<WineSugar rdf:ID="Sweet">
  <owl:differentFrom rdf:resource="#Dry"/>
</WineSugar>
<WineSugar rdf:ID="OffDry">
  <owl:differentFrom rdf:resource="#Dry"/>
  <owl:differentFrom rdf:resource="#Sweet"/>
</WineSugar>
```

- Declares a set of mutually distinct individuals

- ```
<owl:AllDifferent>
 <owl:distinctMembers rdf:parseType="Collection">
 <vin:WineColor rdf:about="#Red"/>
 <vin:WineColor rdf:about="#White"/>
 <vin:WineColor rdf:about="#Rose"/>
 </owl:distinctMembers>
</owl:AllDifferent>
```

# Complex classes



- Additional (set) constructors are available to form class expressions
  - disjunction or union: `owl:unionOf`
    - DL operator:  $\sqcup$
  - conjunction or intersection: `owl:intersectionOf`
    - DL operator:  $\sqcap$
  - negation or complement: `owl:complementOf`
    - DL operator:  $\neg$
  - enumerated classes: explicitly state class extensions by enumerating individuals as instances (`owl:oneOf`)
    - often used to represent a set of mutually disjoint choices
    - could be approximated as a disjunction of mutually disjoint class names

# Class intersection

WhiteWine  $\sqcap$  Wine  $\sqcap$   $\exists$  hasColor.{White}

- ```
<owl:Class rdf:ID="WhiteWine">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Wine"/>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasColor"/>
      <owl:hasValue rdf:resource="#White"/>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```
- ```
<owl:Class rdf:ID="WhiteBurgundy">
 <owl:intersectionOf rdf:parseType="Collection">
 <owl:Class rdf:about="#Burgundy" />
 <owl:Class rdf:about="#WhiteWine" />
 </owl:intersectionOf>
</owl:Class>
```

White Burgundy  $\equiv$  Burgundy  $\sqcap$  WhiteWine

# Class union

Fruit  $\sqcup$  SweetFruit  $\sqcup$  NonSweetFruit

- ```
<owl:Class rdf:ID="Fruit">  
  <owl:unionOf rdf:parseType="Collection">  
    <owl:Class rdf:about="#SweetFruit"/>  
    <owl:Class rdf:about="#NonSweetFruit"/>  
  </owl:unionOf>  
</owl:Class>
```

Fruit \sqsubseteq SweetFruit
Fruit \sqsubseteq NonSweetFruit

- This is completely different to

- ```
<owl:Class rdf:ID="Fruit">
 <rdfs:subClassOf rdf:resource="#SweetFruit"/>
 <rdfs:subClassOf rdf:resource="#NonSweetFruit"/>
</owl:Class>
```
- this implicitly defines **Fruit** as the intersection of **SweetFruit** and **NonSweetFruit**
- which is possibly the empty set (**owl:nothing**)

# Class complement

NonConsumableThing  $\neg$  ConsumableThing

- `<owl:Class rdf:ID="ConsumableThing"/>`
- `<owl:Class rdf:ID="NonConsumableThing">`  
  `<owl:complementOf rdf:resource="#ConsumableThing"/>`  
`</owl:Class>`
- `<owl:Class rdf:ID="NonFrenchWine">`  
  `<owl:intersectionOf rdf:parseType="Collection">`  
    `<owl:Class rdf:about="#Wine"/>`  
    `<owl:Class>`  
      `<owl:complementOf>`  
        `<owl:Restriction>`  
          `<owl:onProperty rdf:resource="#locatedIn"/>`  
          `<owl:hasValue rdf:resource="#FrenchRegion"/>`  
        `</owl:Restriction>`  
      `</owl:complementOf>`  
    `</owl:Class>`  
  `</owl:intersectionOf>`  
`</owl:Class>`

NonFrenchWine  $\sqcap$  Wine  $\neg \exists$  locatedIn.{FrenchRegion}

# Enumerated classes

- OWL DL provides the means to specify a class via a direct enumeration of its members
  - done using the `owl:oneOf` construct
  - this definition completely specifies the class extension
  - no other individual can be declared to belong to such a class

```
<owl:Class rdf:ID="WineColor">
 <rdfs:subClassOf rdf:resource="#WineDescriptor"/>
 <owl:oneOf rdf:parseType="Collection">
 <owl:WineColor rdf:about="#White"/>
 <owl:WineColor rdf:about="#Rose"/>
 <owl:WineColor rdf:about="#Red"/>
 </owl:oneOf>
</owl:Class>
```

WineColor  $\sqsubseteq$  WineDescriptor

WineColor = ({White}  $\sqcup$  {Rose}  $\sqcup$  {Red})

# Disjoint classes

- Guarantees that an individual that is a member of one class cannot simultaneously be an instance of a specified other class
  - ```
<owl:Class rdf:ID="Pasta">  
  <rdfs:subClassOf rdf:resource="#EdibleThing"/>  
  <owl:disjointWith rdf:resource="#Meat"/>  
  <owl:disjointWith rdf:resource="#Fowl"/>  
  <owl:disjointWith rdf:resource="#Seafood"/>  
  <owl:disjointWith rdf:resource="#Dessert"/>  
  <owl:disjointWith rdf:resource="#Fruit"/>  
</owl:Class>
```
 - Note that this only asserts that `Pasta` is disjoint from all of these other classes. It does not assert, for example, that `Meat` and `Fruit` are disjoint. In order to assert that a set of classes is mutually disjoint, there must be an `owl:disjointWith` assertion for every pair.

Datatypes and anonymous individuals

- The following individual axiom contains two anonymous individuals, namely some **Measurement** and some **Quantity**
 - ```
<Measurement>
 <observedSubject rdf:resource="#JaneDoe"/>
 <observedPhenomenon rdf:resource="#Weight"/>
 <observedValue>
 <Quantity>
 <quantityValue rdf:datatype="&xsd;float">
 59.5
 </quantityValue>
 <quantityUnit rdf:resource="#Kilogram"/>
 </Quantity>
 </observedValue>
 <timeStamp rdf:datatype="&xsd;dateTime">
 2003-01-24T09:00:08+01:00
 </timeStamp>
</Measurement>
```
  - **JaneDoe**'s measured value of the **Weight** is some quantity, which has a value of 59.5 kilogram
  - the time of measurement is January 24, 2003, eight seconds past nine in the morning, in the time zone UTC+1

# Use of datatypes

- When using datatypes, please note that even if a property is defined to have a range of a certain datatype, RDF/XML still requires that the datatype be specified each time the property is used
- An example is the declaration of a property that we used earlier in the [Measurement](#) example
  - ```
<owl:DatatypeProperty rdf:about="#timeStamp">  
  <rdfs:domain rdf:resource="#Measurement"/>  
  <rdf:range rdf:resource="&xsd;dateTime"/>  
</owl:DatatypeProperty>
```
 - ```
<Measurement>
 <timeStamp rdf:datatype="&xsd;dateTime">
 2003-01-24T09:00:08+01:00
 </timeStamp>
</Measurement>
```

# Enumerated datatypes

- The example below specifies the range of the property `tennisGameScore` to be the list of integer values {0, 15, 30, 40}
  - ```
<owl:DatatypeProperty rdf:ID="tennisGameScore">
  <rdfs:range>
    <owl:DataRange>
      <owl:oneOf>
        <rdf:List>
          <rdf:first rdf:datatype="&xsd;integer"> 0
        </rdf:first>
        <rdf:rest>
          <rdf:List>
            <rdf:first rdf:datatype="&xsd;integer">15
          </rdf:first>
          <rdf:rest>
            <rdf:List>
              <rdf:first rdf:datatype="&xsd;integer">30
            </rdf:first>
            <rdf:rest>
              <rdf:List>
                <rdf:first rdf:datatype="&xsd;integer">40
              </rdf:first>
              <rdf:rest rdf:resource="&rdf:nil"/>
            </rdf:List>
          </rdf:rest>
        </rdf:List>
      </owl:oneOf>
    </owl:DataRange>
  </rdfs:range>
</owl:DatatypeProperty>
```

OWL Lite RDF schema features



■ Classes

- built-in: owl:Thing (\top) and owl:Nothing (\perp)

■ RDF(S)

- subclasses
- (sub)properties
 - objects
 - datatypes
 - domain
 - range

■ Individuals

- class membership
- role membership for pairs

OWL Lite equality, inequality, properties



- Equivalent
 - classes
 - properties
 - individuals
- Different individuals
- Property characteristics
 - inverse
 - transitive
 - symmetric
 - functional
 - inverse functional
- Property restrictions
 - universal
 - existential

More OWL Lite restrictions



■ Cardinalities

- minimum only 0 or 1
- maximum only 0 or 1
- exactly only 0 or 1

■ Class expressions

- only intersection
- no union or negation

OWL DL adds to OWL Lite...



- Enumerated classes (oneOf)
- Property values can be individuals
- Classes
 - disjointness
 - boolean combination (and, or, not)
- No restriction on numbers in cardinalities
- Complex classes formed by arbitrary expressions

OWL Full adds to OWL DL...



- No type separation
 - class could be
 - individual
 - property
 - property could be
 - individual
 - class
- Object and datatype properties not necessarily disjoint