UNIVERSITETET
I OSLO

INF1300
Introduction to databases

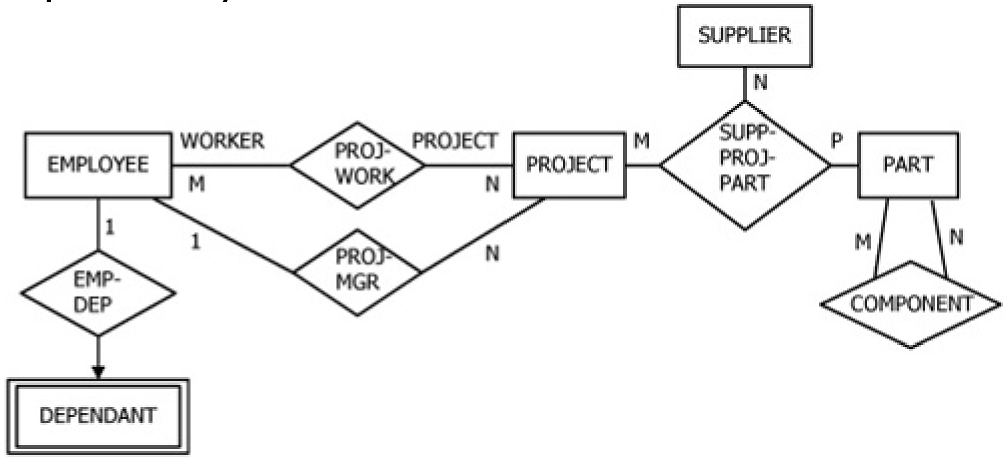
Today's topics:

- Brief intro to ER and UML
- Resource Description Framework (RDF)
- SPARQL: RDF query language

Intro to Entity–Relationship (ER) Modeling

Based on *Ch 8.1* in Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, Second Edition

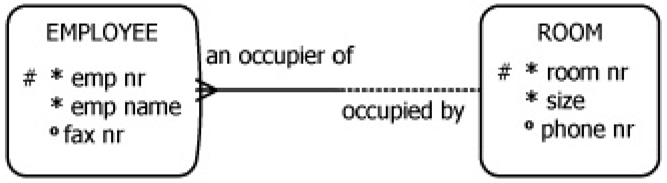
- Entity–relationship model (ER model) is a data model for describing a database in an abstract way
- ER models business domains in terms of entities that have attributes and participate in relationships
- Very popular data modeling approach for databases
- Originally proposed by Peter Chen in 1976



Barker notation

Based on *Ch 8.1* in Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, Second Edition

- Building blocks: entities, relationships, and attributes

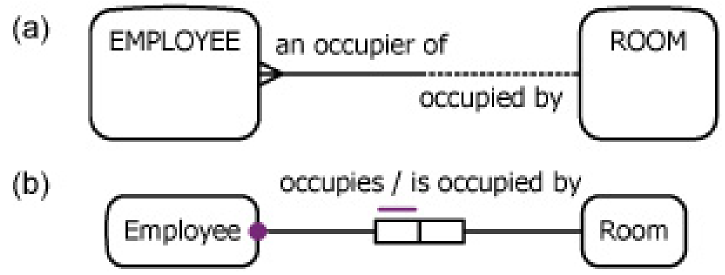


- Attributes:
  - “ # ” indicates that the attribute is, or is a component of, the primary identifier of the entity type
  - “ * ” indicates that the attribute is mandatory
  - “ ° ” indicates the attribute is optional
- Relationships are restricted to binaries
  - A solid half-line denotes a mandatory role, and a dotted half-line indicates an optional role
  - Crow’s foot notation is used for cardinality; intuitively indicates “many”, by its many “toes”; the absence of a crow’s foot intuitively indicates “one”

Barker notation (cont')

Based on *Ch 8.1* in Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, Second Edition

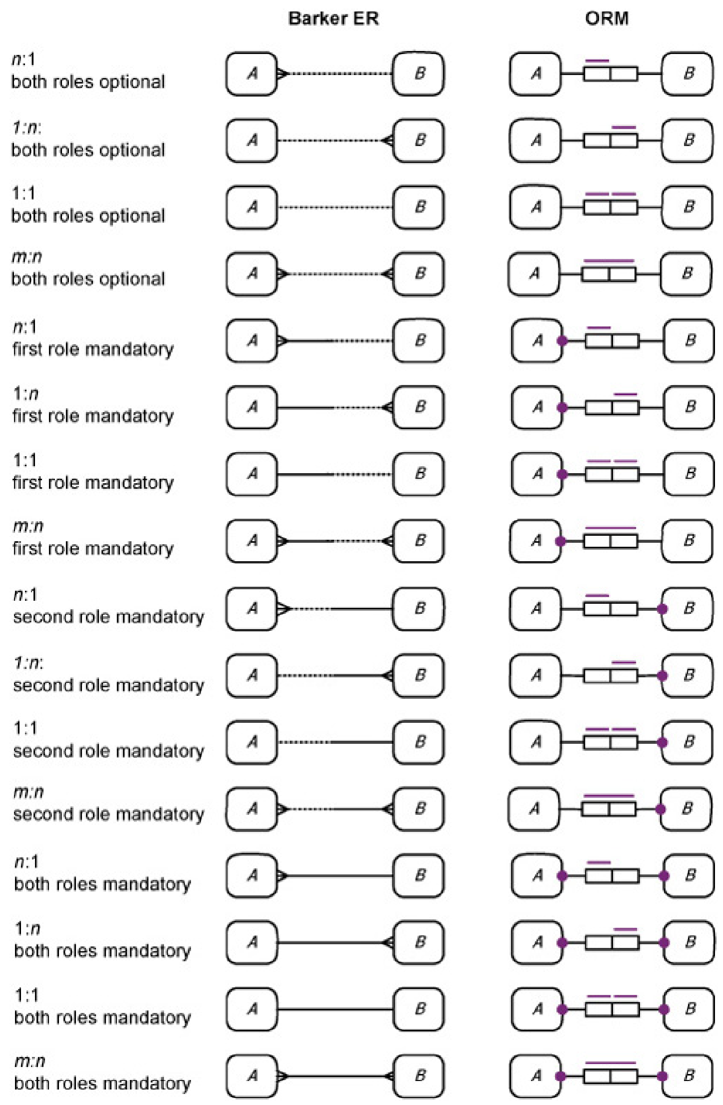
- ER diagram (a) and its equivalent to ORM (b)



- Verbalization: **Each A (must | may) be *R* (one and only one *B* | one or more** *B-plural-form)*
  - **Each** Employee **must be** an occupier of one and **only one** Room; **Each** Room **may be** occupied by **one or** more Employees.

Equivalent Barker ER and ORM diagrams

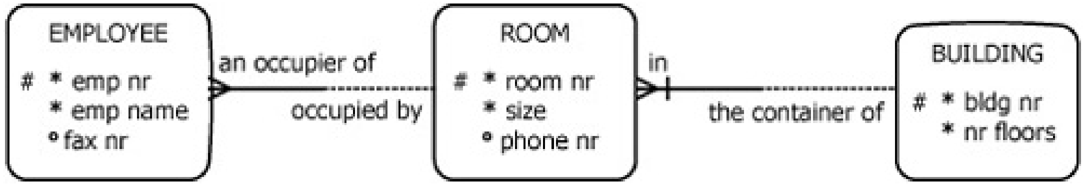
Based on *Ch 8.2* in Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, Second Edition


Barker ER
ORM
n:1 both roles optional
1:*n*: both roles optional
1:1 both roles optional
m:*n* both roles optional
n:1 first role mandatory
1:*n* first role mandatory
1:1 first role mandatory
m:*n* first role mandatory
n:1 second role mandatory
1:*n*: second role mandatory
1:1 second role mandatory
m:*n* second role mandatory
n:1 both roles mandatory
1:*n* both roles mandatory
1:1 both roles mandatory
m:*n* both roles mandatory
A
B

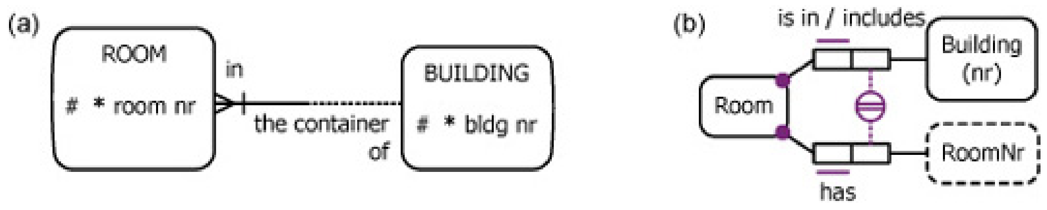
Composite identification in Barker ER

Based on *Ch 8.2* in Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, Second Edition

- A bar “|” across one end of a relationship indicates that the relationship is a component of the primary identifier for the entity type at that end.



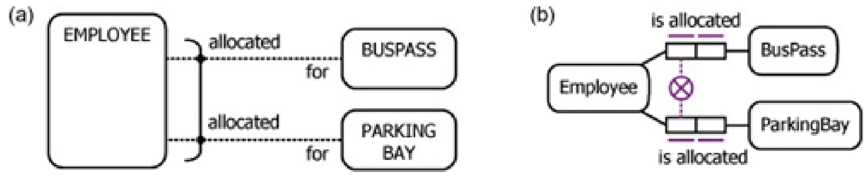
- Composite identification in (a) Barker ER and (b) ORM



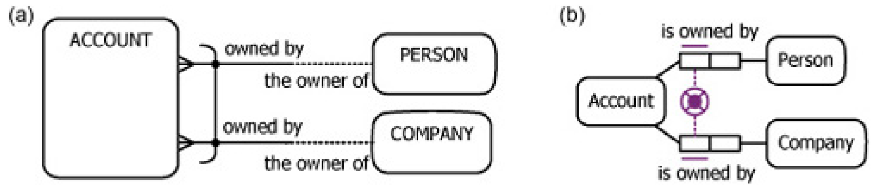
Other constraints in Barker ER

Based on *Ch 8.2* in Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, Second Edition

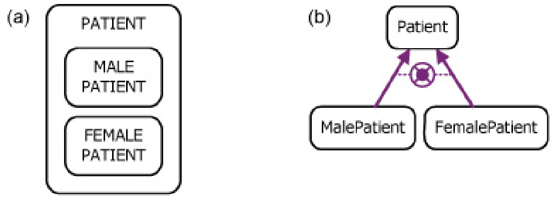
- Exclusion constraints are shown as an “exclusive arc” connected to the roles with a small dot or circle



- Mutually exclusive and disjunctively mandatory constraints uses the exclusive arc, but each role is shown as mandatory (solid line)

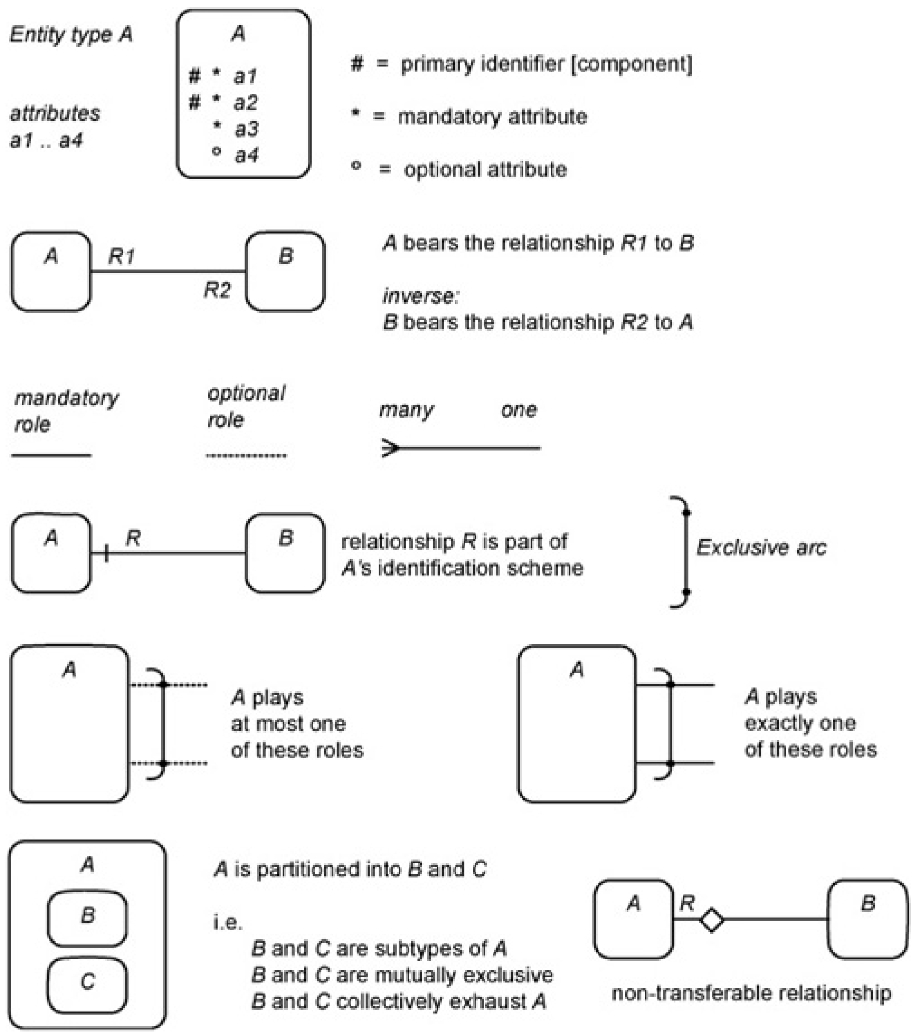


- Subtyping is depicted with a version of Euler diagrams.



Barker ER notation – summary

Based on *Ch 8.2* in Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, Second Edition

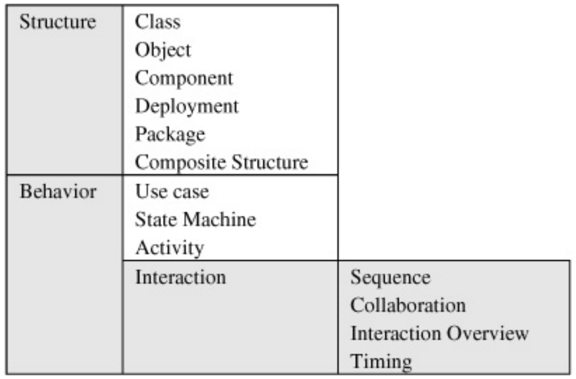


Entity type A
A
# * a1
# * a2
* a3
° a4
attributes a1 .. a4
# = primary identifier [component]
* = mandatory attribute
° = optional attribute
A
R1
R2
B
A bears the relationship R1 to B
inverse:
B bears the relationship R2 to A
mandatory role
optional role
many
one
A
R
B
relationship R is part of A's identification scheme
Exclusive arc
A
A plays at most one of these roles
A
A plays exactly one of these roles
A
B
C
A is partitioned into B and C
i.e.
B and C are subtypes of A
B and C are mutually exclusive
B and C collectively exhaust A
A
R
B
non-transferable relationship

Intro to Unified Modeling Language (UML)

Based on *Ch 9.1* in Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, Second Edition

- Mainly used for designing object-oriented program code
- De facto standard in industry for object-oriented software design
- UML notation is a set of languages



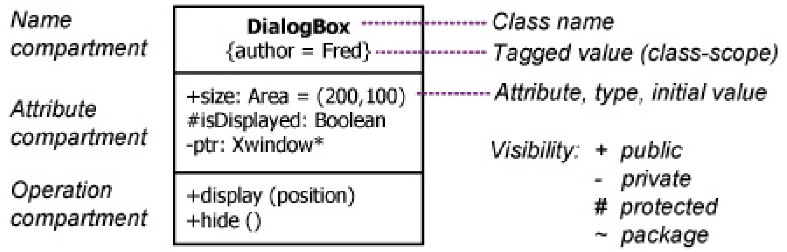
Structure	Class Object Component Deployment Package Composite Structure	
Behavior	Use case State Machine Activity	
	Interaction	Sequence Collaboration Interaction Overview Timing

- Class diagrams are used for the data schema
  - Like ER, UML uses attributes

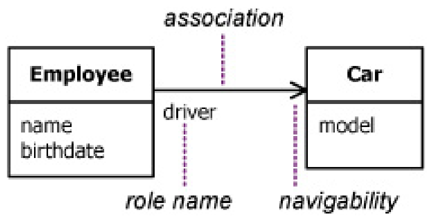
UML Class diagrams

Based on *Ch 9.2* in Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, Second Edition

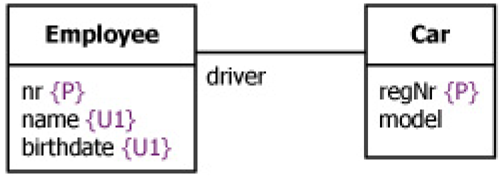
- Class structure – three compartments



- Associations - depicted by a line between the classes; the open arrow is a navigability setting (issues related to performance, not conceptual issues about the business domain)



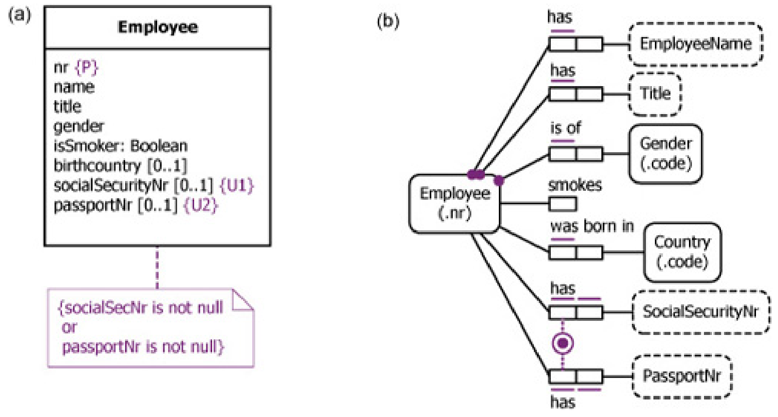
- Class diagrams can be used for conceptual analysis.
  - But no identification schemas are provided for the classes
  - Nonstandard identification schemas:
    "{P}" for preferred reference and "{U*n*}" for uniqueness *(n* > 0)



Attributes in UML Class diagrams

Based on *Ch 9.3* in Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, Second Edition

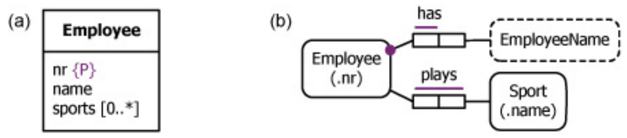
- In UML, attributes are mandatory and single valued by default
- UML attributes (a) depicted as ORM relationship types (b)



- Multiplicities:

Multipl.	Abbrev.	Meaning	Note
0..1		0 or 1 (at most one)	
0..*	*	0 to many (zero or more)	
1..1	1	exactly 1	Assumed by default
1..*		1 or more (at least 1)	
n..*		n or more (at leastn)	n≥0
n..m		at least n and at most m	m > n≥0

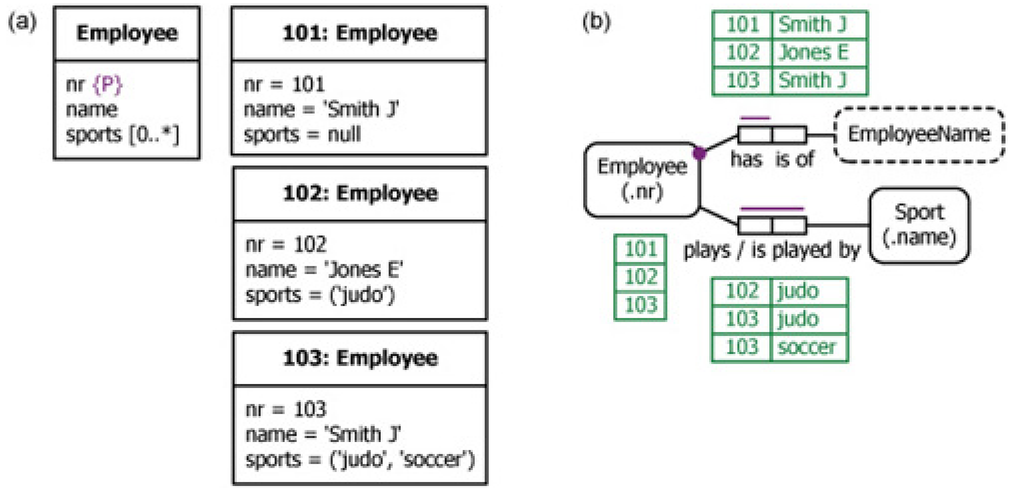
Multivalued UML sports attribute depicted as (b) ORM *m:n* fact type:



UML object diagrams

Based on *Ch 9.3* in Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, Second Edition

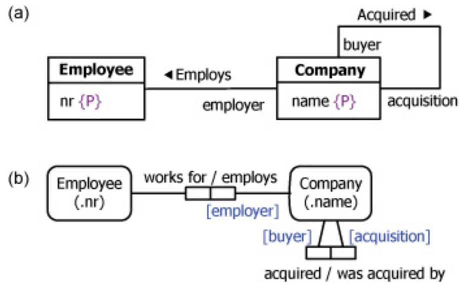
- Object diagrams: Class diagrams in which each object is shown as a separate instance of a class, with data values supplied for its attributes
  - They easily become unwieldy if multiple instances for more complex cases are displayed
- Populated models in (a) UML and (b) ORM:



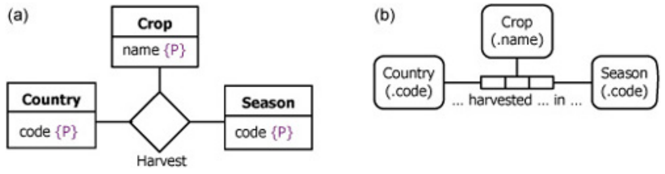
Associations

Based on *Ch 9.3* in Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, Second Edition

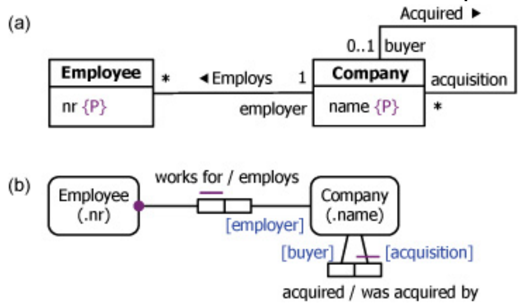
- Binary associations are depicted as lines between classes
  - Association names are optional, but role names are mandatory
  - If two or more roles are played by the same class, the roles must be given different names to distinguish them



- Ternary and higher arity associations in UML are depicted as a diamond connected by lines to the classes

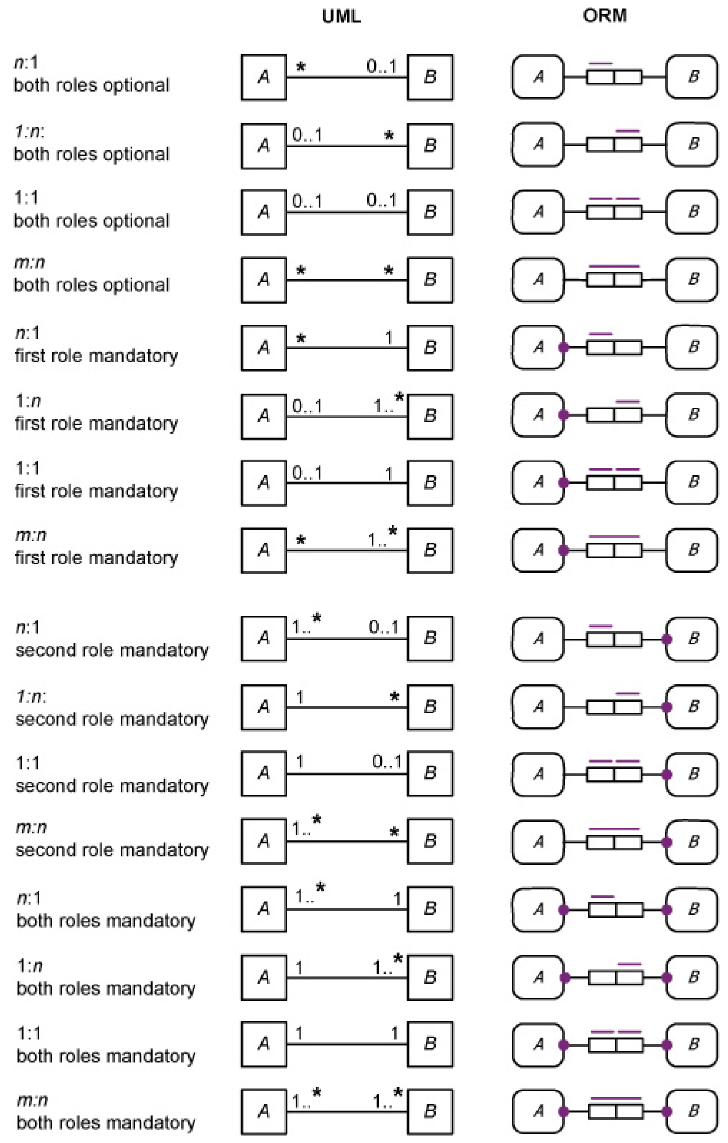


- Multiplicity constraints on associations similar to those on attributes; multiplicities are written next to the relevant roles



Equivalent constraint patterns in UML and ORM

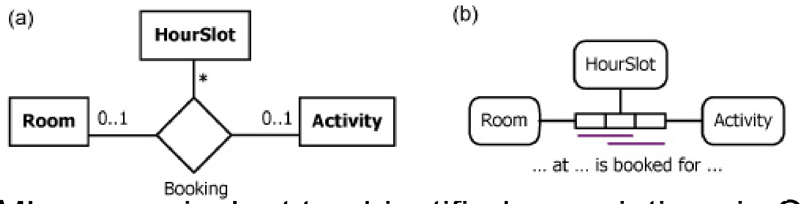
Based on *Ch 9.3* in Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, Second Edition



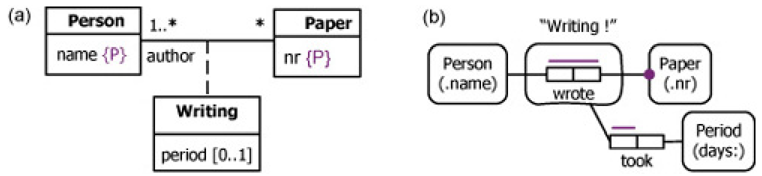
n-ary associations, association classes, and qualified associations in UML

Based on *Ch 9.4* in Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, Second Edition

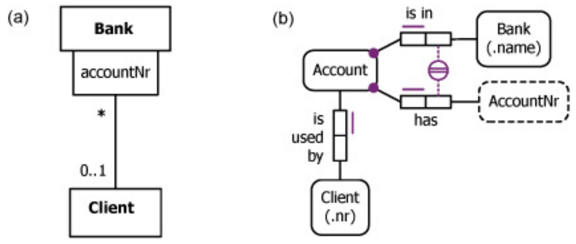
- A multiplicity constraint on a role of an n-ary association constrains the population of the other roles combined



- Association classes in UML are equivalent to objectified associations in ORM



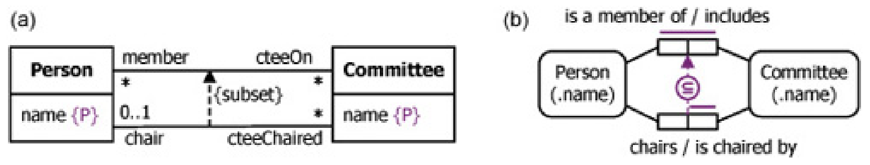
- Cases where ORM uses an external uniqueness constraint for coreferencing can be modeled in UML using qualified associations



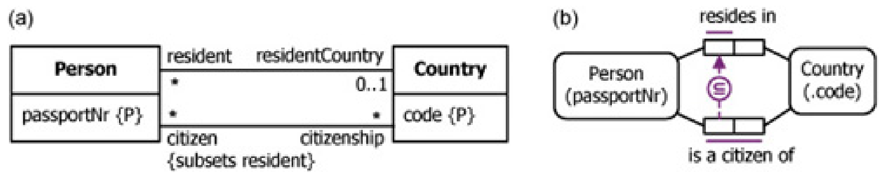
Set-Comparison Constraints in UML

Based on *Ch 9.5* in Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, Second Edition

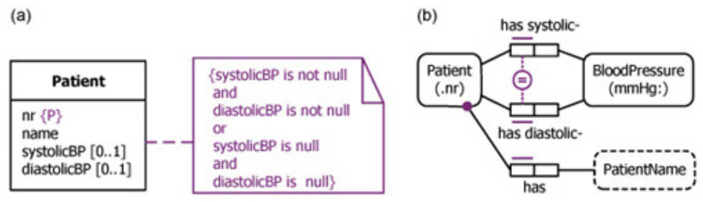
- Subset constraints can be specified between whole associations by attaching the constraint label "{subset}" next to a dashed arrow between the associations



- Subsets properties are used to indicate that the population of an attribute or association role must be a subset of the population of another compatible attribute or association role respectively
  - However many subset constraint cases in ORM that cannot be represented graphically as a subset constraint in UML



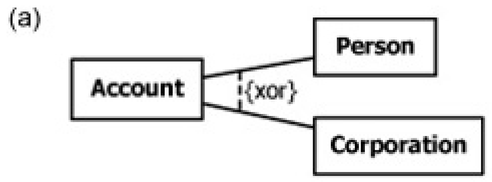
- UML has no graphic notation for equality constraints
  - May be specified as textual constraints in notes

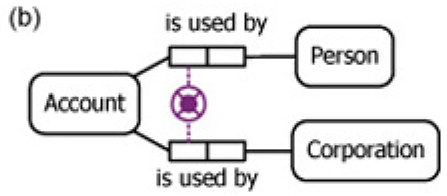


Set-Comparison Constraints in UML (cont')

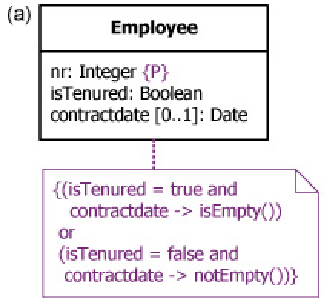
Based on *Ch 9.5* in Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, Second Edition

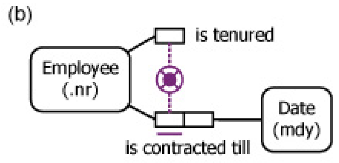
- Exclusion constraints: not directly supported in UML, but does include an *exclusive-or constraint* to indicate that each instance of a class plays *exactly one* association role from a specified set of alternatives





- UML has no symbols for exclusion or inclusive-or constraints
  - UML has no graphic notation for exclusion between attributes, or between attributes and association roles
  - An exclusion constraint in such cases may often be captured as a textual constraint

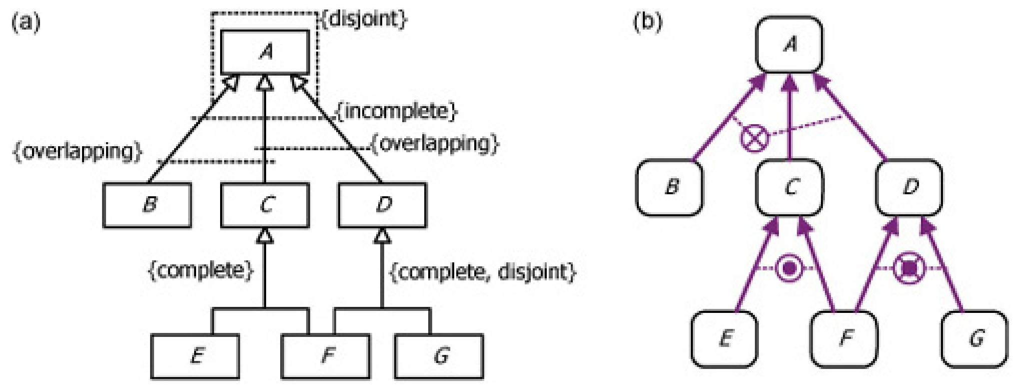




Subtyping constraints in UML

Based on *Ch 9.5* in Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, Second Edition

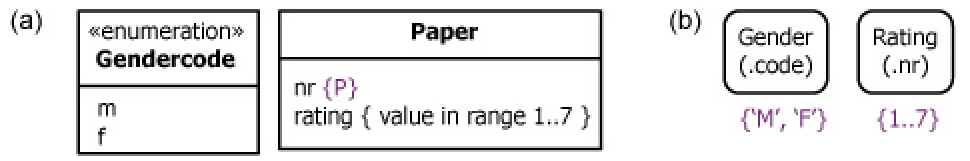
- Single and multiple inheritance is allowed both in UML and ORM
  - Subtype inherits all the roles of its supertypes in ORM
  - A subclass inherits all the attributes, associations, and operations/methods of its supertype(s) in UML
- UML and ORM both display subtyping using directed acyclic graphs



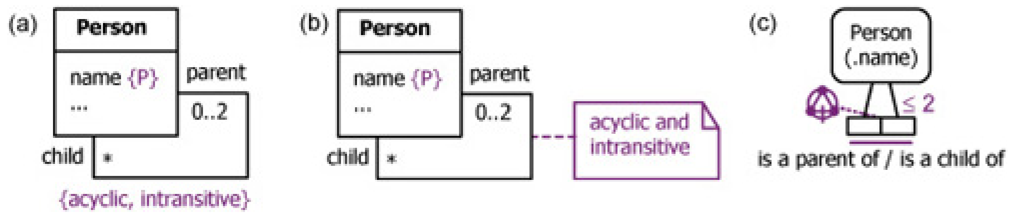
Other constraints in UML

Based on *Ch 9.7* in Halpin, T. & Morgan, T. 2008, *Information Modeling and Relational Databases*, Second Edition

- Value constraints
  - Enumeration types may be modeled as classes, stereotyped as enumerations, with their values listed as attributes
  - Ranges and mixtures may be specified by declaring a textual constraint in braces



- Ring constraints: UML does not provide ring constraints built in
  - Can be specified as a textual constraint or as a note



Intro to Resource Description Framework (RDF)

(Most of the examples in the upcoming slides are taken from: http://www.w3.org/TR/rdf-primer/)

- RDF is a language that enable to describe making statements on resources
  - John is father of Ann
- Statement (or triple) as a logical formula P(x, y), where the binary predicate P relates the object x to the object y
- Triple data model:

  <subject, predicate, object>

  - **Subject**: Resource or blank node
  - **Predicate**: Property
  - **Object**: Resource (or collection of resources), literal or blank node
- Example:

  <ex:john, ex:father-of, ex:ann>
- RDF offers only binary predicates (properties)

Resources

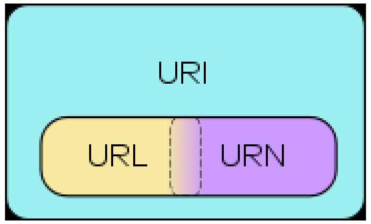
- A resource may be:
  - Web page (e.g. `http://www.w3.org`)
  - A person (e.g. `http://www.w3.org/People/Berners-Lee/`)
  - A book (e.g. `urn:isbn:4-534-34674-4`)
  - Anything denoted with a URI!

- A URI is an *identifier* and **not** a location on the Web

- RDF allows making statements about resources:
  - `http://www.w3.org` **has the format** text/html
  - `http://www.w3.org/People/Berners-Lee/` **has first name** Tim
  - `urn:isbn:0-345-33971-1` **has author** John

URI, URN, URL

- A Uniform Resource Identifier (URI) is a string of characters used to identify a name or a resource on the Internet



- A URI can be a URL or a URN
- A Uniform Resource Name (URN) defines an item's identity
  - the URN *urn:isbn:urn:isbn:4-534-34674-4* is a URI that specifies the identifier system, i.e. International Standard Book Number (ISBN), as well as the unique reference within that system and allows one to talk about a book, but doesn't suggest where and how to obtain an actual copy of it
- A Uniform Resource Locator (URL) provides a method for finding it
  - the URL *https://www.uio.no/studier/emner/matnat/ifi/INF1300* identifies a resource (INF1300's home page) and implies that a representation of that resource (such as the home page's current HTML code, as encoded characters) is obtainable via HTTP from a network host named *https://www.uio.no*

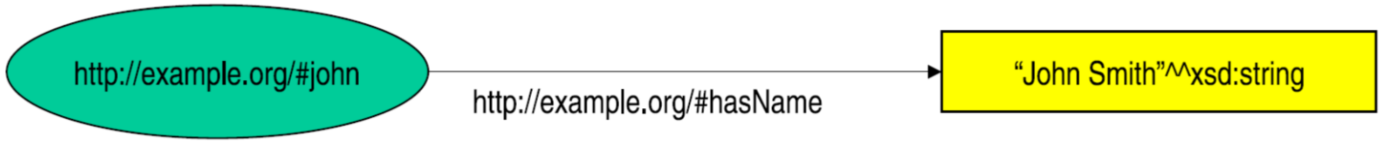
Literals

- Plain literals
  - E.g. **`”any text”`**
  - Optional language tag, e.g. **`”Hello, how are you?”@en-GB`**

- Typed literals
  - E.g. **`"hello"^^xsd:string`**, **`"1"^^xsd:integer`**
  - Recommended datatypes:
    - XML Schema datatypes

- Only as *object* of a triple, e.g.:

```
⟨<http://example.org/#john>,
                 <http://example.org/#hasName>,
                                          ”John Smith”ˆˆxsd:string⟩
```



Datatypes

- One pre-defined datatype: **`rdf:XMLLiteral`**
  - Used for embedding XML in RDF

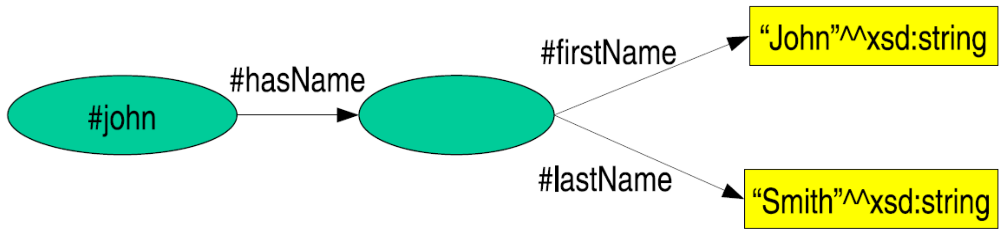
- Recommended datatypes are XML Schema datatypes, e.g.:
  - **`xsd:string`**
  - **`xsd:integer`**
  - **`xsd:float`**
  - **`xsd:anyURI`**
  - **`xsd:boolean`**

Blank Nodes I

- Blank nodes are nodes without a URI
  - Unnamed resources
  - More complex constructs
- Representation of blank nodes is syntax-dependent
  - *Blank node identifier*

- For example:

```
⟨<#john>, <#hasName>, _:johnsname⟩
⟨_:johnsname, <#firstName>, "John"^^xsd:string⟩
⟨_:johnsname, <#lastName>, "Smith"^^xsd:string⟩
```



Blank Nodes II

- **Representation of complex data**

  A blank node can be used to indirectly attach to a resource a consistent set of properties which together represent a complex data

- **Anonymous classes in OWL**

  The ontology language OWL uses blank nodes to represent anonymous classes such as unions or intersections of classes, or classes called restrictions, defined by a constraint on a property

RDF Containers

- Grouping property values:

“The lecture is attended by John, Mary and Chris”	**`Bag`**
“[RDF-Concepts] is edited by Graham and Jeremy (in that order)”	**`Seq`**
“The source code for the application may be found at ftp1.example.org, ftp2.example.org, ftp3.example.org”	**`Alt`**

RDF Containers 2

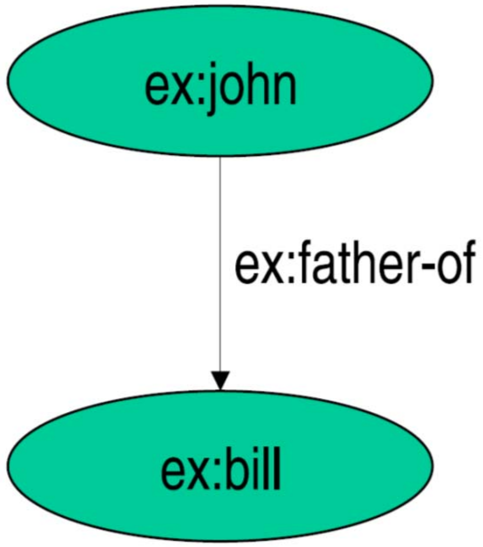
- Three types of containers:
  - **`rdf:Bag`** - unordered set of items
  - **`rdf:Seq`** - ordered set of items
  - **`rdf:Alt`** - set of alternatives
- Every container has a triple declaring the **`rdf:type`**
- Items in the container are denoted with
  - **`rdf:_1, rdf:_2, . . . ,rdf:_n`**

RDF Containers 2

- Three types of containers:
  - **`rdf:Bag`** - unordered set of items
  - **`rdf:Seq`** - ordered set of items
  - **`rdf:Alt`** - set of alternatives
- Every container has a triple declaring the **`rdf:type`**
- Items in the container are denoted with
  - **`rdf:_1, rdf:_2, . . . ,rdf:_n`**
- Limitations:
  - Semantics of the container is up to the application
  - What about closed sets?
    - How do we know whether Graham and Jeremy are the only editors of [RDF-Concepts]?

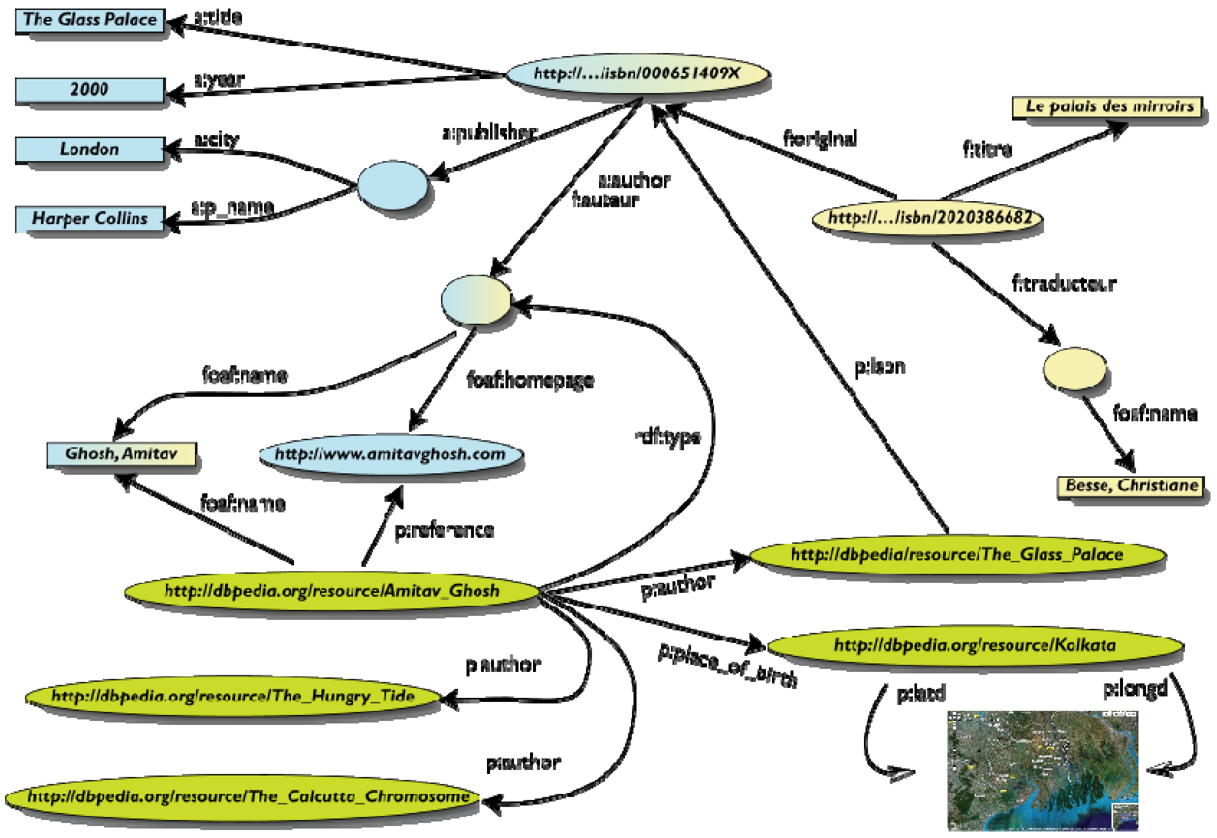
RDF Triple Graph Representation

- The triple data model can be represented as a graph
- Such graph is called in the Artificial Intelligence community a **semantic net**
- Labeled, directed graphs
  - **Nodes**: resources, literals
  - **Labels**: properties
  - **Edges**: statements



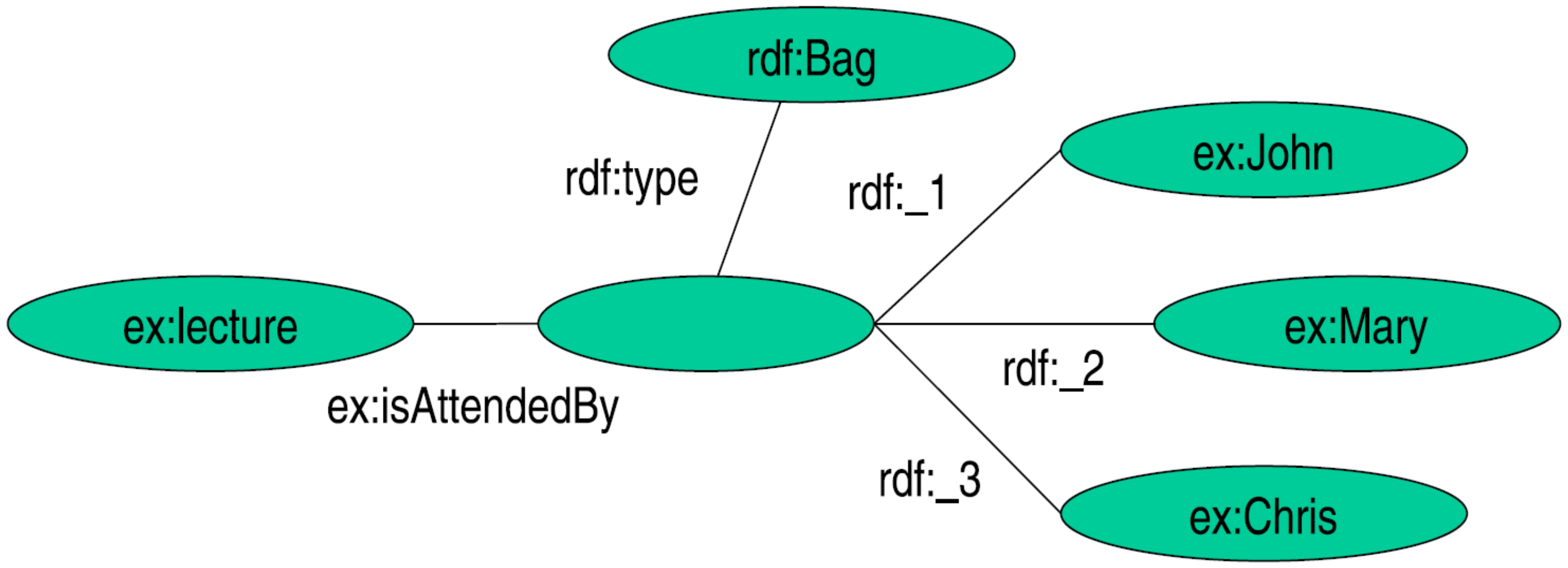
RDF: a Direct Connected Graph based Model

- Different interconnected triples lead to a more complex graphic model
- Basically a RDF document is a direct connect graph
  - http://en.wikipedia.org/wiki/Connectivity_%28graph_theory%29



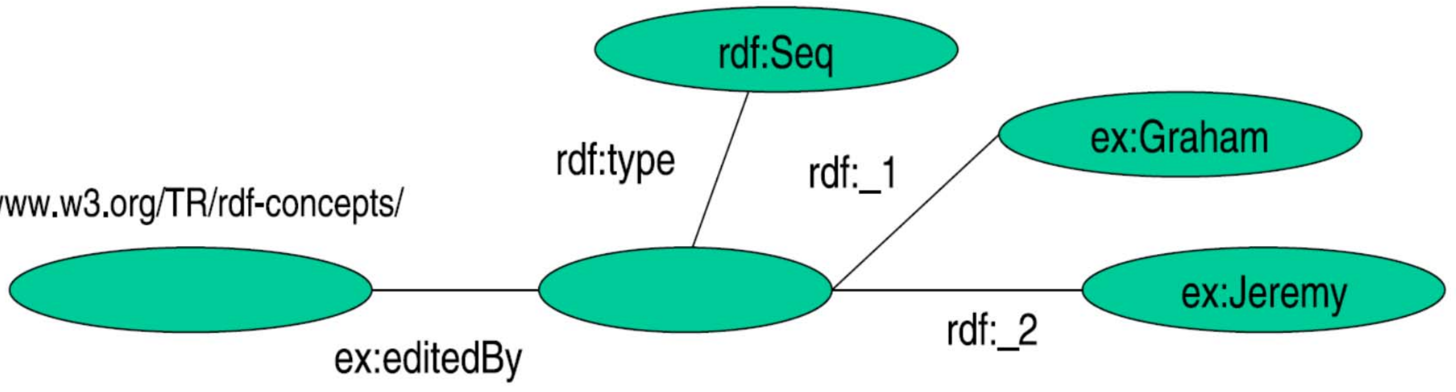
RDF Containers Graph Representation: Bag

“The lecture is attended by John, Mary and Chris”



RDF Containers Graph Representation: Seq

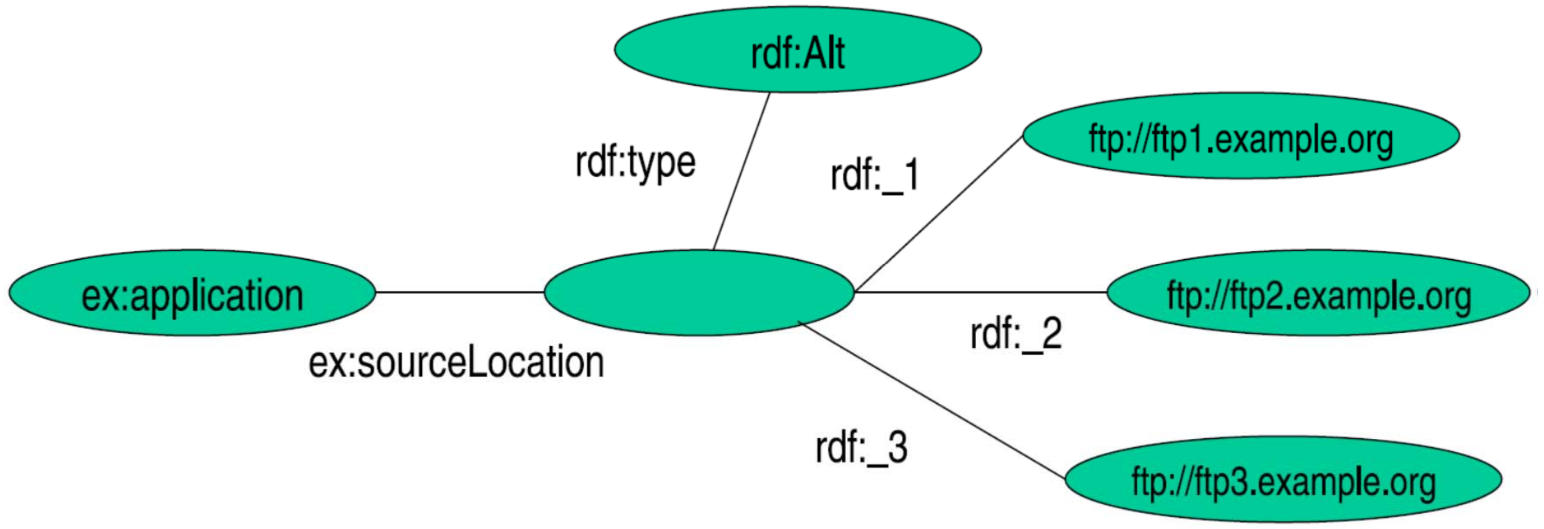
“[RDF-Concepts] is edited by Graham and Jeremy (<u>in that order</u>)”



RDF Containers Graph Representation: Alt

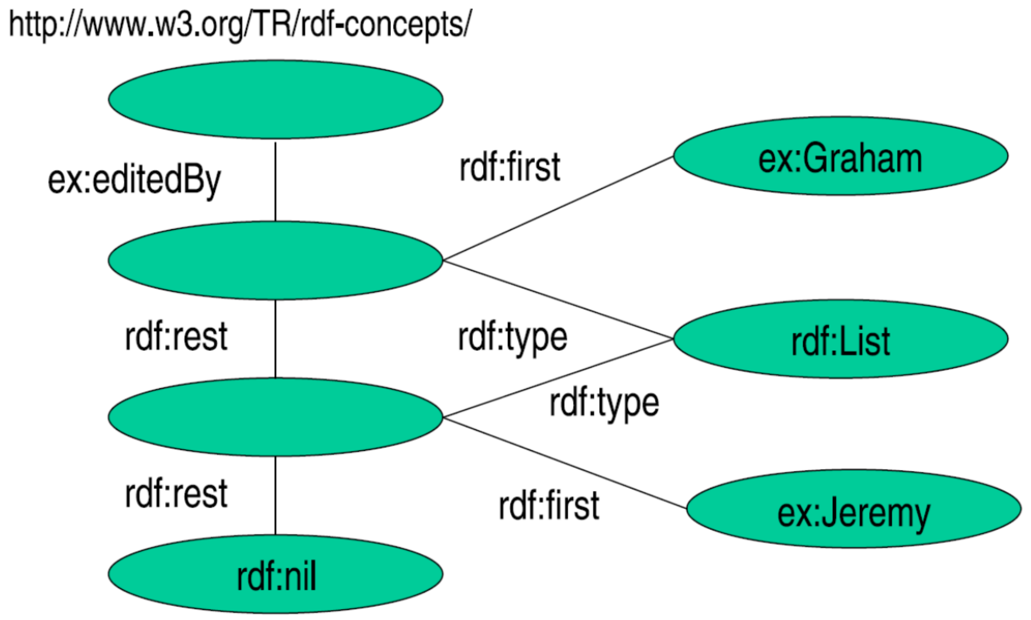
“The source code for the application may be found at

***ftp1.example.org**, **ftp2.example.org**, **ftp3.example.org**”*



RDF Collections

"[RDF-Concepts] is edited by Graham and Jeremy (<u>in that order</u>) and nobody else"



Reification I

- Reification: statements about statements

Mary claims that John’s name is “John Smith”.

```
⟨<#myStatement>, rdf:type, rdf:Statement⟩
⟨<#myStatement>, rdf:subject, <#john>⟩
⟨<#myStatement>, rdf:predicate, <#hasName>⟩
⟨<#myStatement>, rdf:object, ”John Smith”⟩
```

This kind of statement can be used to describe belief or trust in other statements, which is important in some kinds of applications

Necessary because there are only triples in RDF: we cannot add an identifier directly to a triple (then it would be a quadruple)

Reification II

- Reification: statements about statements

Mary claims that John’s name is “John Smith”.

```
⟨<#myStatement>, rdf:type, rdf:Statement⟩
⟨<#myStatement>, rdf:subject, <#john>⟩
⟨<#myStatement>, rdf:predicate, <#hasName>⟩
⟨<#myStatement>, rdf:object, ”John Smith”⟩
```

$$\Updownarrow$$

```
⟨<#john>, <#hasName>, ”John Smith”⟩
```

In such a way we attached a label to the statement.

Reification III

- Reification: statements about statements

Mary claims that John’s name is “John Smith”.

```
⟨<#myStatement>, rdf:type, rdf:Statement⟩
⟨<#myStatement>, rdf:subject, <#john>⟩
⟨<#myStatement>, rdf:predicate, <#hasName>⟩
⟨<#myStatement>, rdf:object, ”John Smith”⟩

⟨<#mary>, <#claims>, <#myStatement>⟩
```

RDF uses only binary properties. This restriction seems quite serious because often we use predicates with more than two arguments. Luckily, such predicates can be simulated by a number of binary predicates.

RDF Vocabulary

- RDF defines a number of resources and properties
- We have already seen: **`rdf:XMLLiteral`**, **`rdf:type`**, . . .
- RDF vocabulary is defined in the namespace:
  `http://www.w3.org/1999/02/22-rdf-syntax-ns#`

- Classes:
  - **`rdf:Property, rdf:Statement, rdf:XMLLiteral`**
  - **`rdf:Seq, rdf:Bag, rdf:Alt, rdf:List`**
- Properties:
  - **`rdf:type, rdf:subject, rdf:predicate, rdf:object,`**
  - **`rdf:first, rdf:rest, rdf:_n`**
  - **`rdf:value`**
- Resources:
  - **`rdf:nil`**

RDF Vocabulary

- Typing using **`rdf:type`**:

  `<A, rdf:type, B>`

  “A belongs to class B”

- All properties belong to class **`rdf:Property`**:

  `<P, rdf:type, rdf:Property>`

  “P is a property”

  `<rdf:type, rdf:type, rdf:Property>`

  “rdf:type is a property”

RDF Schema (RDFS)

- Types in RDF:

```
    <#john, rdf:type, #Student>
```

- What is a "**`#Student`**"?
- RFD is not defining a vocabulary about the statements, but only to express statements
- We know that "**`#Student`**" identifies a category (a concept or a class), but this is only implicitly defined in RDF

RDF Schema (RDFS)

- We need a language for defining RDF types:
  - Define classes:
    - *“**`#Student`** is a class”*
  - Relationships between classes:
    - *“**`#Student`** is a sub-class of **`#Person`**”*
  - Properties of classes:
    - *“**`#Person`** has a property **`hasName`**”*

- RDF Schema is such a language

RDF Schema (RDFS)

- Classes:
  `<#Student, rdf:type, #rdfs:Class>`
- Class hierarchies:
  `<#Student, rdfs:subClassOf, #Person>`

- Properties:
  `<#hasName, rdf:type, rdf:Property>`
- Property hierarchies:
  `<#hasMother, rdfs:subPropertyOf, #hasParent>`

- Associating properties with classes (a):
  - “The property **`#hasName`** only applies to **`#Person`**”
    `<#hasName, rdfs:domain, #Person>`

- Associating properties with classes (b):
  - “The type of the property **`#hasName`** is **`#xsd:string`**”
    `<#hasName, rdfs:range, xsd:string>`

RDFS Vocabulary

- RDFS Extends the RDF Vocabulary
- RDFS vocabulary is defined in the namespace:

http://www.w3.org/2000/01/rdf-schema#

RDFS Classes

- `rdfs:Resource`
- `rdfs:Class`
- `rdfs:Literal`
- `rdfs:Datatype`
- `rdfs:Container`
- `rdfs:ContainerMembershipProperty`

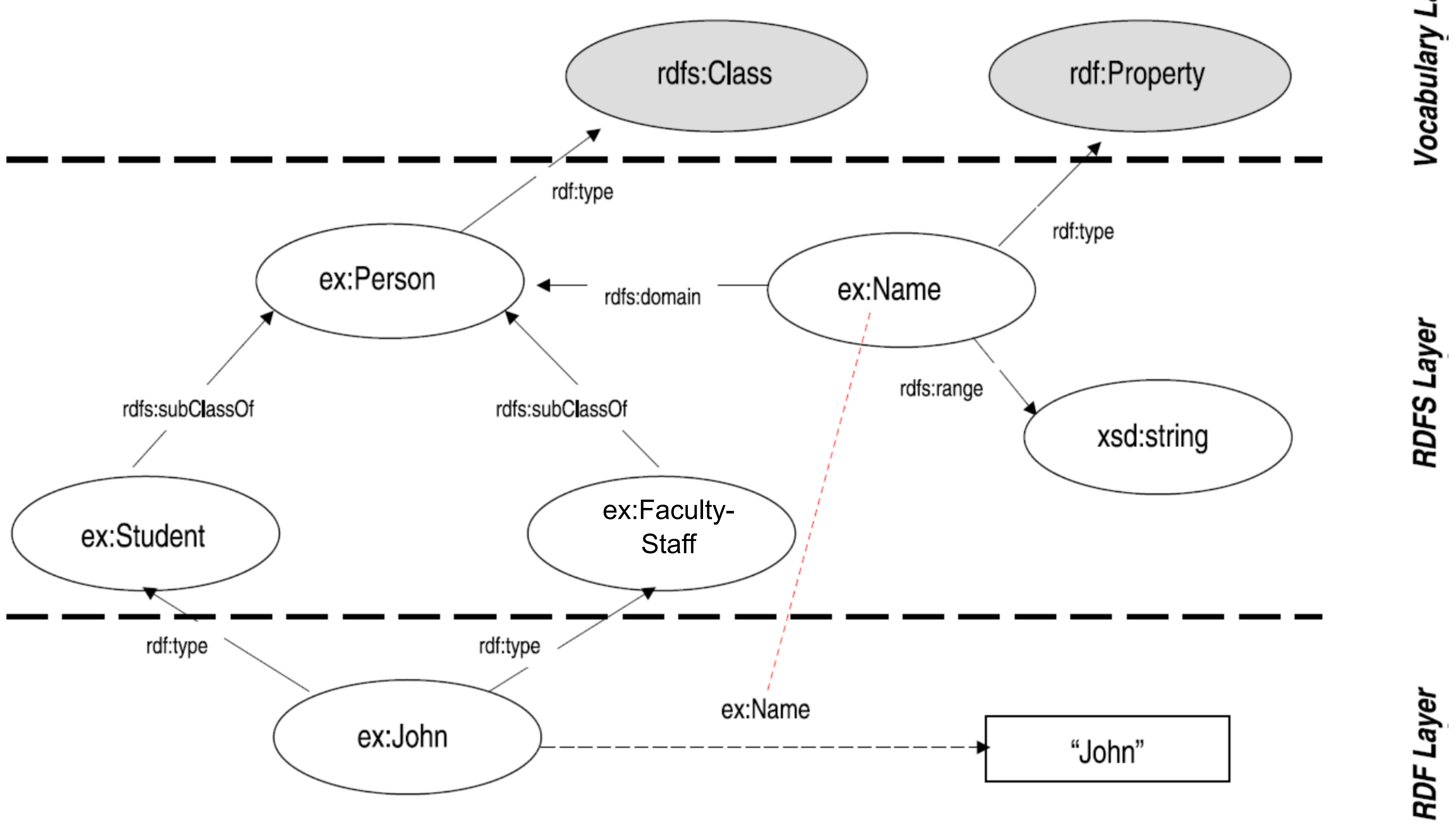
RDFS Properties

- `rdfs:domain`
- `rdfs:range`
- `rdfs:subPropertyOf`
- `rdfs:subClassOf`
- `rdfs:member`
- `rdfs:seeAlso`
- `rdfs:isDefinedBy`
- `rdfs:comment`
- `rdfs:label`

RDFS Principles

- **Resource**
  - All resources are implicitly instances of `rdfs:Resource`
- **Class**
  - Describe sets of resources
  - Classes are resources themselves - e.g. Webpages, people, document types
    - Class hierarchy can be defined through `rdfs:subClassOf`
    - Every class is a member of `rdfs:Class`
- **Property**
  - Subset of RDFS Resources that are properties
    - **Domain**: class associated with property: `rdfs:domain`
    - **Range**: type of the property values: `rdfs:range`
    - Property hierarchy defined through: `rdfs:subPropertyOf`

RDFS Example



RDFS Metadata Properties

- Metadata is “data about data”
- Any meta-data can be attached to a resource, using:
  - **`rdfs:comment`**
    - Human-readable description of the resource, e.g.
      - ⟨**`<ex:Person>, rdfs:comment, ”A person is any human being”`**⟩
  - **`rdfs:label`**
    - Human-readable version of the resource name, e.g.
      - ⟨**`<ex:Person>, rdfs:label, ”Human being”`**⟩
  - **`rdfs:seeAlso`**
    - Indicate additional information about the resource, e.g.
      - ⟨**`<ex:Person>, rdfs:seeAlso, <http://xmlns.com/wordnet/1.6/Human>`**⟩
  - **`rdfs:isDefinedBy`**
    - A special kind of rdfs:seeAlso, e.g.
      - ⟨**`<ex:Person>,rdfs:isDefinedBy,<http://xmlns.com/wordnet/1.6/Human>`**⟩

Databases and RDF (cont’)

- Relational database are a well established technology to store information and provide query support (SQL)
- Relational database have been designed and implemented to store concepts in a predefined (not frequently alterable) schema.

- How can we store the following RDF data in a relational database?

```
<rdf:Description rdf:about="12345">
      <rdf:type rdf:resource="&uni;lecturer"/>
      <uni:name>Joe Doe</uni:name>
      <uni:title>University Professor</uni:title>
</rdf:Description>
```

Databases and RDF

- Possible approach: Relational “Traditional” approach

Lecturer		
id	**name**	**title**
12345	Joe Doe	University Professor

- We can create a table “Lecturer” to store information about the “Lecturer” RDF Class.
- Query: Find the names of all the lecturers

```
SELECT NAME FROM LECTURER
```

- Drawbacks: Every time we need to add new content we have to create a new table -> Not scalable, not dynamic, not based on the RDF principles (triples)

Databases and RDF

- Another possible approach: Relational “Triple” based approach

Statement			
Subject	**Predicate**	**ObjectURI**	**ObjectLiteral**
101	102	103	null
101	104		201
101	105		202
103	…	…	null

Resources	
Id	**URI**
101	21345
102	rdf:type
103	uni:lecturer
104	…

Literals	
Id	**Value**
201	Joe Doe
202	University Professor
203	…
…	…

- We can create a table to maintain all the triples S P O (and distinguish between URI objects and literals objects)
- Drawbacks: We are flexible w.r.t. adding new statements dynamically without any change to the database structure…but what about querying?
  - Query: Find the names of all the lecturers
  - The query is quite complex: 5 JOINS!
  - This require a lot of optimization specific for RDF and triple data storage, that it is not included in the DB
  - For achieving efficiency a layer on top of a database is required
  - SQL is not appropriate to extract RDF fragments

```
SELECT L.Value FROM Literals AS L
INNER JOIN Statement AS S ON S.ObjectLiteral=L.ID
INNER JOIN Resources AS R ON R.ID=S.Predicate
INNER JOIN Statement AS S1 ON
S1.Predicate=S.Predicate
INNER JOIN Resources AS R1 ON R1.ID=S1.Predicate
INNER JOIN Resources AS R2 ON R2.ID=S1.ObjectURI
WHERE R.URI = “uni:name”
AND R1.URI = “rdf:type”
AND R2.URI = “uni:lecturer”
```

SPARQL: RDF Query language

- SPARQL
  - RDF Query language
  - Uses SQL-like syntax

- Example:

```
PREFIX uni: <http://example.org/uni/>

SELECT ?name
FROM <http://example.org/personal>
WHERE { ?s uni:name ?name.
?s rdf:type uni:lecturer }
```

SPARQL Queries

```
PREFIX uni: <http://example.org/uni/>
SELECT ?name
FROM <http://example.org/personal>
WHERE { ?s uni:name ?name. ?s rdf:type uni:lecturer }
```

- PREFIX
  - Prefix mechanism for abbreviating URIs
- SELECT
  - Identifies the variables to be returned in the query answer
  - SELECT DISTINCT
  - SELECT REDUCED
- FROM
  - Name of the graph to be queried
  - FROM NAMED
- WHERE
  - Query pattern as a list of triple patterns
- LIMIT
- OFFSET
- ORDER BY

SPARQL Query keywords

- PREFIX: based on namespaces
- DISTINCT: The DISTINCT solution modifier eliminates duplicate solutions. Specifically, each solution that binds the same variables to the same RDF terms as another solution is eliminated from the solution set.
- REDUCED: While the DISTINCT modifier ensures that duplicate solutions are eliminated from the solution set, REDUCED simply permits them to be eliminated. The cardinality of any set of variable bindings in an REDUCED solution set is at least one and not more than the cardinality of the solution set with no DISTINCT or REDUCED modifier.
- LIMIT: The LIMIT clause puts an upper bound on the number of solutions returned. If the number of actual solutions is greater than the limit, then at most the limit number of solutions will be returned.

SPARQL Query keywords

- OFFSET: OFFSET causes the solutions generated to start after the specified number of solutions. An OFFSET of zero has no effect.

- ORDER BY: The ORDER BY clause establishes the order of a solution sequence.

- Following the ORDER BY clause is a sequence of order comparators, composed of an expression and an optional order modifier (either ASC() or DESC()). Each ordering comparator is either ascending (indicated by the ASC() modifier or by no modifier) or descending (indicated by the DESC() modifier).

Example RDF Graph

```
<http://example.org/#john> <http://.../vcard-rdf/3.0#FN> "John Smith“

<http://example.org/#john> <http://.../vcard-rdf/3.0#N> :_X1
_:X1 <http://.../vcard-rdf/3.0#Given> "John"
_:X1 <http://.../vcard-rdf/3.0#Family> "Smith“

<http://example.org/#john> <http://example.org/#hasAge> "32“

<http://example.org/#john> <http://example.org/#marriedTo> <#mary>

<http://example.org/#mary> <http://.../vcard-rdf/3.0#FN> "Mary Smith“

<http://example.org/#mary> <http://.../vcard-rdf/3.0#N> :_X2
_:X2 <http://.../vcard-rdf/3.0#Given> "Mary"
_:X2 <http://.../vcard-rdf/3.0#Family> "Smith"

<http://example.org/#mary> <http://example.org/#hasAge> "29"
```

SPARQL Queries: All Full Names

“Return the full names of all people in the graph”

```
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?fullName
WHERE {?x vCard:FN ?fullName}
```

result:

```
fullName
=================
"John Smith"
"Mary Smith"
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

SPARQL Queries: Properties

“Return the relation between John and Mary”

```
PREFIX ex: <http://example.org/#>
SELECT ?p
WHERE {ex:john ?p ex:mary}
```

result:

```
p
===================
<http://example.org/#marriedTo>
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

SPARQL Queries: Complex Patterns

```
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
PREFIX ex: <http://example.org/#>
SELECT ?y
WHERE {?x vCard:FN "John Smith".
       ?x ex:marriedTo ?y}
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

SPARQL Queries: Complex Patterns

"Return the spouse of a person by the name of John Smith"

```
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
PREFIX ex: <http://example.org/#>
SELECT ?y
WHERE {?x vCard:FN "John Smith".
       ?x ex:marriedTo ?y}
```

result:

```
y
=====================
<http://example.org/#mary>
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

SPARQL Queries: Blank Nodes

```
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?name, ?firstName
WHERE {?x vCard:N ?name .
       ?name vCard:Given ?firstName}
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

SPARQL Queries: Blank Nodes

“Return the first name of all people in the KB”

```
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?name, ?firstName
WHERE {?x vCard:N ?name .
      ?name vCard:Given ?firstName}
```

result:

```
name firstName
===================
_:a "John"
_:b "Mary"
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

SPARQL Queries: Building RDF Graph

“Rewrite the naming information in original graph by using the **`foaf:name`**”

```
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>

CONSTRUCT { ?x foaf:name ?name }
WHERE  { ?x vCard:FN ?name }
```

result:

```
#john foaf:name “John Smith"
#marry foaf:name “Marry Smith"
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

SPARQL Queries: Building RDF Graph

"Rewrite the naming information in original graph by using the **`foaf:name`**"

```
PREFIX vCard: <http://www.w3.org/2001/vcard-rdf/3.0#>
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>
```

```
CONSTRUCT { ?x foaf:name ?name }
WHERE   { ?x vCard:FN ?name }
```

result:

```
#john foaf:name "John Smith"
#marry foaf:name "Marry Smith"
```

```
<rdf:RDF
       xmlns:rdf="http://www.w3.org/1999/02/22-rdf-
              syntax-ns#"
       xmlns:foaf="http://xmlns.com/foaf/0.1/"
       xmlns:ex="http://example.org">
  <rdf:Description rdf:about=ex:john>
        <foaf:name>John Smith</foaf:name>
  </rdf:Description>
  <rdf:Description rdf:about=ex:marry>
        <foaf:name>Marry Smith</foaf:name>
  </rdf:Description>
</rdf:RDF>
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

SPARQL Queries: Testing if the Solution Exists

"Are there any married persons in the KB?"

```
PREFIX ex: <http://example.org/#>
ASK { ?person ex:marriedTo ?spouse }
```

result:

```
yes
===================
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

SPARQL Queries: Constraints (Filters)

“Return all people over 30 in the KB”

```
PREFIX ex: <http://example.org/#>
SELECT ?x
WHERE {?x hasAge ?age .
FILTER(?age > 30)}
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

result:

```
x
================
<http://example.org/#john>
```

SPARQL Queries: Optional Patterns

“Return all people and (optionally) their spouse”

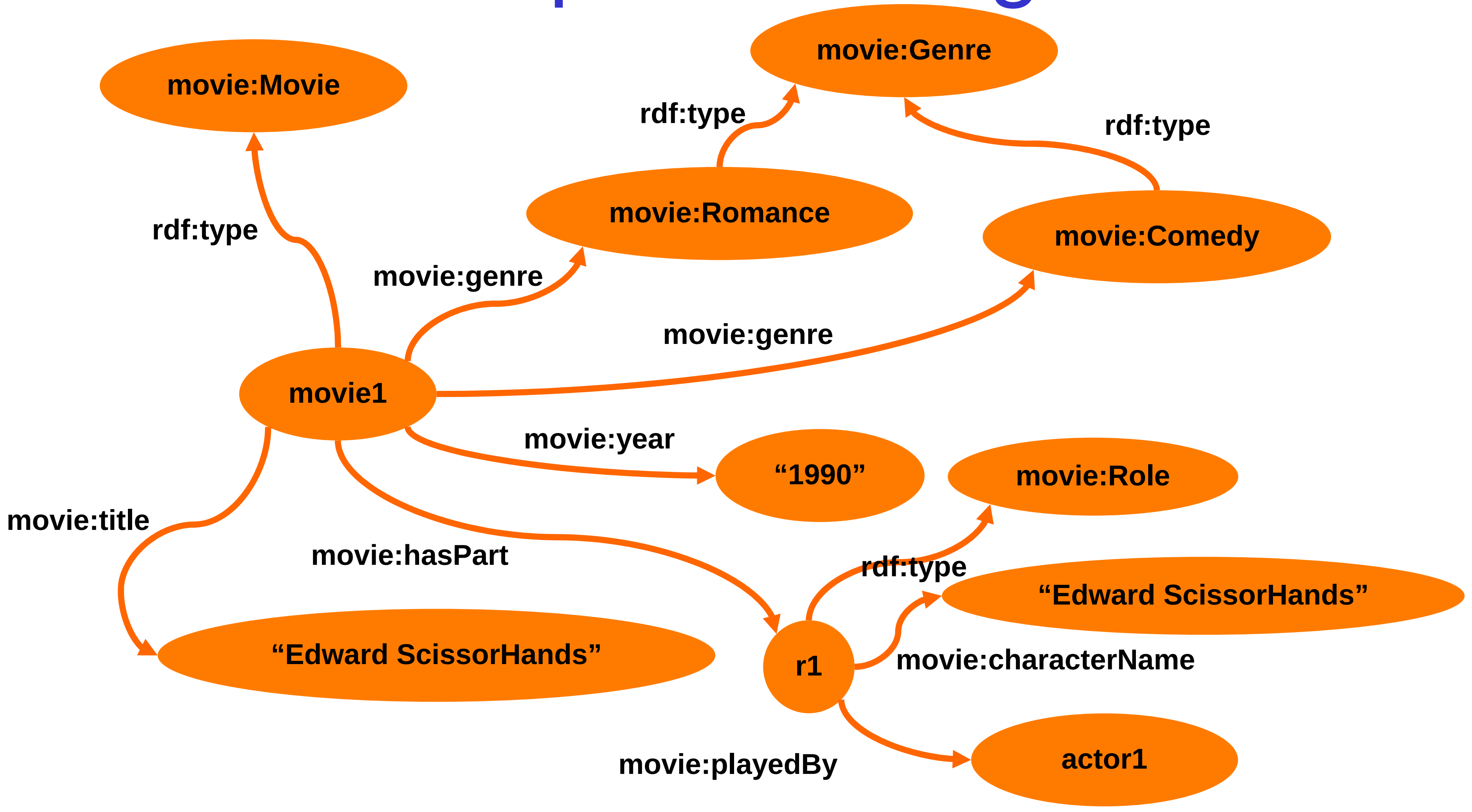
```
PREFIX ex: <http://example.org/#>
SELECT ?person, ?spouse
WHERE {?person ex:hasAge ?age .
OPTIONAL { ?person ex:marriedTo ?spouse } }
```

```
@prefix ex: <http://example.org/#> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
ex:john
  vcard:FN "John Smith" ;
  vcard:N [
    vcard:Given "John" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 32 ;
  ex:marriedTo :mary .
ex:mary
  vcard:FN "Mary Smith" ;
  vcard:N [
    vcard:Given "Mary" ;
    vcard:Family "Smith" ] ;
  ex:hasAge 29 .
```

result:

```
?person ?spouse
=============================
<http://example.org/#mary>
<http://example.org/#john> <http://example.org/#mary>
```

A RDF Graph Modeling Movies



[http://www.openrdf.org/conferences/eswc2006/Sesame-tutorial-eswc2006.ppt]

Example Query 1

- Select the movies that has a character called “Edward Scissorhands”

```
PREFIX movie: <http://example.org/movies/>

SELECT DISTINCT ?x ?t
WHERE {
                    ?x movie:title ?t ;
                    movie:hasPart ?y .
                    ?y movie:characterName ?z .
                    FILTER (?z = “Edward Scissorhands”@en)
      }
```

Example Query 1

```
PREFIX movie: <http://example.org/movies/>

SELECT DISTINCT ?x ?t
WHERE {
                    ?x movie:title ?t ;
                    movie:hasPart ?y .
                    ?y movie:characterName ?z .
                    FILTER (?z = “Edward Scissorhands”@en)
      }
```

- Note the use of “;” This allows to create triples referring to the previous triple pattern (extended version would be **?x movie:hasPart ?y**)
- Note as well the use of the language speciation in the filter @**en**

Example Query 2

- Create a graph of actors and relate them to the movies they play in (through a new ‘playsInMovie’ relation)

```
PREFIX movie: <http://example.org/movies/>
PREFIX foaf:   <http://xmlns.com/foaf/0.1/>

CONSTRUCT {
       ?x foaf:firstName ?fname.
       ?x foaf:lastName ?lname.
       ?x movie:playInMovie ?m
       }
WHERE  {
              ?m movie:title ?t ;
              movie:hasPart ?y .
              ?y movie:playedBy ?x .
       ?x foaf:firstName ?fname.
       ?x foaf:lastName ?lname.
}
```

Example Query 3

- Find all movies which share at least one genre with “Gone with the Wind”

```
PREFIX movie: <http://example.org/movies/>

SELECT DISTINCT ?x2 ?t2
WHERE {
                      ?x1 movie:title ?t1.
                      ?x1 movie:genre ?g1.
                      ?x2 movie:genre ?g2.
                      ?x2 movie:title ?t2.
                      FILTER (?t1 = “Gone with the Wind”@en &&
                      ?x1!=?x2 && ?g1=?g2)
      }
```