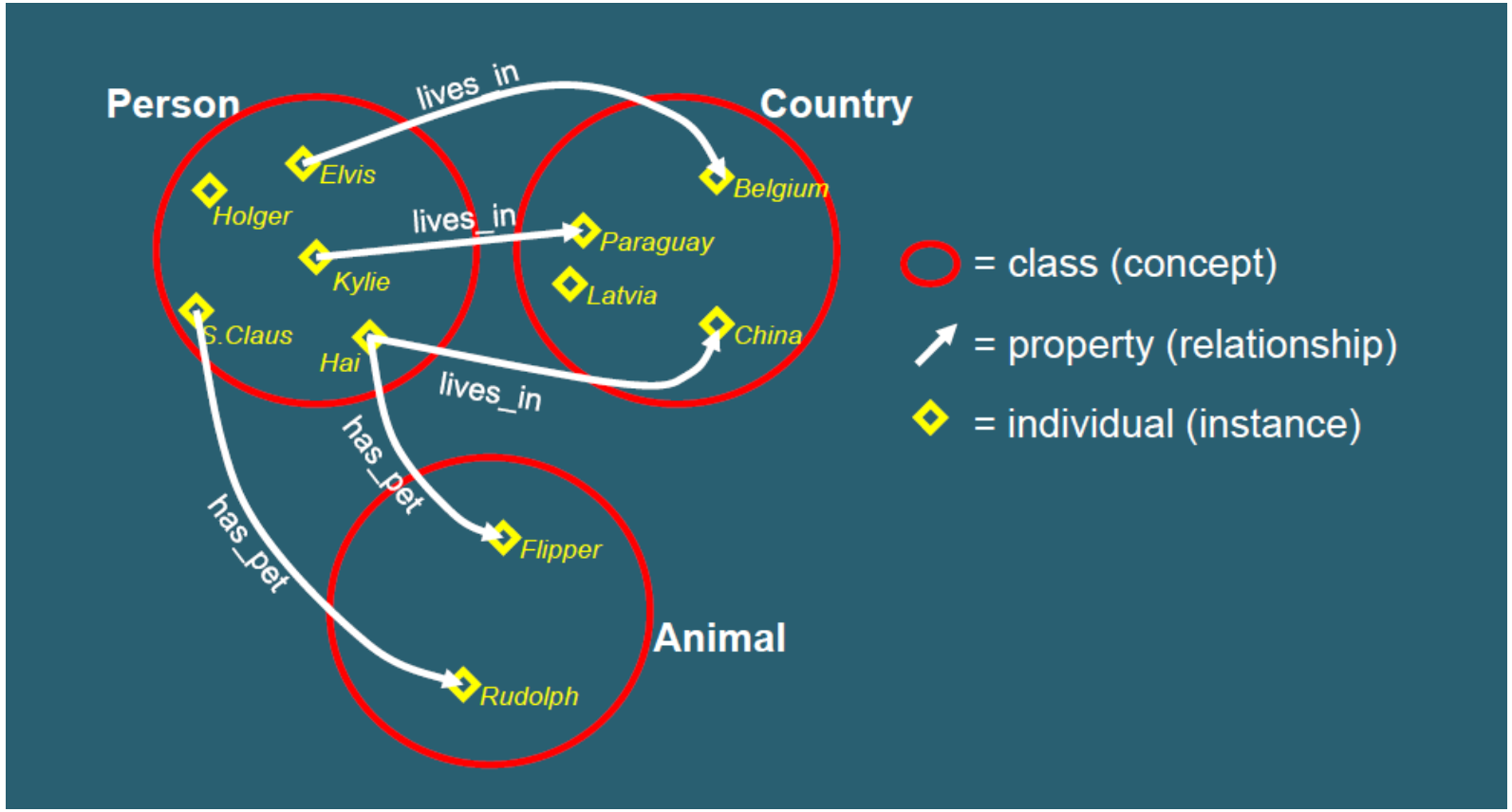
Introduction to protégé

OWL review

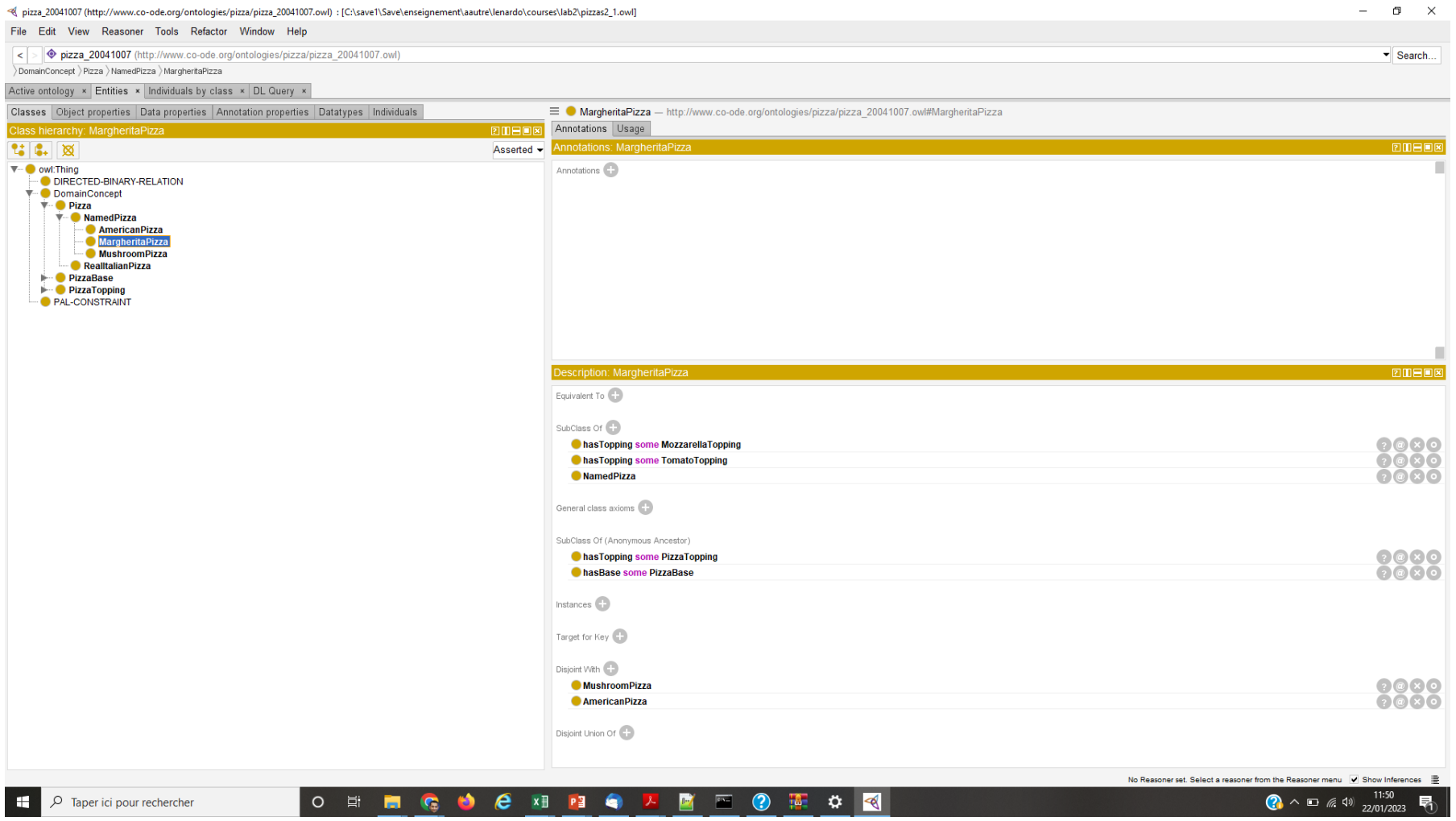
- OWL…
  - is a W3C standard –Web Ontology Language
  - is generally found in XML/RDF syntax
  - is therefore not much fun to write by handw

- So, we have tools to help us

OWL constructs



Protégé OWL plug-ins



Exercise 1 : Create a new OWL ontology

- **1. Click the “Add a SubClass” 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. Document your class using the rdfs:comment field in Annotations tab**
- **4. 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 “Add subClass”*
- **5. Create two more subclasses of DomainConcept called “PizzaTopping” and “PizzaBase”.**
  - *Any mistakes use the “Class” button “Create Class”*

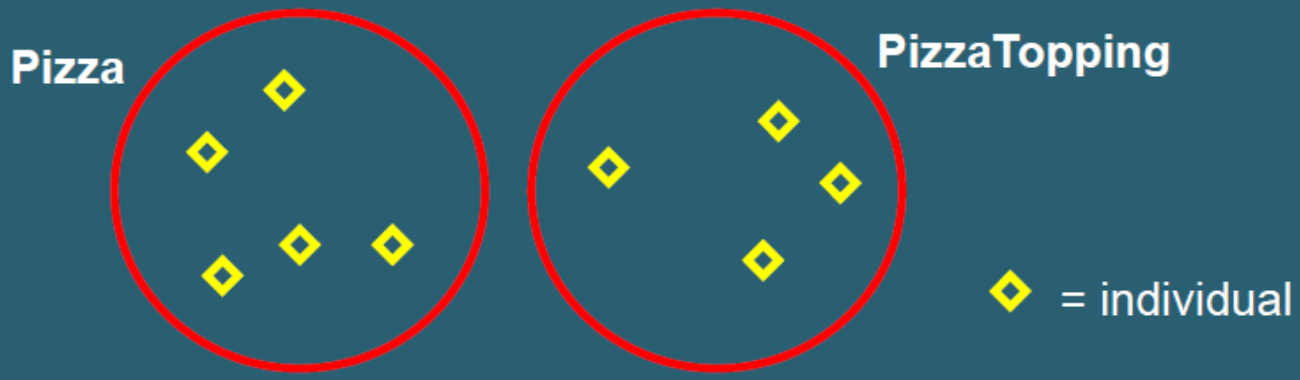
Disjointness

- OWL assumes that classes overlap

- This means an individual could be **both** a **Pizza** and a **PizzaTopping** 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 **Pizza** and a **PizzaTopping** at the same time
- We must do this **explicitly** in the interface

Make class disjoint

- ***1. Select the* Pizza class**
  - *You will notice that the Disjoint with is empty*
- **2. Use "*Disjoint with* " or "*Disjoint Union Of*" button in Pizza Class to add**
  - **PizzaTopping** *and* **PizzaBase**
- **4. Select the PizzaTopping class**
  - **Pizza** *and* **PizzaBase** *are already in the disjoints widget*
- **5. Note that the same applies for PizzaBase**

Save your work

- **1. Select File** => **Save Project**
  - *A dialog (as shown) will pop up*
  - *Save you file in OWL format*
- **2. Select a file using a file selector by clicking the button on the top right**
  - *A files is created : .owl – the OWL file*
  - *this is where your ontology is stored in RDF/OWL format*

Create Pizza toppings

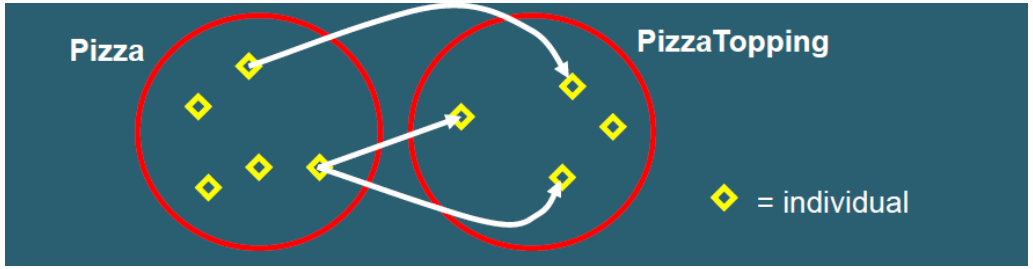
- **1. Create subclasses of PizzaTopping:**
  - CheeseTopping
  - VegetableTopping
  - MeatTopping
- **2. Make these subclasses all disjoint from one another**
  - *(remember to chose "Mutually between all siblings" when prompted)*
- **3. Create subclasses of CheeseTopping:**
  - MozzarellaTopping, ParmesanTopping
- **4. Make these subclasses all disjoint from one another**
- **5. Create subclasses of VegetableTopping and make them disjoint:**
  - TomatoTopping, MushroomTopping
- **6. Save**

What have you got?

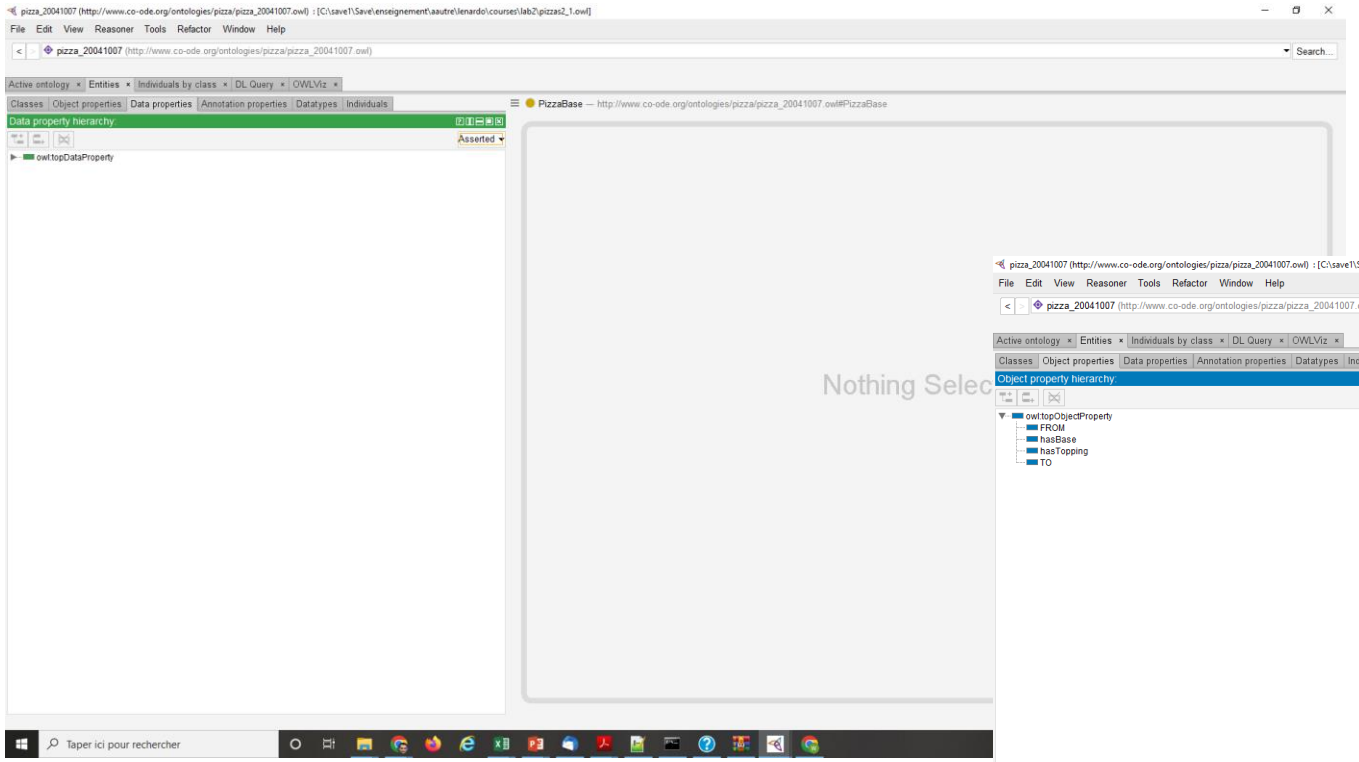
- We've created a tree of disjoint classes
- Disjoints are inherited down the tree eg something that is a **TomatoTopping** cannot be a **Pizza** because its superclass, **PizzaTopping**, is disjoint from **Pizza**
- You should now be able to select every class (except **DomainConcept**) and see its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

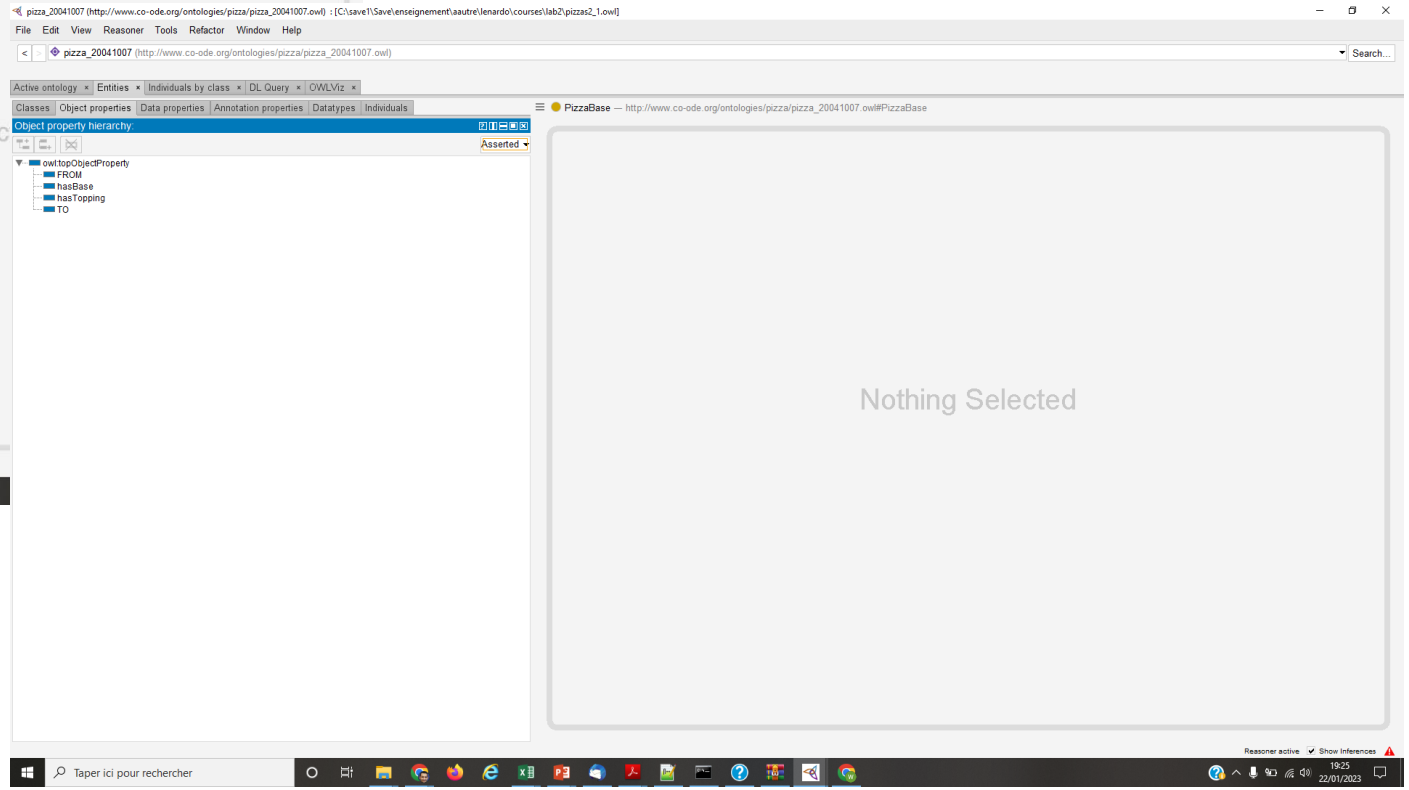
What are missing?

- 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



Object and Data Properties tabs





Create a prperty

- *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*

Associating Properties with classes

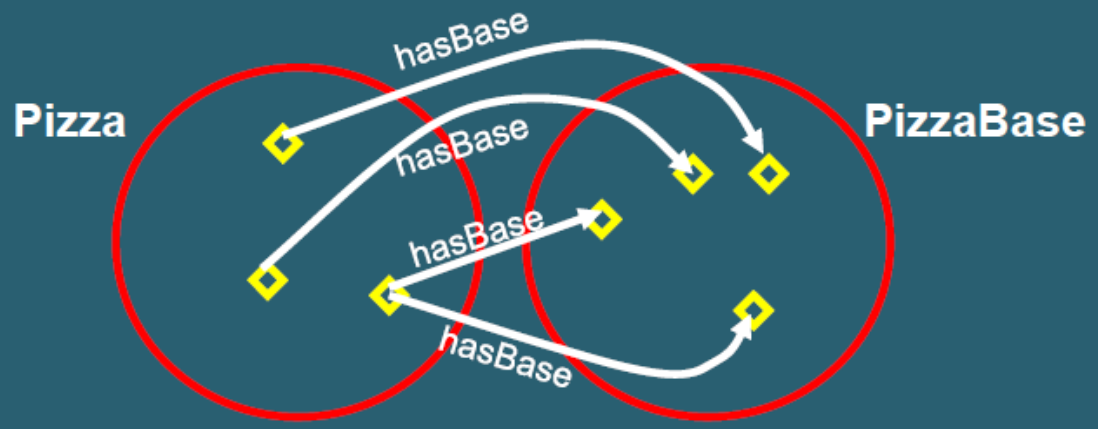
- We now have two properties we want to use to describe **Pizza** individuals.
- To do this, we must go back to the **Pizza** class and add some further information
- This comes in the form of Restrictions (which are a type of Condition)

Create a restriction

- ***1. Switch to the OWL Classes tab***
- ***2. Select* Pizza**
- ***3. Click the "SubClass of" button***
  - *A dialog pops up that we will investigate in a minute*
- ***4. Select "hasBase" from the* *Object restriction creator***
- ***5. Leave the Restriction type as "someValuesFrom"***
- ***6. Select "PizzaBase" in the restriction Filler***
- ***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**"

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”**

Exercise: another existential restriction

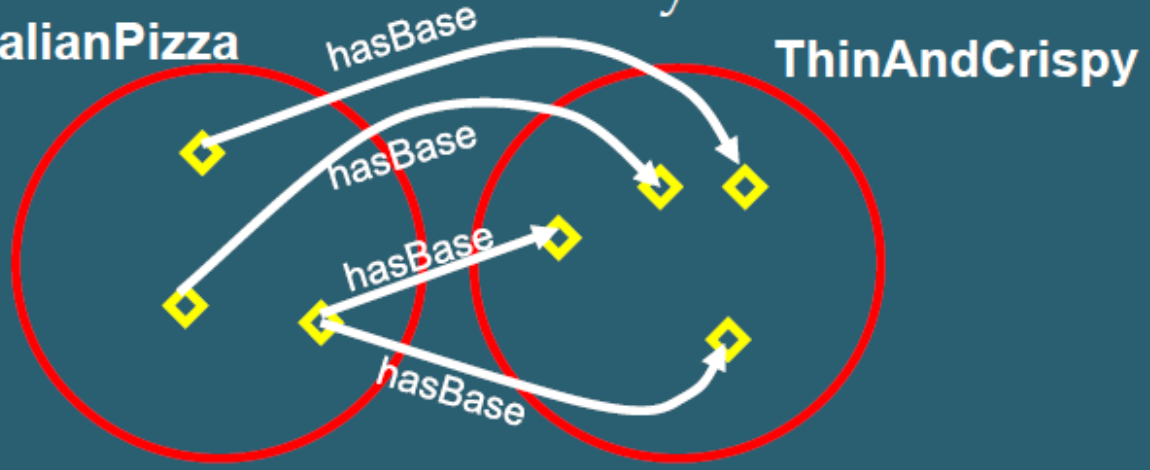
- *1. Make sure* Pizza *is selected*
- *2. Create a new Existential (SomeValuesFrom) Restriction with the hasTopping property and a filler of* PizzaTopping

Exercise: create universal restriction

- *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"**

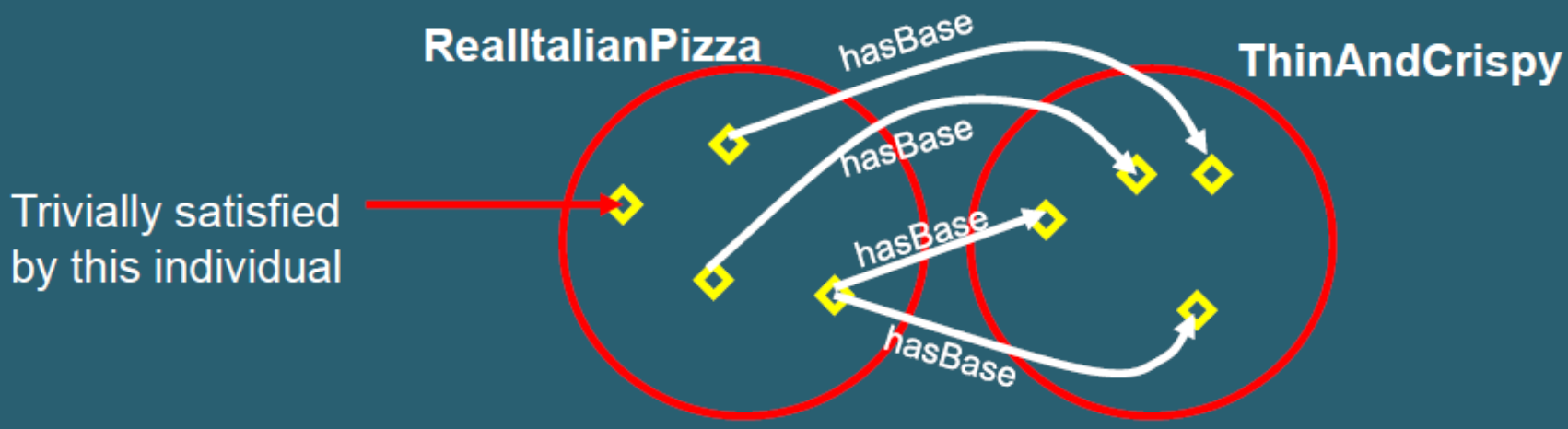
What does this mean?

- 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"
- **ie Universal Restrictions by themselves** do not **state "at least one"**

Summury

- 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

Exercise: Create a Margherita Pizza

- *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"*

Exercise: Create other Pizza

- *Extend the example by creating new types of pizzas that does not already exist. To do so:*
  - *1. Add more topping ingredients as subclasses of PizzaTopping. Use the hierarchy, but be aware of disjoints*
  - *2. Create new classes that represent the new types of pizzas.*
  - *3. Express the fact how this new class is related to other types using a disjunction constraint.*
  - *4. Create restrictions on these pizzas to describe their ingredients*

Vegetarian Pizza

- Start from the pizzas2_1.owl available on the Labs page:
  - Select File => chose pizzas2_1.owl
- Create a new pizza called “VegetarianPizza” under Pizza
  - make this disjoint from its siblings as we have been doing
- Select MargheritaPizza.
  - you will notice that it only has a single parent, NamedPizza
- Add VegetarianPizza as a new parent
  - notice that MargheritaPizza now occurs in 2 places in the asserted hierarchy we have asserted that MargheritaPizza has 2 parents

Reaseoning

- We’d like to be able to check the logical consistency of our model
- 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)
- Pellet is a reasone to be used (make sure that you have added the associated plug-in in Protégé)

Reasoning about our Pizzas

- *Classify your ontology*
  - *You will see an inferred hierarchy appear, which will show any movement of classes in the hierarchy*
  - we need to use the *reaseaner menu* to start Pellet Reasoner
  - *You will also see a results window appear at the bottom of the screen which describes the results of the reasoner; if there is an error*
- *MargheritaPizza turns out to be inconsistent – why?*

Why is MargheritaPizza inconsistent?

- We are asserting that a **MargheritaPizza** is a subclass of two classes we have stated are disjoint
- The disjoint means nothing can be a **NamedPizza** and a **VegetarianPizza** at the same time
- This means that the class of **MargheritaPizza**s can never contain any individuals
- The class is therefore inconsistent

Attempting again

- *1. Close the inferred hierarchy and classification results panel that could be found by using inferred in the menu*
- *2. Remove the disjoint between* **VegetarianPizza** *and its siblings*
  - *When prompted, choose to remove only between this class and its siblings*
- *3. Synchronize your reasoner*
  - *This should now be accepted by the reasoner with no inconsistencies*

Primitive classes

- All classes in our ontology so far are primitive
  - We describe primitive pizzas
- Primitive Class = only subclass of Conditions
- They are marked as yellow disc in the class hierarchy

Defined Classses

- Have a definition (equivalent to).
  - That is Necessary and Sufficient condition
- Are marked with equivalence log in the interface
- Classes, all of whose individuals satisfy this definition, can be inferred to be subclasses
- Reasoners can perform this inference

Describing a meaty pizza

- Start pizzas2_3 owl, *close the reasoner*
- *1. Create a subclass of* **Pizza** *called* **MeatyPizza**
  - *Don't put in disjoints or you'll get same problems as before*
  - *In general, defined classes are not disjoint*
- *2. Add a restriction to say:*

  "Every **MeatyPizza** *must have at least one meat topping"*
- *3. Classify your ontology by synchronizing the Pellet reasoner*
  - *What happens?*

Defining a MeatyPizza

- *1. move* ∃ *hasTopping* **MeatTopping** *restriction from “*Necessary*” to “*Necessary & Sufficient*”*
  - *The* **MeatyPizza** *class now turns orange, denoting that it is now a defined class*
- *2. Move* **Pizza** *Superclass from “*Necessary*” to “*Necessary & Sufficient*”*
- *3. Restart the reasoner*

How do you define a vegetarian pizza

- Define in words?
  - “a pizza with only vegetarian toppings”?
  - “a pizza with no meat (or fish) toppings”?
  - “a pizza that is not a MeatyPizza”?
- More than one way to model this

Defining a vegetarian topping

- Start with pizzas2_5.owl
- *1. Create a subclass of* **PizzaTopping** *called* **VegetarianTopping**
- *2. Click "Create New Expression" in the Conditions Widget of equivalent to Type in each of the top level* **PizzaToppings** *that are not meat or fish (ie* **DairyTopping**, **FruitTopping** *etc) and between each, type insert the union logic symbol*
- *3. Press Return when finished you have created an anonymous class described by the expression*
- *4. Classify your ontology*

Vegetarian Pizza second attempts

- *1. Select* **MargheritaPizza** *and remove* **VegetarianPizza** *from its superclasses*
- *2. Select* **VegetarianPizza** *and create a restriction to say that it "only has toppings from* **VegetarianTopping***"*
- *3. Make this a defined class by moving all conditions from "*Necessary*" to "*Necessary & Sufficient*"*
- *4. Classify your ontology*
  - *What happens?*

Open World Assumption

- The reasoner does not have enough information to classify pizzas under **VegetarianPizza**
- Typically several Existential restrictions on a single property with different fillers – like primitive pizzas
- Existential should be paraphrased by “amongst other things…”
- We need closure for the given property

Closure

- Example: **MargheritaPizza**
  - All **MargheritaPizzas** must have:
    - at least 1 topping from **MozzarellaTopping** and
    - at least 1 topping from **TomatoTopping** and
    - only toppings from **MozzarellaTopping** or **TomatoTopping**
- The last part is paraphrased into “no other toppings”
- The union closes the hasTopping property on **MargheritaPizza**

Closing Pizza descriptions

- Start with pizzas2_7.owl
- *1. Select* **MargheritaPizza**
- *2 Create a hasTopping property*
- *2. Universal Restriction on the with a filler of "**TomatoTopping U MozzarellaTopping**"*

  Remember, you can type "or" to achieve this, or you can use the expression palette
- *3. Close your other pizzas*
  - *Each time you need to create a filler with the union of all the classes used on the hasTopping property (ie all the toppings used on that pizza)*
- *4. Classify your ontology (restart your resonner*
  - *Finally, the defined class* **VegetarianPizza** *should subsume any classes that only have vegetarian toppings*

Final Exercise

- Create an OWL ontology that models the following concepts:
- 1. There should be three classes: Customer, Shop and Product.
- 2. Customer and Shop should be equipped with properties name (xsd:string) and email (xsd:string), which could be equivalent to some properties in Vcard.
- 3. Each Product should have an order number (xsd:int). An order number can be unambiguously assigned to a Product.
- 4. A Shop should have a property sells (range: Product) and a Product should have a property soldBy (range: Shop) respectively.
- 5. Instances of class Shop that sell more than 100 products should belong to a new class BigShop.
- 6. A Product must not be a Customer.
- 7. Instances that are both, Shop and Customer should belong to a class PurchaseAndSale.

    - **Note**: Import the Vcard ontology