Protégé-OWL Tutorial

Session 1: Primitive Classes

Nick Drummond

This session

- ► Review: OWL Basics
- ► Intro: Protégé-OWL
- ► Interface: Creating Classes
- ► Tools: The Reasoner
- ► Concept: Disjointness
- ► Interface: Creating Properties
- ► Concept: Describing Classes
- ► Interface: Creating Restrictions

Review of OWL (30 secs)

OWL…

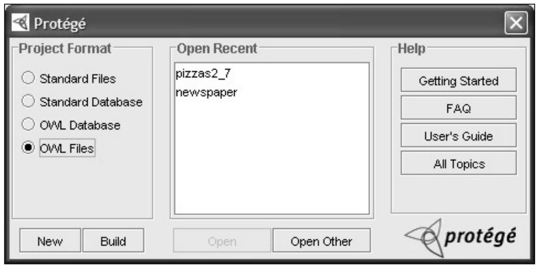
- ► is a W3C standard – Web Ontology Language
- ► comes in 3 flavours (lite, DL and full)
  - ► we are using OWL DL (Description Logic)
  - ► DL = decidable fragment of First Order Logic (FOL)
- ► is generally found in XML/RDF syntax
- ► is therefore not much fun to write by hand

So, we have tools to help us

Starting Protégé-OWL

Run Protégé from Start Menu

1. *Select "OWL Files"*
2. *Select "New"*



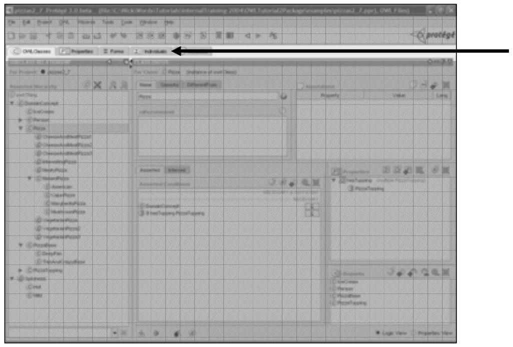
Project Format

○ Standard Files
○ Standard Database
○ OWL Database
● OWL Files

Open Recent

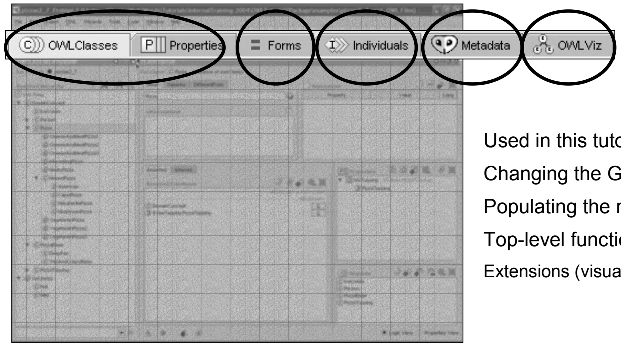
pizzas2_7
newspaper

Protégé OWL plugin



Protégé tabs

Protégé OWL plugin: Tabs



Used in this tutorial

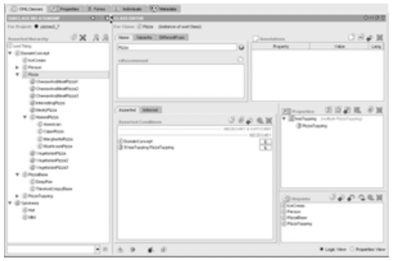
Changing the GUI

Populating the model

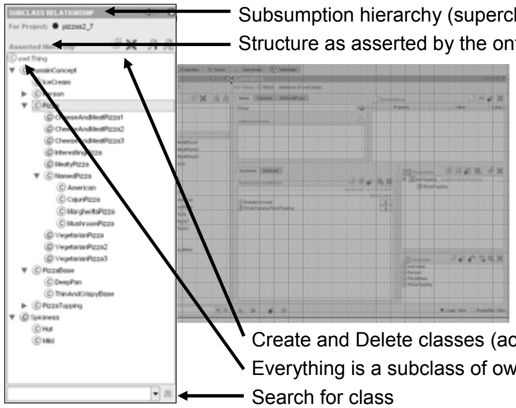
Top-level functionality

Extensions (visualisation)

Classes Tab



Classes Tab: Asserted Class Hierarchy



Subsumption hierarchy (superclass/subclass)

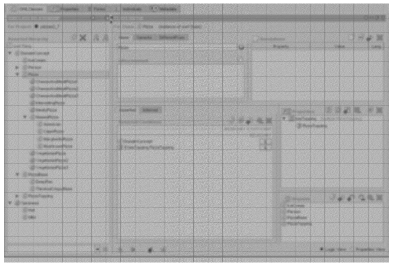
Structure as asserted by the ontology engineer

Create and Delete classes (actually subclasses!!)

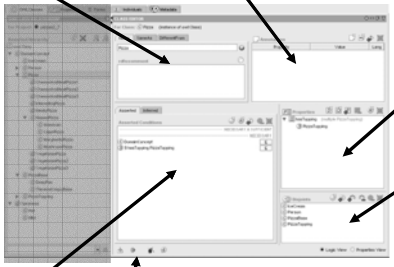
Everything is a subclass of owl:Thing

Search for class

Classes Tab: Class Editor



Classes Tab: Class Editor



Class annotations (for class metadata)

Class name and documentation

Properties “available” to Class

Disjoints widget

Conditions Widget

Class-specific tools (find usage etc)

Create a Class Hierarchy

Start with your empty ontology

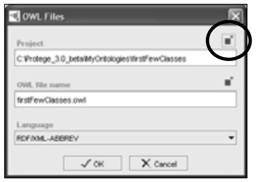
1. *Click the “Create Class” button*
   (this is above the class hierarchy)
   A new class will be created as a subclass of **owl:Thing**
2. *Type in a new name “DomainConcept” over the default*
   (return updates the hierarchy)
3. *Create another class called “Pizza” using the same method*
   You will notice that **Pizza** *has been created as a subclass of* **DomainConcept** *as this was the class selected when the button was pressed. You can also right-click any class and select “Create Class”*
4. *Create two more subclasses of* **DomainConcept** *“PizzaTopping” and “PizzaBase”.*
   Any mistakes, use the “Delete Class” button next to “Create Class”
5. *Create subclasses of* **PizzaTopping: CheeseTopping, VegetableTopping and MeatTopping**

Save Your Work

OWL = easy to make mistakes – save regularly

1. *Select File → Save*
   A dialog (as shown) will pop up
2. *Select a file using a file selector by clicking the button on the top right*
   You will notice that 2 files are created
   .pprj – the project file
   - *this just stores information about the GUI and the workspace*

   .owl – the OWL file
   - *this is where your ontology is stored in RDF/OWL format*
3. *Select OK*



OWL Files

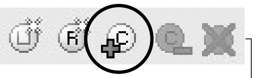
Project: C:\Protege_3.0_beta\MyOntologies\firstFewClasses

OWL file name: firstFewClasses.owl

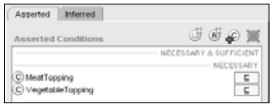
Language: RDF/XML-ABBREV

Create an odd PizzaTopping

Start with your existing ontology

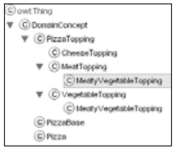


1. *Create a subclass of* **VegetableTopping** *called "MeatyVegetableTopping"*
   You will notice that the Conditions Widget has **VegetableTopping** *listed – this means it is an asserted superclass of* **MeatyVegetableTopping**



MeatTopping ⊑
VegetableTopping ⊑

2. *Add* **MeatTopping** *as another parent of* **MeatyVegetableTopping** *using the "Add Named Class" button on the conditions widget*
   MeatyVegetableTopping *can now be seen underneath both parents in the asserted class hierarchy*
   We have asserted that **MeatyVegetableTopping** *has 2 parents*



```
owl:Thing
▼ DomainConcept
  ▼ PizzaTopping
      CheeseTopping
    ▼ MeatTopping
        MeatyVegetableTopping
    ▼ VegetableTopping
        MeatyVegetableTopping
    PizzaBase
    Pizza
```

Reasoning

- ► We've just created a class that doesn't really make sense – what is a Meaty Vegetable Topping?
- ► We'd like to be able to check the logical consistency of our model
- ► Later we'd also like to make automatic inferences about the subsumption hierarchy. A process known as classifying
  - ► ie Moving classes around in the hierarchy based on their logical definition
- ► Generic software capable of these tasks are known as reasoners (although you may hear them being referred to as Classifiers)
- ► RACER is a reasoner

Running Racer

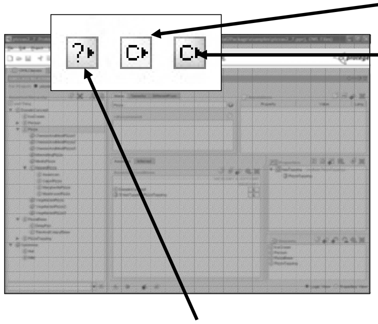
1. *Run racer.exe from wherever it was installed*

   A cmd window will open and two "service enabled" messages will appear in the ouput

   Racer is now ready for use as an http server using a standard interface called DIG

NB. Alternative DIG reasoners like FaCT can also be used

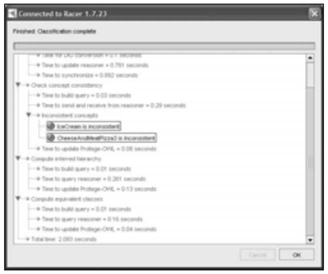
Accessing the Reasoner



Classify taxonomy (and check consistency)

Compute inferred types (for individuals)

Just check consistency (for efficiency)

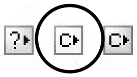


Finished: Classification complete

```
    Time for DIG conversion = 0.1 seconds
    Time to update reasoner = 0.761 seconds
    Time to synchronize = 0.892 seconds
▼ Check concept consistency
    Time to build query = 0.03 seconds
    Time to send and receive from reasoner = 0.29 seconds
  ▼ Inconsistent concepts
      IceCream is inconsistent
      CheeseAndMeatPizza3 is inconsistent
    Time to update Protege-OWL = 0.08 seconds
▼ Compute inferred hierarchy
    Time to build query = 0.01 seconds
    Time to query reasoner = 0.261 seconds
    Time to update Protege-OWL = 0.13 seconds
▼ Compute equivalent classes
    Time to build query = 0.01 seconds
    Time to query reasoner = 0.16 seconds
    Time to update Protege-OWL = 0.04 seconds
  Total time: 2.093 seconds
```

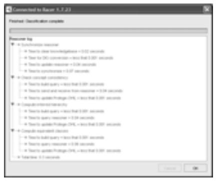
Reasoning about our Pizzas

Start with your existing ontology



1. *Classify your ontology*
   We could just use the "Check Consistency" button but we'll get into the habit of doing a full classification as we'll be doing this later

   The reasoner dialog will pop up while the reasoner works



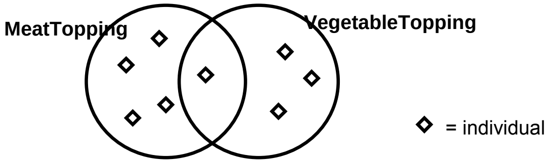
2. *When the reasoner has finished, press OK*
   You will see an inferred hierarchy appear, which will show any movement of classes in the hierarchy

   If the reasoner has inferred anything about our model, this is reported in the reasoner dialog and in a seperate results window.

   Not much appears to have happened – why has the reasoner not picked up on this odd class?

Disjointness

► OWL assumes that classes overlap

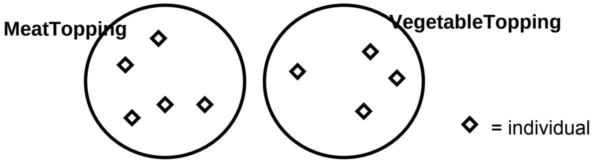


► This means an individual could be both a **MeatTopping** and a **VegetableTopping** at the same time

► We want to state this is not the case

Disjointness

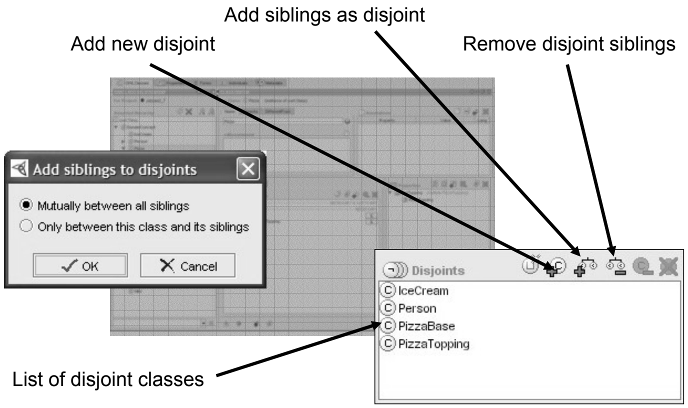
► If we state that classes are disjoint



► This means an individual cannot be both a **MeatTopping** and a **VegetableTopping** at the same time

► We must do this explicitly in the interface

ClassesTab: Disjoints Widget



Add siblings as disjoint

Add new disjoint

Remove disjoint siblings

Add siblings to disjoints

● Mutually between all siblings

○ Only between this class and its siblings

✓ OK ✗ Cancel

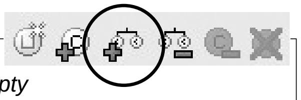
Disjoints

- IceCream
- Person
- PizzaBase
- PizzaTopping

List of disjoint classes

Make Classes Disjoint

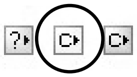
Close the inferred hierarchy



1. *Select the* **Pizza** class
   The disjoints widget is currently empty
2. *Click the "Add all siblings…" button*
   The "Add siblings to disjoints dialog pops up
3. *Select the "Mutually between all siblings" option and OK*
   PizzaTopping *and* **PizzaBase** *appear in the disjoints widget*
4. *Select the* **PizzaTopping** *class*
   Pizza *and* **PizzaBase** *are already in the disjoints widget*
   Note that the same applies for **PizzaBase**
5. *Add disjoints between subclasses of* **PizzaTopping**

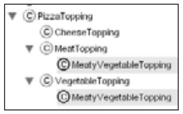
Running the Reasoner again

Start with your existing ontology

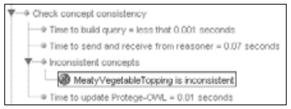


1. *Classify your ontology*

You will see **MeatyVegetableTopping** *highlighted in red in both hierarchies – this highlights that a class is inconsistent*



You will also see messages in both the reasoner dialog and a results window appear at the bottom of the screen which describes the results of the reasoner



```
Check concept consistency
    Time to build query = less that 0.001 seconds
    Time to send and receive from reasoner = 0.07 seconds
    Inconsistent concepts
        MeatyVegetableTopping is inconsistent
    Time to update Protege-OWL = 0.01 seconds
```

MeatyVegetableTopping *turns out to be inconsistent*

Why is MeatyVegetableTopping inconsistent?

- We are asserting that a **MeatyVegetableTopping** is a subclass of two classes we have stated are disjoint
- The disjoint means nothing can be a **MeatTopping** and a **VegetableTopping** at the same time
- This means that the class of **MeatyVegetableTopping** can never contain any individuals
- The class is therefore inconsistent
- This is what we expect!

- It can be useful to create classes we expect to be inconsistent to "test" your model – often we refer to these classes as "probes" – generally it is a good idea to document them as such to avoid later confusion

Create More Sensible PizzaToppings

Start with your existing ontology

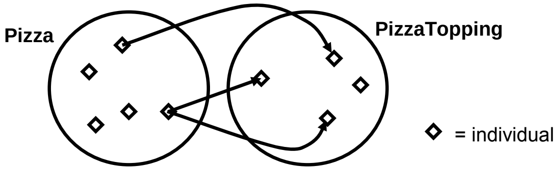
1. *Create subclasses of* **CheeseTopping:**
   MozzarellaTopping, **ParmesanTopping**
2. *Make these subclasses all disjoint from one another*
3. *Create subclasses of* **VegetableTopping** *and make them disjoint:*
   TomatoTopping, **MushroomTopping**
4. *Save to another file using File → Save As…*

What have we got?

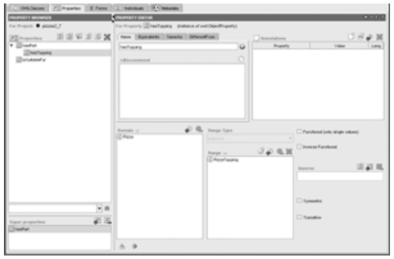
- ► We've created a tree of disjoint classes
- ► Disjoints are inherited down the tree
  eg something that is a **TomatoTopping** cannot be a **Pizza** because its superclass, **PizzaTopping**, is disjoint from **Pizza**
- ► You should now be able to select every class (except **DomainConcept**) and see its siblings in the disjoints widget

What are we missing?

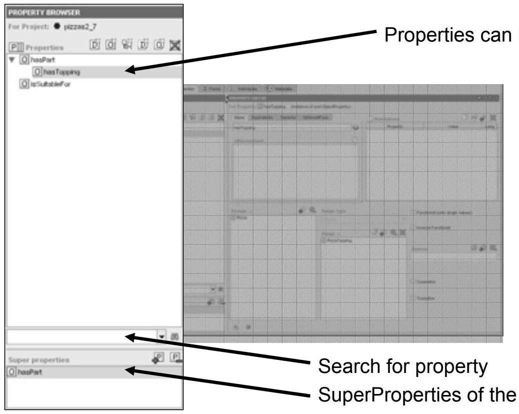
- ► This is not a semantically rich model
- ► Apart from "is kind of" and "is not kind of", we currently don't have any other information of interest
- ► We want to say more about **Pizza** individuals, such as their relationship with other individuals
- ► We can do this with properties



Properties Tab



Properties Tab: Property Browser

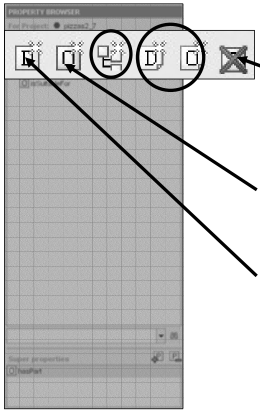


Properties can be in a hierarchy

Search for property

SuperProperties of the current selected

Properties Tab: Property Browser



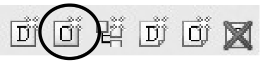
Delete Property

New Object Property:
Associates an individual to another individual

not used today:
- New Datatype Property (String, int etc)
- New Annotation Properties for metadata
- New SubProperty – ie create "under" the current selection

Create a Property

Start with your existing ontology



1. *Switch to the Properties tab*
   There are currently no properties, so the list is blank
2. *Create a new Object property using the button in the property browser*
3. *Call the new Property "hasTopping"*
4. *Create another Object Property called "hasBase"*
5. *Save under a new filename*

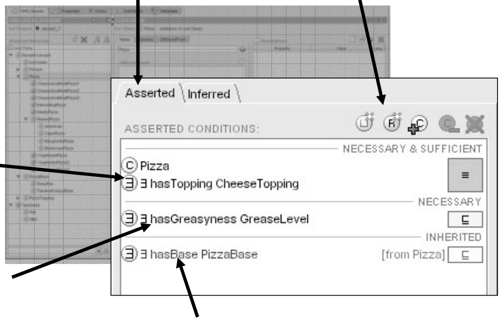
Associating Properties with Classes

- ► We now have two properties we want to use to describe **Pizza** individuals.
- ► To do this, we must go back to the **Pizza** class and add some further information
- ► This comes in the form of Restrictions (which are a type of Condition)

ClassesTab: Conditions Widget

Conditions asserted by the ontology engineer

Add different types of condition



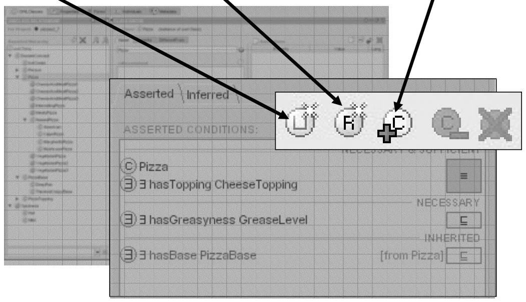
Pizza
∃ hasTopping CheeseTopping
∃ hasGreasyness GreaseLevel
∃ hasBase PizzaBase [from Pizza]

Definition of the class (later)

Description of the class

Conditions inherited from superclasses

Conditions Types



Create Restriction (next)

Create Class Expression

Add Named Superclass

Asserted \ Inferred

ASSERTED CONDITIONS:

Pizza

∃ hasTopping CheeseTopping

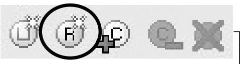
NECESSARY

∃ hasGreasyness GreaseLevel

INHERITED

∃ hasBase PizzaBase [from Pizza]

Create a Restriction

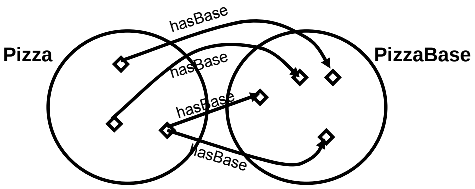


Start with your existing ontology

1. *Switch to the OWL Classes tab*
2. *Select* **Pizza**
   Notice that the conditions widget only contains one item, **DomainConcept** *with a Class icon.*
   Superclasses show up in the conditions widget in this way
3. *Click the “Create Restriction” button*
   A dialog pops up that we will investigate in a minute
4. *Select “hasBase” from the Restricted Property pane*
5. *Leave the Restriction type as “someValuesFrom”*
6. *Type “PizzaBase” in the Filler expression editor*
7. *Click OK*
   A restriction has been added to the Conditions widget

What does this mean?

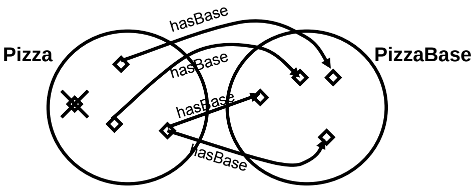
- We have created a restriction: ∃ hasBase **PizzaBase** on Class **Pizza** as a necessary condition



- “If an individual is a member of this class, it is necessary that it has at least one hasBase relationship with an individual from the class **PizzaBase**”
- “Every individual of the **Pizza** class must have at least one base from the class **PizzaBase**”

What does this mean?

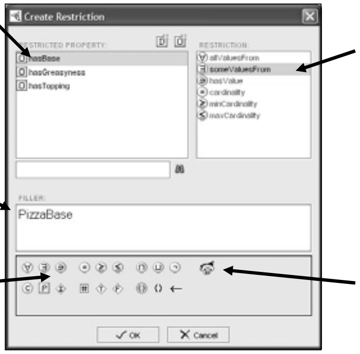
- We have created a restriction: ∃ hasBase **PizzaBase** on Class **Pizza** as a necessary condition



- “There can be no individual, that is a member of this class, that does not have at least one hasBase relationship with an individual from the class **PizzaBase**”

Restrictions Popup

Restricted Property



Create Restriction

RESTRICTED PROPERTY:
- hasBase
- hasGreasyness
- hasTopping

RESTRICTION:
- allValuesFrom
- someValuesFrom
- hasValue
- cardinality
- minCardinality
- maxCardinality

FILLER: PizzaBase

Restriction Type

Filler Expression

Expression Construct Palette

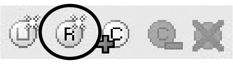
Syntax check

Restriction Types

∃	Existential, someValuesFrom	“Some”, “At least one”
∀	Universal, allValuesFrom	“Only”
∍	hasValue	“equals x”
=	Cardinality	“Exactly n”
≤	Max Cardinality	“At most n”
≥	Min Cardinality	“At least n”

Another Existential Restriction

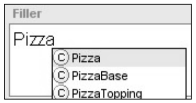
Start with your existing ontology



1. *Make sure* **Pizza** *is selected*
2. *Create a new Existential (SomeValuesFrom) Restriction with the hasTopping property and a filler of* **PizzaTopping**

   When entering the filler, you have 2 shortcut methods rather than typing the entire classname:

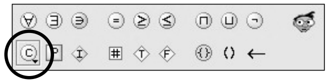
   1) enter a partial name and use Tab to autocomplete



Filler

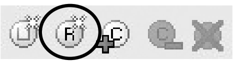
Pizza
- Pizza
- PizzaBase
- PizzaTopping

   2) use the select Class button on the editor palette



Create a Universal Restriction

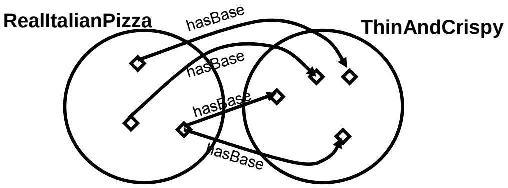
Start with your existing ontology



1. *Create 2 disjoint subclasses of* **PizzaBase** *called “ThinAndCrispy” and “DeepPan”*
2. *Create a subclass of* **Pizza** *called “RealItalianPizza”*
3. *Create a new Universal (AllValuesFrom) Restriction on* **RealItalianPizza** *with the hasBase property and a filler of* **ThinAndCrispy**

What does this mean?

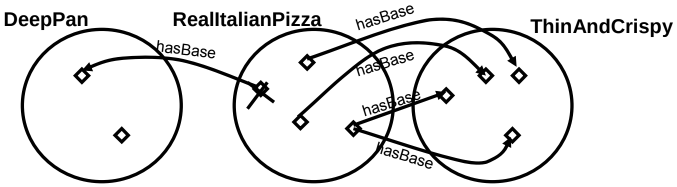
- ► We have created a restriction: ∀ hasBase **ThinAndCrispy** on Class **RealItalianPizza** as a necessary condition



- ► "If an individual is a member of this class, it is necessary that it must only have a hasBase relationship with an individual from the class **ThinAndCrispy**"

What does this mean?

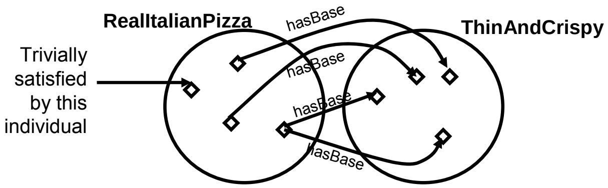
- ► We have created a restriction: ∀ hasBase **ThinAndCrispy** on Class **RealItalianPizza** as a necessary condition



- ► "No individual of the **RealItalianPizza** class can have a base from a class other than **ThinAndCrispy**"

Universal Warning: Trivial Satisfaction

- ► If we had not already inherited: ∃ hasBase **PizzaBase** from Class **Pizza** the following could hold

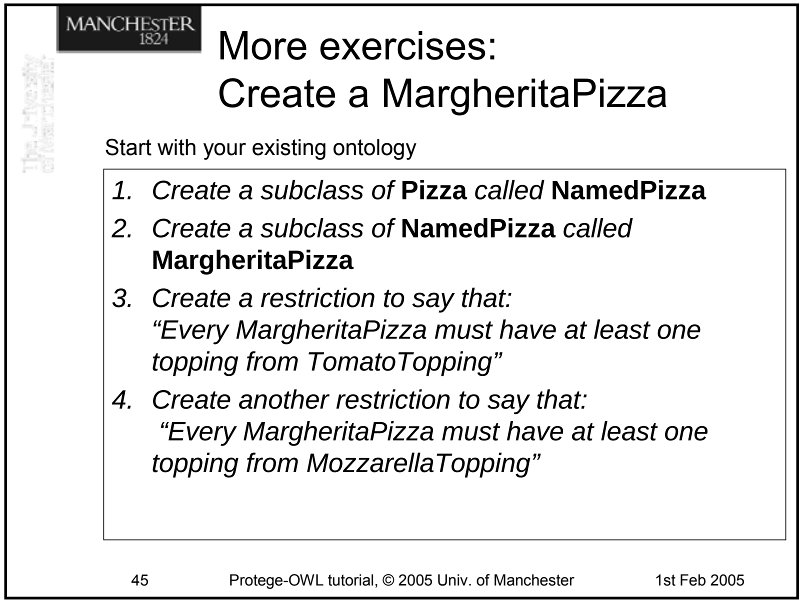


- ► "If an individual is a member of this class, it is necessary that it must only have a hasBase relationship with an individual from the class **ThinAndCrispy,** or no hasBase relationship at all"
- ► Universal Restrictions by themselves do not state "at least one"

Summary

You should now be able to:

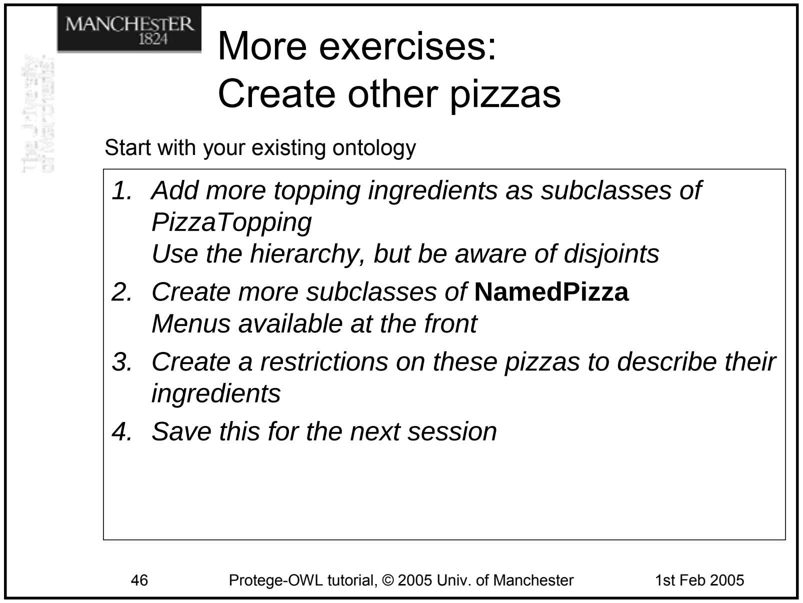
- ► identify components of the Protégé-OWL Interface
- ► create Primitive Classes
- ► create Properties
- ► create some basic Restrictions on a Class using Existential and Universal qualifiers



More exercises: Create a MargheritaPizza

Start with your existing ontology

1. *Create a subclass of* **Pizza** *called* **NamedPizza**
2. *Create a subclass of* **NamedPizza** *called* **MargheritaPizza**
3. *Create a restriction to say that:*
   "Every MargheritaPizza must have at least one topping from TomatoTopping"
4. *Create another restriction to say that:*
   "Every MargheritaPizza must have at least one topping from MozzarellaTopping"



More exercises: Create other pizzas

Start with your existing ontology

1. *Add more topping ingredients as subclasses of PizzaTopping*
   Use the hierarchy, but be aware of disjoints
2. *Create more subclasses of* **NamedPizza**
   Menus available at the front
3. *Create a restrictions on these pizzas to describe their ingredients*
4. *Save this for the next session*