# Week 2: Knowledge Representation & OWL

Lectures 21-22 | Exam-Focused Notes

## Contents

# 1 Lecture 21 – Knowledge Representation with RDF & RDFS

## 1.1 Big picture and exam link

This lecture covers the **foundations of knowledge representation** using RDF and RDFS. Exam questions will test:

- Understanding RDF triples (subject, predicate, object).
- Converting knowledge graphs to RDF/XML and Turtle formats.
- Understanding RDFS vocabulary: classes, properties, domain, range.
- RDF containers: Bag, Seq, Alt, Collection.
- Blank nodes (anonymous nodes).
- RDFS inference rules (type inheritance, property constraints).
- SKOS vocabulary for taxonomies and thesauri.
- Limitations of RDFS (when you need OWL).

**Key exam mindset:** You must be able to:

1. Read a knowledge graph and write RDF triples.
2. Given RDF/XML or Turtle, draw the corresponding graph.
3. Apply RDFS constraints (domain, range, subClassOf).
4. Perform simple inference (if X subClassOf Y, then instances of X are also Y).

   If you can do these four things from memory, Lecture 21 is covered.

## 1.2 From Web of Documents to Web of Data

**Problem with traditional web:** [file:5]

- Web pages are written for *humans* to read (HTML).
- Machines can *display* pages but cannot *understand* them.
- Example: "John is a lecturer who teaches Big Data Management at Heriot-Watt University."
  - Human knows: John = person, teaches = relationship, Big Data Management = course.
  - Machine sees: just a string of text with no semantic meaning.

  **Solution: Web of Data** [file:5]

- Represent knowledge as a **graph** of entities and relationships.
- Use **semantic technologies** (RDF, RDFS, OWL) to encode meaning.
- Machines can now:
  - Search based on entities and relationships (not just keywords).
  - Integrate data from multiple sources automatically.
  - Perform reasoning and inference.

## 1.3 Knowledge Graphs: The Core Idea

**Definition:** [file:5]

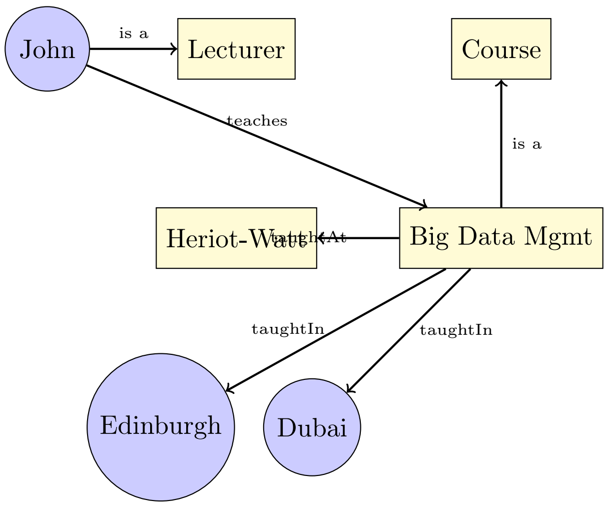
A knowledge graph represents knowledge as:

- **Nodes:** Things (resources) like people, places, concepts.
- **Edges:** Relationships (properties) between things.

**Example sentence:** [file:5]

"John is a lecturer who teaches Big Data Management course at Heriot-Watt University taught in Edinburgh and Dubai."

**As a knowledge graph:**



**Key characteristics of knowledge graphs:** [file:5]

1. **Scaling:** Same graph can represent multiple domains (geography, people, organizations).
2. **Agreement:** Need standardized relationships (everyone uses same terms).
3. **Structure:** Must define which nodes connect via which relationships.
4. **Plurality:** Same relationship can appear multiple times (e.g., "taughtIn").
5. **Asymmetry:** Relationships are directed (rarely symmetric).

## 1.4 Reading a Knowledge Graph as Triples

**Triple = (Subject, Predicate, Object)** [file:5]

A knowledge graph can be written as a set of triples (also called statements).

**Format:**

```
Subject  Predicate  Object
```

**Example from the graph above:** [file:5]

1. John `isA` Lecturer
2. John `teaches` BigDataManagement
3. BigDataManagement `isA` Course
4. BigDataManagement `taughtAt` HeriotWatt
5. BigDataManagement `taughtIn` Edinburgh
6. BigDataManagement `taughtIn` Dubai

## 1.5 RDF: Resource Description Framework

**What is RDF?** [file:5]

RDF is a **standard language** to represent knowledge graphs using triples.

**RDF Triple Components:** [file:5]

- **Subject:** A resource (identified by IRI).
- **Predicate:** A property/relationship (identified by IRI).
- **Object:** Can be:
  - A resource (IRI), or
  - A literal value (string, number, date with datatype).

**Key point:** RDF defines the *semantics* (meaning) in a machine-interpretable format. [file:5]

### 1.5.1 RDF Serialization Formats

RDF is an abstract model. To write it in files, we use serialization formats: [file:5]

1. **RDF/XML:** Original W3C standard (verbose, XML-based).
2. **Turtle:** Human-readable, compact (uses prefixes).
3. **N-Triples:** One triple per line, full IRIs (no abbreviations).
4. **N-Quads:** N-Triples + graph context (4th element).
5. **JSON-LD:** JSON format for RDF (used by Google).
6. **RDFa:** Embeds RDF in HTML attributes.

**Exam tip:** You'll mostly see RDF/XML and Turtle. Know how to convert between them!

## 1.6 RDF/XML: Key Elements

**Essential RDF/XML elements:** [file:5]

| **Element** | **Description** |
|---|---|
| `rdf:RDF` | Root element of RDF documents |
| `rdf:Description` | Describes a resource (identified by `rdf:about`) |
| `rdf:type` | Specifies the class/type of a resource ("instance of") |
| `rdf:Bag` | Unordered container of resources |
| `rdf:Seq` | Ordered container of resources |
| `rdf:Alt` | Set of alternative values (user picks one) |

### 1.6.1 RDF Attributes

| **Attribute** | **Usage** |
|---|---|
| `rdf:ID` | Abbreviates a node IRI (fragment identifier) |
| `rdf:about` | References an existing resource (full or relative IRI) |
| `rdf:resource` | Defines property object as a resource (not literal) |

## 1.7 RDF Containers: Bag, Seq, Alt (EXAM IMPORTANT!)

### 1.7.1 1. rdf:Bag (Unordered Collection)

**Use case:** Order doesn't matter. [file:5]
**Example:** Beatles band members (no particular order).
**RDF/XML:**

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:cd="http://www.recshop.fake/cd#">

  <rdf:Description rdf:about="http://www.recshop.fake/cd/Beatles">
    <cd:artist>
      <rdf:Bag>
        <rdf:li>John</rdf:li>
        <rdf:li>Paul</rdf:li>
        <rdf:li>George</rdf:li>
        <rdf:li>Ringo</rdf:li>
      </rdf:Bag>
    </cd:artist>
  </rdf:Description>
</rdf:RDF>
```

**Turtle equivalent:** [file:5]

```
@prefix cd: <http://www.recshop.fake/cd#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

<http://www.recshop.fake/cd/Beatles>
  cd:artist [
    a rdf:Bag ;
    rdf:_1 "John" ;
    rdf:_2 "Paul" ;
    rdf:_3 "George" ;
    rdf:_4 "Ringo"
  ] .
```

### 1.7.2 2. rdf:Seq (Ordered Collection)

**Use case:** Order matters (e.g., ranked list, chronological). [file:5]
**Example:** Beatles members in joining order.
**RDF/XML:**

```
<cd:artist>
  <rdf:Seq>
    <rdf:li>John</rdf:li>
    <rdf:li>Paul</rdf:li>
    <rdf:li>George</rdf:li>
    <rdf:li>Ringo</rdf:li>
  </rdf:Seq>
</cd:artist>
```

**Turtle:** [file:5]

```
cd:artist [
  a rdf:Seq ;
  rdf:_1 "George" ;
  rdf:_2 "John" ;
  rdf:_3 "Paul" ;
  rdf:_4 "Ringo"
] .
```

#### 1.7.3 3. rdf:Alt (Alternatives)

**Use case:** Multiple options, user chooses one. [file:5]
**Example:** Album formats (CD, Record, Tape).
**RDF/XML:**

```
<cd:format>
  <rdf:Alt>
    <rdf:li>CD</rdf:li>
    <rdf:li>Record</rdf:li>
    <rdf:li>Tape</rdf:li>
  </rdf:Alt>
</cd:format>
```

#### 1.7.4 4. rdf:Collection (Closed List)

**Use case:** Fixed set of members (no additions allowed). [file:5]
**RDF/XML:**

```
<cd:artist rdf:parseType="Collection">
  <rdf:Description rdf:about="cd:George"/>
  <rdf:Description rdf:about="cd:John"/>
  <rdf:Description rdf:about="cd:Paul"/>
  <rdf:Description rdf:about="cd:Ringo"/>
</cd:artist>
```

**Turtle:**

```
cd:artist ( cd:George cd:John cd:Paul cd:Ringo ) .
```

**Exam tip:** Be able to identify which container type to use:

- Unordered → `Bag`
- Ordered → `Seq`
- Alternatives → `Alt`
- Fixed list → `Collection`

### 1.8 Blank Nodes (Anonymous Nodes)

**What are blank nodes?** [file:5]
Nodes that don't need external visibility (no IRI assigned).
**When to use:**

- Intermediate nodes in complex structures.
- Nodes that only exist within a specific context.

**Example:** [file:5]

John knows someone named Paul Smith with email paul.smith@example.com, but we don't need a global IRI for Paul.

**RDF/XML:**

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:pep="http://example.com/people#"
  xml:base="http://example.com/">

  <rdf:Description rdf:about="pep:John">
    <pep:knows>
      <rdf:Description>
        <pep:surname>Smith</pep:surname>
        <pep:givenName>Paul</pep:givenName>
        <pep:email>paul.smith@example.com</pep:email>
      </rdf:Description>
    </pep:knows>
  </rdf:Description>
</rdf:RDF>
```

**Turtle:** [file:5]

```
@prefix pep: <http://example.com/people#> .

<pep:john>
  pep:knows [
    pep:surname "Smith" ;
    pep:givenName "Paul" ;
    pep:email "paul.smith@example.com"
  ] .
```

**Key point:** Square brackets `[ ]` in Turtle denote blank nodes.

## 1.9 XML Schema Datatypes for Literals

RDF literals have datatypes (from XML Schema): [file:5]

| **Datatype** | **Example** |
|---|---|
| `xsd:string` | "Hello World" |
| `xsd:int` | 42 |
| `xsd:float` | 3.14 |
| `xsd:boolean` | true, false |
| `xsd:date` | 2026-01-31 |
| `xsd:dateTime` | 2026-01-31T12:00:00 |

**Turtle syntax:**

```
ex:Bob foaf:age "32"^^xsd:int .
ex:Bob foaf:birthdate "1990-07-14"^^xsd:date .
```

## 1.10 RDF Limitations (Why We Need RDFS)

**Problem:** [file:5]

Pure RDF has no type mechanism:

- Nothing prevents using property “lastName” on a Car!
- No way to specify which properties apply to which types.
- Cannot define hierarchies (e.g., Cat is a Mammal is an Animal).

**Issues:**

1. **Interpretability:** Different people interpret predicates differently.
2. **Automatization:** Software cannot validate or make accurate inferences.

**Solution:** RDFS (RDF Schema) adds vocabulary to define classes, properties, and constraints!

## 1.11 RDFS: RDF Schema

**What is RDFS?** [file:5]
RDFS = RDF Vocabulary Description Language.
**Purpose:**

- Define the vocabulary used in RDF models.
- Create metamodels with:
  - **Concepts:** Resource, Literal, Class, Datatype.
  - **Relationships:** subClassOf, subPropertyOf, domain, range.
- Organize concepts into hierarchies.
- Define axioms (rules) for inference.

### 1.11.1 RDFS Classes

| **Class** | **Description** |
|---|---|
| `rdfs:Class` | Defines all classes (metaclass) |
| `rdfs:Resource` | Defines all resources (everything is a resource) |
| `rdfs:Literal` | Defines all literal values (int, string, date, etc.) |
| `rdfs:Property` | Defines all properties/relationships |
| `rdfs:Datatype` | Defines datatypes (subclass of `rdfs:Literal`) |

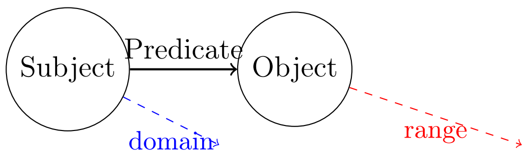
**Important distinction:** [file:5]

- `rdf:resource` (lowercase r) = attribute to reference an object
- `rdfs:Resource` (capital R) = class defining all resources

### 1.11.2 RDFS Properties (EXAM CRITICAL!)

| **Property** | **Description** |
|---|---|
| `rdfs:subClassOf` | Class hierarchy (A subClassOf B means A is more specific than B). Transitive. |
| `rdfs:subPropertyOf` | Property hierarchy (similar to subClassOf). Transitive. |
| `rdfs:domain` | Declares the class of the **subject** for a property |
| `rdfs:range` | Declares the class/datatype of the **object** for a property |

**Visual representation:** [file:5]



**Example:** [file:5]

```
:hasWife rdf:type owl:ObjectProperty ;
  rdfs:domain :Man ;
  rdfs:range :Woman .
```

Meaning: The subject of `hasWife` must be a `Man`, and the object must be a `Woman`.

### 1.11.3 RDFS Metadata Properties

| **Property** | **Description** |
|---|---|
| `rdfs:comment` | Human-readable description of the resource |
| `rdfs:label` | Human-friendly name for the resource |
| `rdfs:isDefinedBy` | Links to the schema/ontology defining this resource |
| `rdfs:seeAlso` | Links to related resources for more info |

## 1.12 RDFS Example 1: Animal Taxonomy

**Goal:** Define a simple class hierarchy for animals. [file:5]

**RDF/XML:**

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:aml="http://www.animals.fake/animals#"
  xml:base="http://www.animals.fake/animals">

  <rdfs:Class rdf:ID="animal"/>

  <rdfs:Class rdf:ID="horse">
    <rdfs:subClassOf rdf:resource="aml:animal"/>
  </rdfs:Class>

  <rdfs:Class rdf:ID="camel">
    <rdfs:subClassOf rdf:resource="aml:animal"/>
  </rdfs:Class>
</rdf:RDF>
```

**Turtle equivalent:** [file:5]

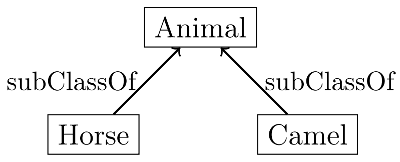
```
@base <http://www.animals.fake/animals> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix aml: <http://www.animals.fake/animals#> .

<aml#animal> a rdfs:Class .

<aml#horse> a rdfs:Class ;
  rdfs:subClassOf <aml#animal> .
```

```
<aml#camel> a rdfs:Class ;
  rdfs:subClassOf <aml#animal> .
```

**Visual hierarchy:**



### 1.13 Schema vs Instance (TBox vs ABox)

**Two levels of knowledge:** [file:5]

1. **Schema (TBox):** Types of things and their relationships.
   - Classes: Person, Animal, Course, University.
   - Properties: teaches, livesIn, worksAt.
2. **Instance (ABox):** Actual data conforming to the schema.
   - Individuals: John, Alice, HeriotWatt, Dubai.
   - Facts: John teaches BigData, Alice livesIn Dubai.

**Exam analogy:**

- **Schema** = Class definition in programming (blueprint).
- **Instance** = Object created from that class.

### 1.14 Inference with RDFS

**What is inference?** [file:5]
Discovering facts not explicitly stated by applying logical rules.

#### 1.14.1 Type 1: Type Inheritance

**Rule:** If X is a subclass of Y, then instances of X are also instances of Y. [file:5]
**Example:**

```
:alice rdf:type foaf:Person .
foaf:Person rdfs:subClassOf foaf:Agent .

==> Inference:
:alice rdf:type foaf:Agent .
```

**Why?** Because Person is a kind of Agent, Alice (a Person) is automatically an Agent.

#### 1.14.2 Type 2: Type Inference from Domain/Range

**Rule:** Use domain and range constraints to infer types. [file:5]
**Example:**

```
foaf:knows rdf:domain foaf:Person .
foaf:knows rdf:range foaf:Person .
:alice foaf:knows :bob .

==> Inference:
:alice rdf:type foaf:Person .
:bob rdf:type foaf:Person .
```

**Why?** Because `knows` requires both subject and object to be Person.

## 1.15 Characteristics of RDFS/RDF

**Key features (and quirks):** [file:5]

1. **No distinction between classes and instances:**

   ```
   <Species, type, Class>
   <Lion, type, Species>
   <Leo, type, Lion>
   ```

   Everything can be both a class and an instance!

2. **Properties can have properties:**

   ```
   <hasDaughter, subPropertyOf, hasChild>
   <hasDaughter, type, familyProperty>
   ```

3. **No distinction between constructors and vocabulary:**

   ```
   <type, range, Class>
   <Property, type, Class>
   <type, subPropertyOf, subClassOf>
   ```

   Constructors can be applied to themselves!

## 1.16 RDFS Vocabularies

RDFS allows defining vocabularies (data models) using: [file:5]

- **Classes:** Represent things (Person, Organization, Document).
- **Relationships:** Links between classes (publishes, depicts).
- **Attributes:** Characteristics (legalName, dateTime).

### 1.16.1 Common Vocabularies Used with RDF/RDFS

[file:5]

| **Vocabulary** | **Purpose** |
|---|---|
| RDF | Core RDF elements and properties |
| RDFS | RDF Schema vocabulary |
| Dublin Core (DC) | Document metadata (author, title, date) |
| SKOS | Structured vocabularies (taxonomies, thesauri) |
| FOAF | People, activities, relationships |
| Schema.org | General-purpose vocabulary for web markup |

### 1.17 Dublin Core (DC)

**Purpose:** Basic metadata for describing resources. [file:5]
**Common properties:**

- `dc:title` - Title of the resource
- `dc:creator` - Author/creator
- `dc:description` - Human-readable description
- `dc:date` - Date of creation
- `dc:publisher` - Publisher organization

**Example:**

```
@prefix dc: <http://purl.org/dc/elements/1.1/> .

<http://example.org/document1>
  dc:title "Big Data Management Notes" ;
  dc:creator "Dr. Radu Mihailescu" ;
  dc:date "2026-01-31" .
```

### 1.18 SKOS: Simple Knowledge Organization System

**What is SKOS?** [file:5]
SKOS is a W3C standard (2009) for representing knowledge organization systems:

- Thesauri
- Taxonomies
- Classification schemes
- Subject headings

**Why SKOS?** Controlled vocabularies lack relationships; SKOS adds structure. [file:5]

#### 1.18.1 SKOS Core Concepts

1. **skos:Concept** [file:5]
The fundamental unit representing an idea.
**Example:**

```
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix ex: <http://www.example.com/> .

ex:cat rdf:type skos:Concept ;
  skos:prefLabel "cat"@en ;
  skos:altLabel "feline"@en .
```

2. **Semantic Relationships** [file:5]

| **Property** | **Meaning** |
|---|---|
| `skos:broader` | More general concept (parent) |
| `skos:narrower` | More specific concept (child) |
| `skos:related` | Associative relationship (not hierarchical) |
| `skos:broaderTransitive` | Explicit transitive version of broader |
| `skos:narrowerTransitive` | Explicit transitive version of narrower |

**Example hierarchy:** [file:5]

```
:animal rdf:type skos:Concept .
:mammal rdf:type skos:Concept ;
  skos:broader :animal .
:cat rdf:type skos:Concept ;
  skos:broader :mammal .

==> By transitivity:
:cat skos:broaderTransitive :animal .
```

#### 1.18.2 SKOS Transitivity

[file:5]
**Transitive property:** If A relates to B, and B relates to C, then A relates to C.
**SKOS properties are NOT automatically transitive!** [file:5]
You must use:

- `skos:broaderTransitive`
- `skos:narrowerTransitive`

**Exam question type:** Given a SKOS hierarchy, determine if a transitive inference is valid.

#### 1.18.3 Disjoint Properties in SKOS

**Rule:** [file:5] `skos:related` is disjoint with both `skos:broader` and `skos:narrower`.
**What does disjoint mean?** Two properties cannot both hold between the same pair of concepts.
**Inconsistent example:**

```
:r1 skos:related :r3 .
:r1 skos:narrower :r3 .
```

This is a contradiction! (related and narrower are disjoint)

#### 1.18.4 SKOS Matching Properties

| **Property** | **Meaning** |
|---|---|
| `skos:exactMatch` | Concepts are essentially identical |
| `skos:closeMatch` | Concepts are similar (not quite identical) |
| `owl:sameAs` | Concepts are literally the same thing |

**Use case:** Aligning vocabularies from different sources. [file:5]

#### 1.18.5 Common SKOS Mistake

[file:5]
**WRONG:**

```
Europe rdfs:subClassOf UK .
```

**Problem:** Europe is an *instance*, not a class!
**CORRECT:**

```
Continent rdf:type skos:Concept .
Country rdf:type skos:Concept .
Country skos:broader Continent .

Europe rdf:type Continent .
UK rdf:type Country .
UK skos:broader Europe .
```

### 1.19 Linked Open Vocabularies (LOV)

**Resource:** https://lov.linkeddata.es/dataset/lov/ [file:5]

Repository of reusable RDF/RDFS vocabularies, searchable by category.

**Exam tip:** Know that LOV exists as a central registry for finding existing vocabularies before creating your own.

### 1.20 Limitations of RDFS (Why We Need OWL)

[file:5]

RDFS cannot express:

1. **Localized range/domain:**
   - Example: `hasChild` should return Person for Person, Elephant for Elephant.
   - RDFS only allows one global domain/range per property.
2. **Cardinality constraints:**
   - Example: Every person has exactly 2 biological parents.
   - RDFS cannot enforce "exactly 2".
3. **Property characteristics:**
   - Transitive: If A ancestor of B, B ancestor of C, then A ancestor of C.
   - Symmetric: If A sibling of B, then B sibling of A.
   - Inverse: `hasPart` is inverse of `isPartOf`.
   - RDFS lacks these!
4. **Reasoning support:**
   - No native reasoners for RDFS.

**Solution:** OWL (Web Ontology Language) addresses all these limitations!

### 1.21 Exam Checklist – Lecture 21

Be able to:

1. Explain the difference between Web of Documents and Web of Data.
2. Draw a knowledge graph from a textual description.
3. Write RDF triples (subject, predicate, object) for a given graph.
4. Convert between RDF/XML and Turtle formats.
5. Identify when to use Bag, Seq, Alt, or Collection.
6. Recognize and use blank nodes in Turtle (square brackets).
7. Define RDFS classes: `rdfs:Class`, `rdfs:Resource`, `rdfs:Literal`.
8. Apply RDFS properties: `subClassOf`, `domain`, `range`.
9. Perform simple RDFS inference (type inheritance, domain/range).
10. Explain SKOS concepts: `broader`, `narrower`, `related`.
11. State the limitations of RDFS (why OWL is needed).

# 2 Lecture 22 – Introduction to OWL

## 2.1 Big picture and exam link

This lecture introduces **OWL (Web Ontology Language)**, the most expressive semantic web language. Exam questions will test:

- Understanding what ontologies are and why we need them.
- OWL components: classes, properties (object & data), individuals.
- Class characteristics: `equivalentClass`, `disjointWith`.
- Property characteristics: `domain`, `range`, `inverseOf`, `symmetric`, `transitive`, `functional`.
- Creating simple ontologies in Protege.
- Understanding reasoning tasks: consistency, satisfiability, classification, realization.
- Ontology development process.

**Key exam mindset:** You must understand:

1. What makes OWL more powerful than RDFS.
2. How to define classes, properties, and individuals in OWL.
3. What reasoners can infer from OWL axioms.
4. The iterative ontology development process.

## 2.2 Recap: Why Do We Need OWL?

**Problems with RDF:** [file:6]

- No types, no restrictions on terms.
- Anyone can use any property on any resource.

**Problems with RDFS:** [file:6]

- Allows creating vocabularies but lacks:
  - Range/domain restrictions (global only).
  - Cardinality constraints.
  - Property characteristics (transitive, symmetric, inverse).

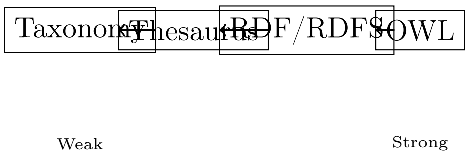
**Problems with SKOS:** [file:6]

- Builds taxonomies and thesauri.
- Limited expressiveness and reasoning.

**Solution:** Need a language with **strong semantics** → **OWL!** [file:6]

## 2.3 The Semantic Spectrum

**From weak to strong semantics:** [file:6]



**Expressiveness increases** from left to right.

## 2.4 What is an Ontology?

**Three levels of meaning:** [file:6]

1. **Metadata:** Data describing content and meaning of resources.
   - Requires parties to use the same *terms*.
2. **Terminologies:** Shared common vocabularies.
   - Requires parties to use the same *meaning*.
3. **Ontologies:** Shared common knowledge of a domain.
   - Meant for exchange, search, discovery.
   - Includes *concepts, relationships, and constraints*.

### 2.4.1 Formal Definition

**Gruber (1993):** [file:6]
"An ontology is an **explicit specification of a conceptualization**."
**In simpler terms:**
An ontology describes:

- Words, concepts, and relationships used to represent knowledge in a specific domain.

### 2.4.2 Forms of Ontologies

[file:6]
Different forms with increasing semantic power:

1. **Taxonomy:** Structured vocabulary (relational model).
2. **Thesaurus:** Words and synonyms (entity-relationship).
3. **Conceptual models:** Complex knowledge with RDF/RDFS.
4. **Logical theory:** Rich, consistent knowledge with OWL, Description Logic, First-Order Logic.

**Well-formed ontology:** Expressed in a well-defined formalism that is machine-interpretable. [file:6]

## 2.5 The Meaning Triangle

$$\begin{tikzcd}
 & \text{Thought} \arrow[dl, <->, "\text{subjective}"'] \arrow[dr, <->, "\text{subjective}"] & \\
\text{Symbol} \arrow[rr, dash, dashed, "\text{objective}"'] & & \text{Referent}
\end{tikzcd}$$

(I want to drink camel milk)

C.a.m.e.l    Arabic: jamal

**Key insight:** [file:6]

- Same *thought* (concept) can have different *symbols* (words).
- Same *symbol* can refer to different *referents* (things).
- Ontologies provide *objective* mapping between symbols and concepts.

## 2.6 Ontology Modeling Components

[file:6]

An ontology explicitly describes a domain using:

1. **Concepts (Classes):** Sets, types, categories.
   - Examples: Person, Animal, Course, Vehicle.
2. **Properties of concepts:**
   - **Data properties:** Attributes with literal values (age, name, weight).
   - **Object properties:** Relationships between concepts (teaches, livesIn, partOf).
3. **Constraints (Axioms):**
   - Type restrictions (age is integer).
   - Range restrictions (age between 0 and 120).
   - Cardinality (exactly 2 parents).
4. **Individuals (Instances):** Actual things.
   - Examples: John, Alice, HeriotWatt, Dubai.

**Terminology:** [file:6]

- **Ontology:** Schema + individuals = **Knowledge Base**.
- **TBox (Terminological Box):** Classes, properties (schema).
- **ABox (Assertional Box):** Individuals and their relationships (data).

## 2.7 Relational Schema vs Ontology

| **Relational Schema** | **Ontology** |
|---|---|
| Organizes data into databases | Shares information with semantics |
| Relationships are implicit (in code or human mind) | Relationships defined formally (logical constructs) |
| No semantics $\rightarrow$ humans/programs need context | Interpretation possible by humans AND machines |
| Closed-world assumption | Open-world assumption |

## 2.8 OWL: Web Ontology Language

**What is OWL?** [file:6]

OWL is a knowledge modeling language with these requirements:

1. Extends existing web standards (XML, RDF, RDFS).
2. Easy to understand and use (familiar to developers).
3. Formally specified (precise meaning).
4. High expressive power.
5. Provides automated reasoning support.

**What is OWL for?** [file:6]

OWL is primarily concerned with:

- Defining terminology for RDF documents.
- Specifying classes and properties with their characteristics.
- Defining individuals (instances) conforming to the ontology.

## 2.9 OWL Document Structure

**An OWL document is an RDF document!** [file:6]
**Typical structure:**

1. **Metadata:** Ontology IRI, version, authors, description.
2. **Namespaces:** Prefixes for external vocabularies.
3. **Class hierarchy:** Definitions and subclass relationships.
4. **Object property hierarchy:** Relationships between classes.
5. **Data property definitions:** Attributes with datatypes.
6. **Property characteristics:** Transitive, symmetric, functional, etc.
7. **Individuals:** Instances with their properties.

## 2.10 OWL Class Relationships

### 2.10.1 1. owl:equivalentClass

**Meaning:** Two classes refer to the same set of individuals. [file:6]
**Turtle example:**

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

**Interpretation:** Every Human is a Person, and every Person is a Human.

### 2.10.2 2. owl:disjointWith

**Meaning:** Individuals cannot belong to both classes simultaneously. [file:6]
**Example:**

```
:Man owl:disjointWith :Woman .
```

**Interpretation:** If someone is a Man, they cannot be a Woman (and vice versa).
**Multiple disjoint classes:** [file:6]

```
[ rdf:type owl:AllDisjointClasses ;
  owl:members ( :C1 :C2 :C3 ) ] .
```

## 2.11 OWL Properties

### 2.11.1 Object Properties vs Data Properties

| **Type** | **Object Property** | **Data Property** |
|---|---|---|
| Connects | Class to Class | Class to Literal |
| Range | Another class (IRI) | Datatype (xsd:int, xsd:string) |
| Example | hasWife, livesIn, teaches | hasAge, hasName, hasWeight |

### 2.11.2 Creating Object Properties

[file:6]

**Best practice:** Use *verbs* to avoid confusion with classes.

**Example:**

```
:marriedTo rdf:type rdf:Property .
:hasHusband rdf:type owl:ObjectProperty .
:hasWife rdf:type owl:ObjectProperty .
```

### 2.11.3 Property Hierarchy

**Turtle syntax:** [file:6]

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

**Meaning:** Every `hasWife` relationship implies a `hasSpouse` relationship.

### 2.11.4 Domain and Range

[file:6]

**Complete example:**

```
:hasWife rdf:type owl:ObjectProperty ;
  rdfs:subPropertyOf :hasSpouse ;
  rdf:type owl:AsymmetricProperty ;
  rdfs:domain :Man ;
  rdfs:range :Woman .
```

**Interpretation:**

- Subject of `hasWife` must be a `Man`.
- Object of `hasWife` must be a `Woman`.
- `hasWife` is asymmetric (if A hasWife B, then NOT B hasWife A).

## 2.12 OWL Property Characteristics (EXAM CRITICAL!)

### 2.12.1 1. owl:inverseOf

**Definition:** If A relates to B via property P, then B relates to A via inverse property Q. [file:6]

**Example:**

```
:hasWife owl:inverseOf :hasHusband .

John :hasWife Mary .
==> Inference:
Mary :hasHusband John .
```

### 2.12.2 2. owl:SymmetricProperty

**Definition:** If A relates to B, then B relates to A (same property).

**Example:**

```
:sibling rdf:type owl:SymmetricProperty .

Alice :sibling Bob .
==> Inference:
Bob :sibling Alice .
```

### 2.12.3 3. owl:AsymmetricProperty

**Definition:** If A relates to B, then B cannot relate to A.
**Example:**

```
:hasChild rdf:type owl:AsymmetricProperty .

John :hasChild Mary .
==> NOT possible:
Mary :hasChild John .
```

### 2.12.4 4. owl:TransitiveProperty

**Definition:** If A relates to B, and B relates to C, then A relates to C.
**Example:**

```
:ancestorOf rdf:type owl:TransitiveProperty .

John :ancestorOf Mary .
Mary :ancestorOf Alice .
==> Inference:
John :ancestorOf Alice .
```

### 2.12.5 5. owl:FunctionalProperty

**Definition:** Each individual can have at most one value for this property.
**Example:**

```
:hasBirthdate rdf:type owl:FunctionalProperty .

John :hasBirthdate "1990-01-01" .
John :hasBirthdate "1991-02-02" .
==> Inconsistency! (Can't have two birthdates)
```

### 2.12.6 6. owl:InverseFunctionalProperty

**Definition:** If two individuals share the same value, they must be the same individual.
**Example:**

```
:hasSSN rdf:type owl:InverseFunctionalProperty .

Person1 :hasSSN "123-45-6789" .
Person2 :hasSSN "123-45-6789" .
==> Inference:
Person1 owl:sameAs Person2 .
```

**Exam tip:** Memorize these six property characteristics! They appear frequently in reasoning questions.

## 2.13 Data Properties

**Purpose:** Attributes describing individuals with literal values. [file:6]
**Example:**

```
:hasAge rdf:type owl:DatatypeProperty ;
  rdfs:domain :Person ;
  rdfs:range xsd:int .

:hasName rdf:type owl:DatatypeProperty ;
  rdfs:domain :Person ;
  rdfs:range xsd:string .
```

## 2.14 Adding Individuals

**Step 1:** Define type (class). [file:6]
**Step 2:** Assert properties.
**Example:**

```
:Mary rdf:type owl:NamedIndividual , :Woman .

:John rdf:type owl:NamedIndividual , :Man ;
  :hasWife :Mary .
```

**Automatic inference:** [file:6]
If `hasWife` has inverse `hasHusband`:

```
==> Reasoner infers:
:Mary :hasHusband :John .
```

## 2.15 Reasoning with OWL

**What is a reasoner?** [file:6]
A reasoner is an algorithm that applies local inference to discover:

1. **Inconsistencies:** Contradictions in the ontology or data.
2. **Subsumptions:** Class inheritance hierarchy.
3. **Realizations:** Inferring the types of individuals.
4. **New relationships:** Using property characteristics (inverse, symmetric, transitive).

### 2.15.1 Reasoning Tasks

[file:6]
**1. Consistency Checking**
**Question:** Are there contradictions?
**Checks:**

- ABox does not contradict TBox.
- Individuals comply with ontology constraints.

**Example inconsistency:**

```
:Man owl:disjointWith :Woman .
:John rdf:type :Man , :Woman .
==> INCONSISTENT!
```

**2. Concept Satisfiability**
**Question:** Can a class have individuals?
**Example unsatisfiable class:**

```
:Parent rdfs:subClassOf :Person .
:Parent rdfs:subClassOf [
  a owl:Restriction ;
  owl:onProperty :hasChild ;
  owl:minCardinality 1
] .

:ChildlessPerson rdfs:subClassOf :Person .
:ChildlessPerson rdfs:subClassOf [
  a owl:Restriction ;
  owl:onProperty :hasChild ;
  owl:maxCardinality 0
] .

:ImpossibleClass rdfs:subClassOf :Parent , :ChildlessPerson .
==> UNSATISFIABLE! (Can't have >=1 and <=0 children)
```

**3. Classification**
**Question:** Does class A subsume (include) class B?
**Task:** Build full inheritance hierarchy.
**4. Realization**
**Question:** What class does individual X belong to?
**Task:** Compute direct types of each individual.
**Example:**

```
:John :hasChild :Mary .
:hasChild rdfs:domain :Parent .
==> Inference:
:John rdf:type :Parent .
```

## 2.16 Protege: OWL Ontology Editor

**What is Protege?** [file:6]
Protege is an open-source ontology editor for creating OWL ontologies with a graphical interface.
**Download:** https://protege.stanford.edu

### 2.16.1 Steps to Create an Ontology in Protege

[file:6]
**1. Creating an ontology**

1. Set IRI (ontology identifier).
2. Add metadata (title, author, version).
3. Define prefixes (namespaces).
4. Save ontology.
5. Select preferred format (RDF/XML, Turtle, OWL/XML).

**2. Create class hierarchy**

- Use meaningful names (nouns for classes).
- Create subclass relationships.
- Add annotations (labels, comments).
- Set characteristics (e.g., disjoint classes).

**3. Create object properties**

- Use verbs to avoid ambiguity.
- Add characteristics (symmetric, transitive, inverse).
- Define domain and range.

**4. Create data properties**

- Define attributes with datatypes.
- Set domain and range.

**5. Add individuals**

- Create instances of classes.
- Assert properties.

**6. Run reasoner**

- Check consistency.
- View inferred information (shown in yellow in Protege).

## 2.17 VOWL: Visual Notation for OWL

**What is VOWL?** [file:6]
VOWL (Visual Notation for OWL) defines a visual language for user-oriented representation of ontologies.
**Key symbols:**

- **Circles:** Classes
- **Solid arrows:** Object properties
- **Dashed arrows:** Subclass relationships
- **Rectangles:** Datatypes
- **Dotted arrows:** Data properties

## 2.18 Ontology Development Process

[file:6]
**Idealized linear process:**

1. Determine scope and domain.
2. Consider reusing existing ontologies.
3. Enumerate important terms.
4. Define classes and class hierarchy.
5. Define properties of classes.
6. Define constraints on properties.
7. Create instances.

**Reality: Iterative process!** [file:6]
You constantly revisit earlier stages based on new insights.

- Define classes → realize you need more terms → go back
- Add properties → realize classes need restructuring → iterate
- Create instances → discover inconsistencies → refine ontology

## 2.19 Principles When Creating an Ontology

[file:6]

1. **No unique perfect model:** Different applications need different ontologies.
2. **Application-driven:** Model should serve your specific application.
3. **Iterative refinement:** Cannot create perfect ontology in one pass.
4. **Close to reality:** Classes and properties should match real/physical objects.
5. **Naming conventions:**
   - Nouns for classes (Person, Vehicle, Course).
   - Verbs for properties (teaches, livesIn, hasChild).

## 2.20 Rules for FAIR Vocabularies

[file:6]

**FAIR = Findable, Accessible, Interoperable, Reusable**
**10-step deployment checklist:**

1. Provide open license allowing repurposing.
2. Determine content governance arrangements.
3. Check minimal term definition completeness.
4. Select domain and service for web identifiers (namespace).
5. Design identifier schema and pattern.
6. Create semantic-standards based vocabulary (RDF/OWL).
7. Add rich metadata (descriptions, examples).
8. Register vocabulary (e.g., with LOV).
9. Make IRIs resolvable (return useful content when accessed).
10. Implement vocabulary maintenance process.

### 2.20.1 Semantic-Standards Compliant Vocabulary

1. Identify terms.
2. Encode term labels and synonyms.
3. Add textual definitions.
4. Add notes/comments for clarification.
5. Add codes and symbols.
6. Define hierarchy of terms.
7. Encode relationships.
8. Define subsets.
9. Define and document the whole vocabulary.

### 2.21 Exam Checklist – Lecture 22

Be able to:

1. Explain what an ontology is (Gruber's definition).
2. Differentiate between TBox (schema) and ABox (instances).
3. List ontology components: classes, properties, individuals, constraints.
4. Distinguish between object properties and data properties.
5. Define `equivalentClass` and `disjointWith`.
6. Explain property characteristics:
    - inverseOf, symmetric, asymmetric, transitive, functional, inverseFunctional
7. Describe the four reasoning tasks:
    - Consistency checking
    - Concept satisfiability
    - Classification
    - Realization
8. Outline the ontology development process (including iteration).
9. State principles for creating ontologies (nouns for classes, verbs for properties).

## 3 Ultra-Short Revision Notes

### 3.1 Lecture 21: Knowledge Representation

**RDF Triple:** (Subject, Predicate, Object)

- Subject: IRI of resource
- Predicate: IRI of property
- Object: IRI or literal

**RDF Containers:**

- `rdf:Bag` - unordered
- `rdf:Seq` - ordered
- `rdf:Alt` - alternatives
- `rdf:Collection` - closed list

**Blank nodes:** Anonymous nodes (no IRI), use `[ ]` in Turtle.
**RDFS Classes:**

- `rdfs:Class`, `rdfs:Resource`, `rdfs:Literal`, `rdfs:Property`

**RDFS Properties:**

- `rdfs:subClassOf` - class hierarchy (transitive)
- `rdfs:domain` - subject type
- `rdfs:range` - object type

- `rdfs:subPropertyOf` - property hierarchy

**RDFS Inference:**

- Type inheritance: X subClassOf Y → instances of X are instances of Y
- Domain/range: property usage infers types

**SKOS:**

- `skos:broader` - more general (parent)
- `skos:narrower` - more specific (child)
- `skos:related` - associative (non-hierarchical)
- Disjoint: related cannot be broader/narrower

**RDFS Limitations:**

- No localized range/domain
- No cardinality constraints
- No property characteristics (transitive, symmetric, inverse)

### 3.2 Lecture 22: OWL

**Ontology Components:**

- Classes (concepts)
- Object properties (relationships between classes)
- Data properties (attributes with literals)
- Individuals (instances)
- Constraints (axioms)

**TBox vs ABox:**

- TBox: Ontology (classes, properties, constraints)
- ABox: Data instances (individuals)

**Class Relationships:**

- `owl:equivalentClass` - same individuals
- `owl:disjointWith` - no shared individuals

**Property Characteristics:**

- `owl:inverseOf` - A P B → B Q A
- `owl:SymmetricProperty` - A P B → B P A
- `owl:AsymmetricProperty` - A P B → NOT (B P A)
- `owl:TransitiveProperty` - A P B, B P C → A P C
- `owl:FunctionalProperty` - at most one value
- `owl:InverseFunctionalProperty` - same value → same individual

**Reasoning Tasks:**

1. Consistency: no contradictions
2. Satisfiability: can class have individuals?
3. Classification: build class hierarchy
4. Realization: infer individual types

**Ontology Development:**

1. Determine scope
2. Reuse existing ontologies
3. Enumerate terms
4. Define classes (nouns)
5. Define properties (verbs)
6. Define constraints
7. Create instances
8. **Iterate!**

## 4 Worked Examples

### 4.1 Example 1: Converting Knowledge Graph to RDF

**Scenario:** Alice is a student at Heriot-Watt University. She studies Computer Science and lives in Dubai.

**Step 1: Identify entities and relationships**

- Entities: Alice, Student, HeriotWatt, ComputerScience, Dubai
- Relationships: isA, studiesAt, studies, livesIn

**Step 2: Write as triples**

1. Alice isA Student
2. Alice studiesAt HeriotWatt
3. Alice studies ComputerScience
4. Alice livesIn Dubai

**Step 3: Turtle format**

```
@prefix ex: <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

ex:Alice rdf:type ex:Student ;
  ex:studiesAt ex:HeriotWatt ;
  ex:studies ex:ComputerScience ;
  ex:livesIn ex:Dubai .
```

### 4.2 Example 2: RDFS Inference

**Given:**

```
:Cat rdfs:subClassOf :Mammal .
:Mammal rdfs:subClassOf :Animal .
:Whiskers rdf:type :Cat .
```

**Question:** What can we infer?
**Answer:**
By transitivity of `subClassOf`:

```
:Cat rdfs:subClassOf :Animal .
```

By type inheritance:

```
:Whiskers rdf:type :Mammal .
:Whiskers rdf:type :Animal .
```

### 4.3 Example 3: OWL Property Inference

**Given:**

```
:hasParent rdf:type owl:TransitiveProperty .
:hasParent owl:inverseOf :hasChild .

:Alice :hasParent :Bob .
:Bob :hasParent :Carol .
```

**Question:** What can reasoner infer?
**Answer:**
By transitivity:

```
:Alice :hasParent :Carol .
```

By inverse:

```
:Bob :hasChild :Alice .
:Carol :hasChild :Bob .
:Carol :hasChild :Alice .
```

### 4.4 Example 4: Detecting Inconsistency

**Ontology:**

```
:Male owl:disjointWith :Female .
:hasFather rdfs:domain :Person ;
  rdfs:range :Male .
```

**Data:**

```
:Alice rdf:type :Person ;
  :hasFather :Bob .
:Bob rdf:type :Female .
```

**Question:** Is this consistent?
**Answer:** NO!
**Reasoning:**

1. From `:Alice :hasFather :Bob`, and range of `hasFather` is `Male`:

```
:Bob rdf:type :Male .
```

2. But we also stated:

```
:Bob rdf:type :Female .
```

3. Since `Male` and `Female` are disjoint, Bob cannot be both!
4. **INCONSISTENT!**

## 5 Final Summary

**Week 2 covered the foundation of semantic web knowledge representation:**

1. **Lecture 21:** RDF, RDFS, SKOS
   - RDF triples and serialization formats
   - RDF containers (Bag, Seq, Alt, Collection)
   - RDFS vocabulary (classes, properties, domain, range)
   - RDFS inference (type inheritance, domain/range)
   - SKOS for taxonomies and thesauri
   - Limitations requiring OWL
2. **Lecture 22:** OWL
   - Ontologies and their components
   - OWL classes and relationships (equivalentClass, disjointWith)
   - Object and data properties
   - Property characteristics (inverse, symmetric, transitive, functional)
   - Reasoning tasks (consistency, satisfiability, classification, realization)
   - Ontology development process
   - Protege ontology editor

   **Key exam strategy:**

- Master RDF triple notation (both RDF/XML and Turtle).
- Memorize RDFS and OWL property characteristics.
- Practice inference examples.
- Be able to detect inconsistencies.
- Understand when to use RDFS vs OWL.

**You're ready for Week 2 exam questions!**

Practice converting between formats, performing inference, and detecting inconsistencies!