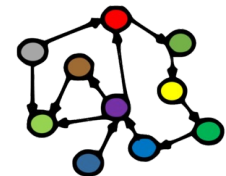


Welcome to INFO216:  
Knowledge Graphs  
Spring 2023

Andreas L Opdahl  
<Andreas.Opdahl@uib.no>

# Session 7: Rules (SHACL and RDFS)

- Themes:
  - why SHACL (SHapes Constraint Language)?
    - node and property constraints
  - why RDFS (RDF Schema)?
    - utility properties
    - classes and subclasses
    - properties and subproperties
    - entailments and axioms
  - motivation for OWL (the Web Ontology Language)



# Readings

- Sources:
  - Allemang, Hendler & Gandon (2020):  
**Semantic Web for the Working Ontologist**, 3<sup>rd</sup> edition:  
chapters 7-8, but chapter 6-7 in the 2<sup>nd</sup> edition (no SHACL)
  - Chapter 5: SHACL in **Validating RDF** (available online)
  - Blumauer & Nagy (2020):  
The Knowledge Graph Cookbook – Recipes that Work:  
e.g., pages 101-106, 136-139 (*supplementary*)
- Resources in the wiki <<http://wiki.uib.no/info216>>:
  - Interactive SHACL Playground
  - W3C's RDF Schema 1.1 (sections 1-3 and 6)
  - Shapes Constraint Language (SHACL) (Editor's Draft)

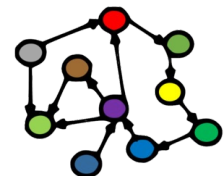


THE KNOWLEDGE GRAPH  
**COOKBOOK**  
RECIPES THAT WORK

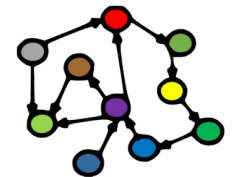


ANDREAS BLUMAUER  
AND HELMUT NAGY

1st edition, 2020

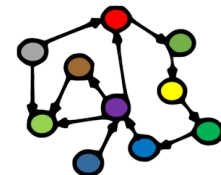


# SHACL



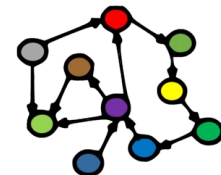
# SHACL (SHapes Constraint Language)

- Constraining the structure of RDF (and RDFS, OWL...) graphs
  - another W3C standard (from around 2018)
- Used to validate:
  - open and other KGs we want to *reuse*
  - graphs resulting from *user input*
  - the KGs *we make ourselves*
- *SHACL constraints are written in RDF*
  - a shapes graph is used to validate
  - a data graph
- *SHACL Core* and several extensions
  - we will focus on *SHACL Core*

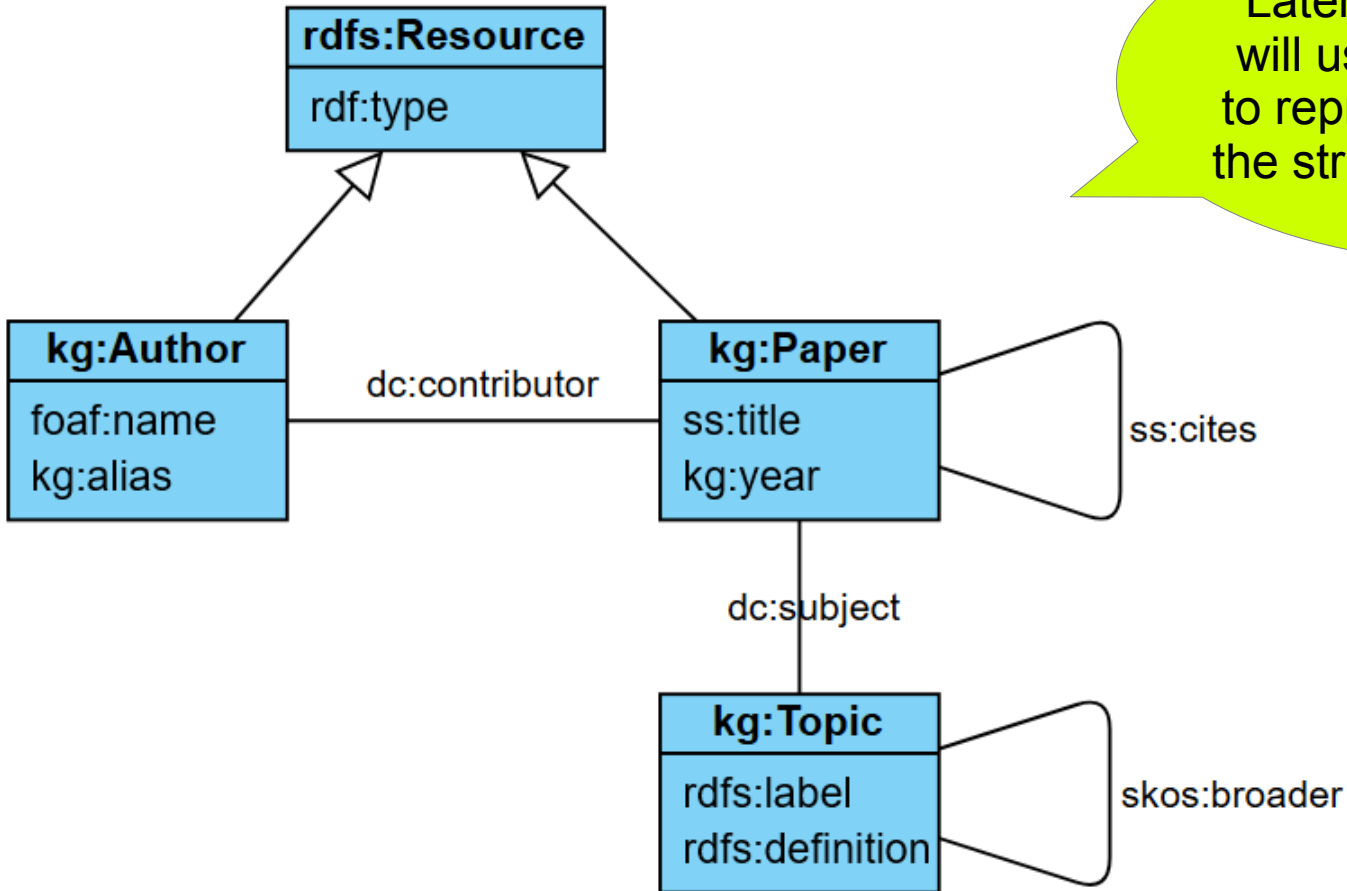


## Example KG (←S03)

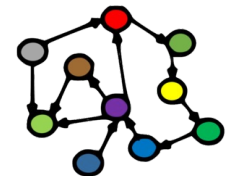
- A knowledge graph of research literature related to “Knowledge Graphs for the News”
  - built to support a recent literature study
  - 78 main papers with 291 authors
  - 4086 other papers with 8990 authors
  - 100s of topics and themes, >300k triples
- Accessible at <http://bg.newsangler.uib.no/>
  - runs on a Blazegraph triple store
  - Blazegraph’s simple web front end, read only



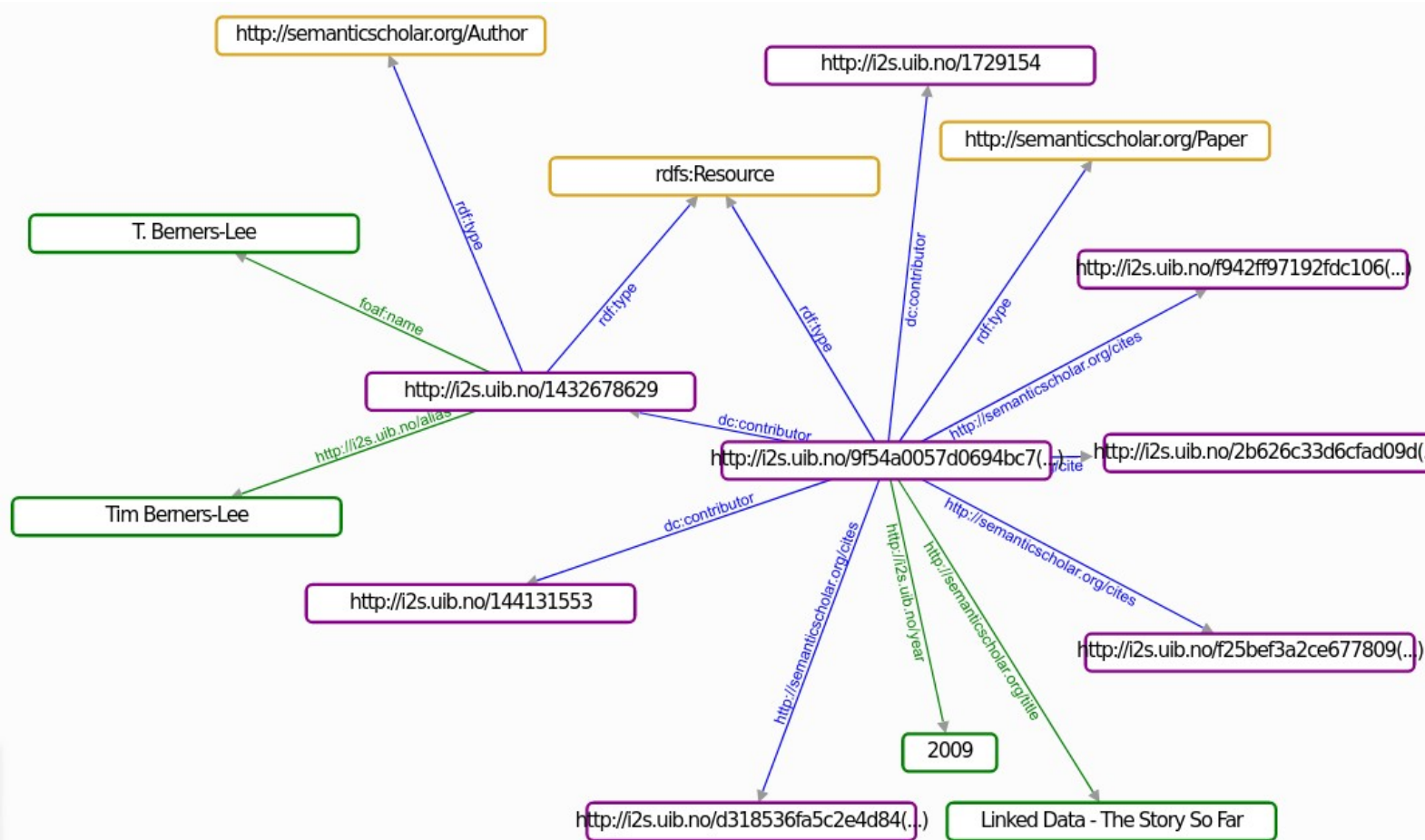
# Example KG: graph structure (←S03)



Later in the course, we will use RDFS and OWL to represent and visualise the structure of ontologies.

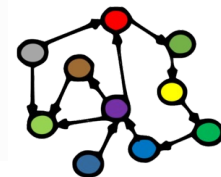


# Example KG: resources (←S03)



(The URIs are simplified.)

(c) Andreas L Opdahl, 2023

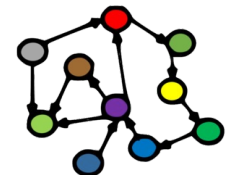




# Example constraints

- Every main paper is the subject of exactly one year property.

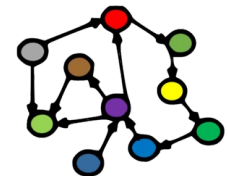
```
kg:MainPaperShape
  a sh:NodeShape ;
  sh:targetClass kg:MainPaper ;
  sh:property [
    sh:path kg:year ;
    sh:minCount 1 ;
    sh:maxCount 1
  ] .
```



# Example constraints

- Every main paper is the subject of exactly one year property.
- Every year value (literal object) of a main paper is an integer.

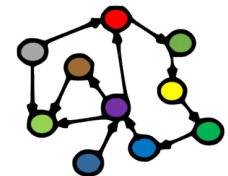
```
kg:MainPaperShape
  a sh:NodeShape ;
  sh:targetClass kg:MainPaper ;
  sh:property [
    sh:path kg:year ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
    sh:datatype xsd:integer
  ] .
```



# Example constraints

- Every main paper has at least one contributor

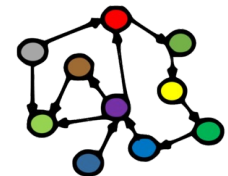
```
kg:MainPaperShape
  a sh:NodeShape ;
  sh:targetClass kg:MainPaper ;
  sh:property [
    sh:path dcterms:contributor ;
    sh:minCount 1
  ] .
```



# Example constraints

- Every main paper has at least one contributor
  - who is a main author

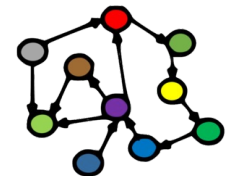
```
kg:MainPaperShape
  a sh:NodeShape ;
  sh:targetClass kg:MainPaper ;
  sh:property [
    sh:path dcterms:contributor ;
    sh:minCount 1 ;
    sh:class kg:MainAuthor
  ] .
```



# Example constraints

- Every main paper has at least one contributor
  - who is a main author
  - and is represented by a URI

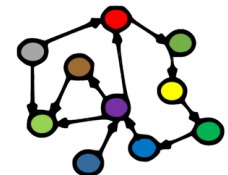
```
kg:MainPaperShape
  a sh:NodeShape ;
  sh:targetClass kg:MainPaper ;
  sh:property [
    sh:path dcterms:contributor ;
    sh:minCount 1 ;
    sh:class kg:MainAuthor ;
    sh:nodeKind sh:IRI
  ] .
```



# Example constraints

- Every main paper has at least one subject
  - whose value is a SKOS concept ( $\rightarrow$ S09) which is represented by a URI

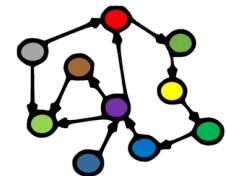
```
kg:MainPaperShape
  a sh:NodeShape ;
  sh:targetClass kg:MainPaper ;
  sh:property [
    sh:path dcterms:subject ;
    sh:minCount 1 ;
    sh:class skos:Concept ;
    sh:nodeKind sh:IRI ;
  ] .
```



# Example constraints

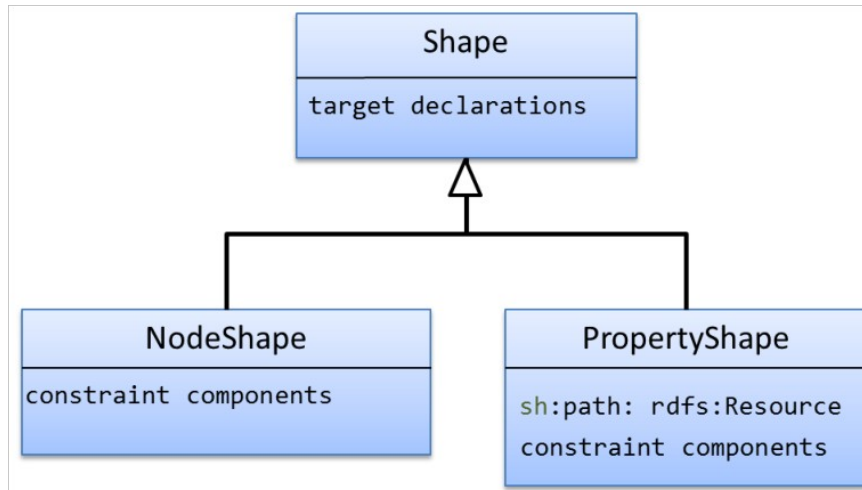
- Every main paper has at least one subject
  - whose value is either
    - a theme,
    - a topic

```
kg:MainPaperShape
  a sh:NodeShape ;
  sh:targetClass kg:MainPaper ;
  sh:property [
    sh:path dcterms:subject ;
    sh:minCount 1 ;
    sh:or ( [sh:class th:Theme]
            [sh:class ss:Topic] ) ;
    sh:nodeKind sh:IRI ;
  ] .
```

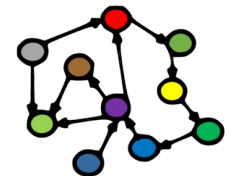


# SHACL constraint structure

- SHACL constrains
  - node shapes
  - property shapes
- The node shapes can act as collections of property shapes belonging to the same class



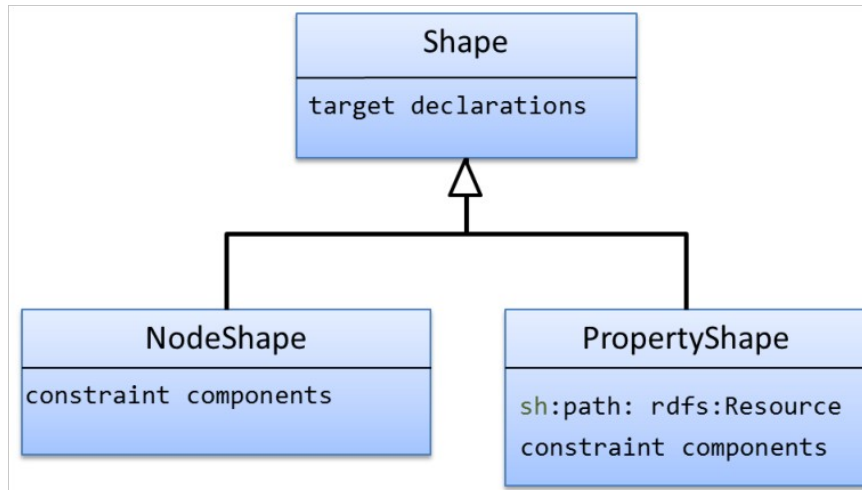
```
kg:MainPaperShape
  a sh:NodeShape ;
  sh:targetClass kg:MainPaper ;
  sh:property [
    sh:path dcterms:subject ;
    sh:minCount 1 ;
    sh:or ( [sh:class th:Theme]
            [sh:class ss:Topic] ) ;
    sh:nodeKind sh:IRI ;
  ] .
```





# SHACL constraint structure

- SHACL constrains
  - node shapes
  - property shapes
- The node shapes are mostly collections of property shapes pertaining to the same class



kg:MainPaperShape

a sh:NodeShape ;

sh:targetClass kg:MainPaper ;

sh:property kg:SubjectShape .

kg:SubjectShape

a sh:PropertyShape ;

sh:path dcterms:subject ;

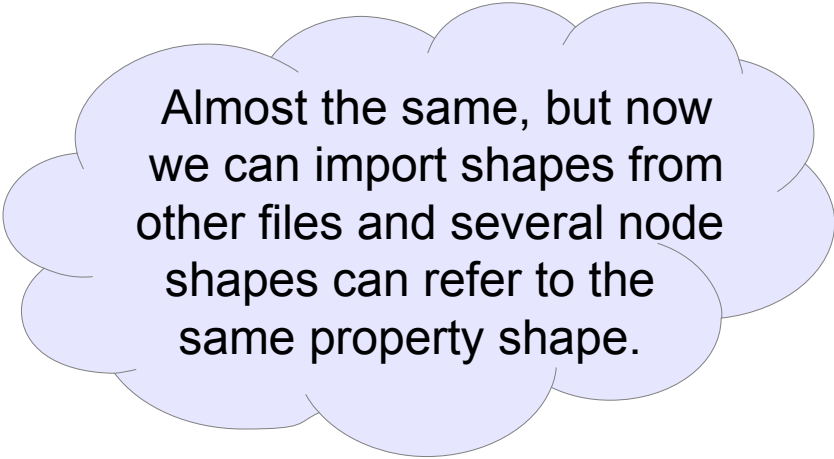
sh:minCount 1 ;

sh:or ( [sh:class th:Theme]  
[sh:class ss:Topic] ) ;

sh:nodeKind sh:IRI .

# SHACL constraint structure

- SHACL constrains
  - node shapes
  - property shapes
- The node shapes are mostly collections of property shapes pertaining to the same class



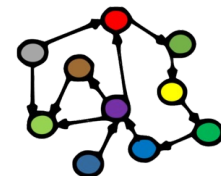
Almost the same, but now we can import shapes from other files and several node shapes can refer to the same property shape.

```
kg:MainPaperShape
  a sh:NodeShape ;
  sh:targetClass kg:MainPaper ;
  sh:property kg:SubjectShape .
```

```
kg:SubjectShape
  a sh:PropertyShape ;
  sh:path dcterms:subject ;
  sh:minCount 1 ;
  sh:or ( [sh:class th:Theme]
          [sh:class ss:Topic] ) ;
  sh:nodeKind sh:IRI .
```

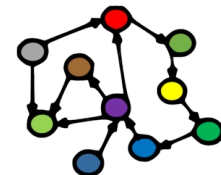
# Node shapes

- Node shapes specify constraints on focus nodes
  - `<node_shape_URI>` a `sh:NodeShape`
  - the focus nodes are often specified as `sh:targetClass <class_URI>`
    - the node constraints apply to all instances of the target class
    - alternatives: `sh:targetNode`, `sh:targetSubjectsOf`, `sh:targetObjectsOf`
  - constraints *on each focus node itself*:
    - `sh:class <class_URI>` or: `sh:datatype <datatype_URI>`
    - `sh:in ( ...list of URIs/values... )` or: `sh:hasValue ...URI/value...`
    - `sh:nodeKind` can be one of
      - `sh:IRI`, `sh:BlankNode`, `sh:Literal`, `sh:IRIOrLiteral`,  
`sh:BlankNodeOrIRI`, `sh:BlankNodeOrLiteral`
      - `sh:pattern <regular_expression>`



# Node shapes

- Node shapes specify constraints on focus nodes
  - constraints on properties *from the focus node* (with the focus node as *subject*):
    - by shape URI:  
`sh:property <property_shape_URI>`
    - by anonymous node:  
`sh:property [  
 a sh:PropertyShape ; # not necessary  
 sh:path <property_URI> ;  
 ... specific property constraints go here ...  
];`



# Property shapes

- Property shapes specify constraints about the values that can be reached from a focus node by some path
  - `<property_shape_URI> a sh:PropertyShape` # usually implicit
  - the property is often specified as `sh:path <property_URI>`
    - alternatively, SPARQL-like *property paths* can be used
    - the property constraints apply to:
      - all uses of the property path from the focus nodes
      - all values reached by the property path from the focus nodes
  - property constraints:
    - `sh:minCount`, `sh:maxCount`, ...
  - node constraints about the property *value* (the *object* resource or literal):
    - `sh:class`, `sh:datatype`, `sh:nodeKind`, `sh:pattern`, ...

# Property paths in SPARQL and SHACL

<i>SPARQL path...</i>	<i>...corresponds to SHACL path</i>
<i>schema:name</i>	schema:name
<i>^schema:knows</i>	[sh:inversePath schema:knows]
<i>schema:knows / schema:name</i>	(schema:knows schema:name)
<i>schema:knows   schema:follows</i>	[sh:alternativePath (schema:knows schema:follows)]
<i>schema:knows?</i>	[sh:zeroOrOnePath schema:knows]
<i>schema:knows+</i>	[sh:oneOrMorePath schema:knows]
<i>schema:knows* / schema:name</i>	([sh:zeroOrMorePath schema:knows] schema:name)

Examples use the schema.org vocabulary <<https://schema.org>>

# Validation reports

- Reports the results applying a SHACL shapes graph to a data graph
  - a `sh:ValidationReport`
  - three components:
    - `sh:conforms` (either true or false)
    - a `results_text` (from pySHACL)
    - zero or more `sh:ValidationResults`

# SHACL validation result properties

Result property	Explanation
<i>sh:focusNode</i>	The focus node that was being validated when the error happened.
<i>sh:resultPath</i>	The path from the focus node. This property is optional usually corresponds to the <i>sh:path</i> declaration of property shapes.
<i>sh:value</i>	The value that violated the constraint, when available .
<i>sh:sourceShape</i>	The shape that the focus node was validated against when the constraint was violated.
<i>sh:source ConstraintComponent</i>	The IRI that identifies the component that caused the violation.
<i>sh:detail</i>	May point to further details about the cause of the error. This property can be used for reporting errors in nested nested shapes.
<i>sh:resultMessage</i>	Textual details about the error. This message can be affected by the <i>sh:message</i> property.
<i>sh:resultSeverity</i>	A value which is equal to the <i>sh:severity</i> value of the shape that caused the violation error, if present. Otherwise the default value will be <i>sh:Violation</i> .



# There is a lot more...

- Logical expressions:
  - `sh:or`, `sh:and`, `sh:xone` (exactly one), and `sh:not`
- String and language constraints:
  - `sh:length`, `sh:minLength`, `sh:maxLength`, `sh:pattern`
  - `sh:uniqueLang`, `sh:languageIn`
- Value-range constraints on integers value nodes
- `sh:severity` of constraints
- Non-validating (informational) constraints:
  - `sh:name`, `sh:description`, `sh:order`, `sh:group`
- One shapes graph can `owl:imports` another

# Programming pySHACL

```
# pip install pyshacl

from pyshacl import validate
from rdflib import Graph

data_graph = Graph()
data_graph.parse('...')

shacl_str = """ ... """

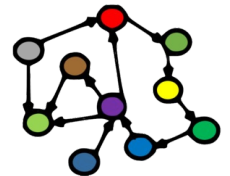
shacl_graph = Graph()
shacl_graph.parse(
    data=shacl_str, format='ttl'
)
```

```
results = validate(
    data_graph,
    shacl_graph=shacl_graph,
    inference='both'
)

(conforms,
 results_graph,
 results_text) = results

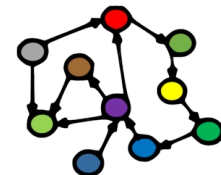
print(results_text)
```

# RDF Schema (RDFS)



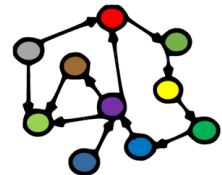
# Why RDF Schema (RDFS)

- *RDF is a good start*
  - excellent as a normal form for facts about individuals
  - less suitable for complex concept systems
    - e.g., vocabularies, ontologies
  - we sometimes need to represent:
    - more specific types of resources (subjects, objects)
    - more specific types of properties (predicates)
    - which types of resources that are the subjects and objects of which properties?
- *RDFS offers more expressive RDF graphs*



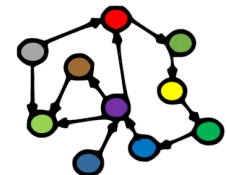
# Why RDF Schema (RDFS)

- *SHACL is excellent for checking RDFS graphs*
  - but what to do with the violations we discover?
  - two options:
    - *correct* them with *SPARQL Update*
    - *prevent* them with *RDFS rules*
    - more specific types of resources
- *RDFS defines*
  - *predefined* triples (*axioms*)
  - *rules* for how some triples can *entail* additional triples (*inference*)
- RDFS *inference engines* enforce the rules automatically



# Why RDF Schema (RDFS)

- *RDFS Schema (RDFS)*:
  - a small RDF vocabulary for more expressive graphs
    - particularly suitable for defining other vocabularies
  - underpinned by precise formal entailment rules
    - the rules define the *semantics* of RDFS
  - RDFS is expressed in RDF
    - many other vocabularies are defined in plain RDFS
    - RDFS is also the foundation for *OWL*
  - conventional prefix:
    - `rdfs: http://www.w3.org/2000/01/rdf-schema#`

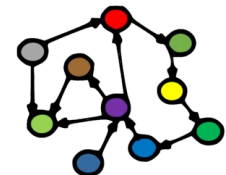


# RDFS resource classes (rdfs:Class)

- Classes are resources that represent a type of similar resources, which are the individuals in the class
  - e.g., `dbpedia:Person`, `schema:Person`, `foaf:Person`
  - class membership is expressed by the `rdf:type` property:  
`<RDF individual> rdf:type <RDFS class> .`  
`<RDF individual> a <RDFS class> .`
  - an individual can belong to several classes (this is common!)
  - RDFS classes are different from classes in typical OO programming
- *By convention, classes are named with an upper-case initial letter, properties with lower-case initial letters...*

# RDFS subclasses (rdfs:subClassOf)

- Whenever an individual resource belongs to some class, it necessarily belongs to another class too, e.g.,
  - *dbpedia:Politician rdfs:subClassOf dbpedia:Person* .
- Why subclasses?
  - the classes' semantics become more precise
  - more complete query answering
  - inferring additional information about resources (entailment)
  - subclasses are important because different vocabularies may define overlapping, but not identical, classes
    - introduce a new class in the merged data set
    - make the old classes subclasses of the new class



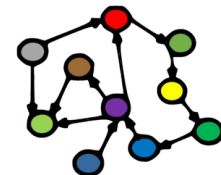


# RDFS entailment

- The meaning of `rdfs:subClassOf` is defined by an *entailment rule*...
- Example: classical *syllogism*
  - “*All men are mortal.*” (Major Premise)
  - “*Socrates is a man.*” (Minor Premise)

---

  - “*Socrates is a mortal.*” (Valid conclusion)
- *This rule is built into all RDFS models!*
  - RDFS defines several other rules (16 in total)
- Entailment means that:
  - many triples are there in our RDFS models *even when we have not added them*

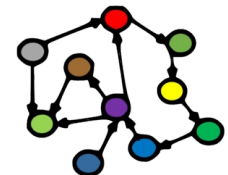


# RDFS entailment

- The meaning of `rdfs:subClassOf` is defined by an *entailment rule*...
- Example: *pattern* for classical syllogism in RDF
  - `?c1 rdfs:SubclassOf ?c2 .` (Major Premise)
  - `?s rdf:type ?c1 .` (Minor Premise)

---

  - `?s rdf:type ?c2 .` (Valid conclusion)
- *This rule is built into all RDFS models!*
  - it is called [rdfs9]



# RDFS entailment [rdfs9]

- The meaning of `rdfs:subClassOf` and the other RDFS concepts is defined by *entailment rules* [rdfs9]:
- The triples  
    ?`s` `rdf:type` ?`c1` .  
    ?`c1` `rdfs:subClassOf` ?`c2` .  
entail that  
    ?`s` `rdf:type` ?`c2` .

```
PREFIX ...  
INSERT {  
    ?s rdf:type ?c2 .  
} WHERE {  
    ?s rdf:type ?c1 .  
    ?c1 rdfs:subClassOf ?c2 .  
}
```

Here, we express the rule using SPARQL Update. But RDFS rules are not really implemented with SPARQL.

# What does entailment mean?

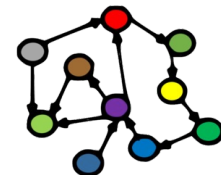
- Entailment means that many *triples are there in our RDFS graphs even when we have not asserted them*
  - 14 entailment rules in RDFS and 2 in RDF
  - full list at <http://www.w3.org/TR/rdf-mt/>
- *Different RDFS tools may support entailment rules in different ways*, e.g.:
  - strategy 1 (“eager”): always add entailed triples when possible
  - strategy 2 (“on demand”): only extract entailed triples when needed
  - we will use the OWL-RL API in the exercises:

```
import owlrl
```

```
engine = owlrl.RDFSClosure.RDFS_Semantics(graph, False, False, False)
```

```
engine.closure()
```

```
engine.flush_stored_triples()
```



<https://github.com/RDFLib/OWL-RL>

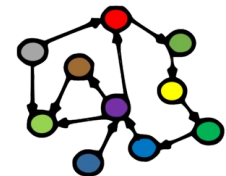
<https://owl-rl.readthedocs.io/en/latest/owlrl.html>

# Transitive properties

- `rdfs:subClassOf` is *transitive*:
  - *dbpedia:President rdfs:subClassOf dbpedia:Politician .*
  - *dbpedia:Politician rdfs:subClassOf dbpedia:Person .*

---

  - *dbpedia:President rdfs:subClassOf dbpedia:Person .*
- Entails new `rdf:type` triples about which classes an individual belongs to

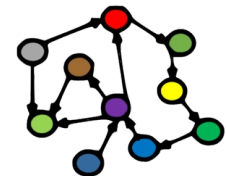


# Transitive properties

- `rdfs:subClassOf` is *transitive*:
  - `?c1 rdfs:subClassOf ?c2 .`
  - `?c2 rdfs:subClassOf ?c3 .`

---

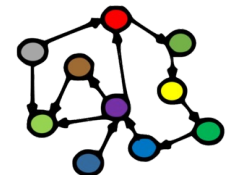
  - `?c1 rdfs:subClassOf ?c3 .`
- Entails new `rdf:type` triples about which classes an individual belongs to



# RDFS entailment [rdfs11]

- `rdfs:subClassOf` is *transitive*:
- The triples  
    `?c1 rdfs:subClassOf ?c2 .`  
    `?c2 rdfs:subClassOf ?c3 .`  
entail that  
    `?c1 rdfs:subClassOf ?c3 .`

```
PREFIX ...  
INSERT {  
    ?c1 rdfs:subClassOf ?c3 .  
} WHERE {  
    ?c1 rdfs:subClassOf ?c2 .  
    ?c2 rdfs:subClassOf ?c3 .  
}
```



# RDFS entailment [rdfs10]

- `rdfs:subClassOf` is also *reflexive*:
- The triple `?c rdf:type rdfs:Class .`  
entails that `?c rdfs:subClassOf ?c .`

PREFIX ...

INSERT {

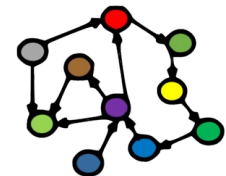
`?c rdfs:subClassOf ?c .`

} WHERE {

`?c rdf:type rdfs:Class .`

}

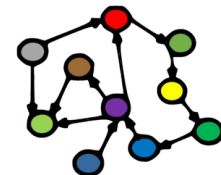
- “Every class is its own subclass...”





# RDF properties (rdf:Property)

- All predicates in a triple have *rdf:type rdf:Property*
  - this is expressed by an *entailment rule* (next slide!)
  - properties have *domains* and *ranges*
    - their subjects and objects always belong to specific classes
  - properties can be *transitive*
- Why properties?
  - similar to subclasses:
    - clearer semantics, entailment, complete answers to queries and defining other concepts, *e.g.*,
    - *most classes are defined by their properties...*



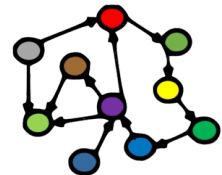
# RDF entailment [rdf1]

- The triple `?s ?p ?o .`  
entails that `?p rdf:type rdf:Property .`

PREFIX `rdf: <...>`

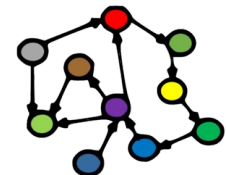
```
INSERT {  
    ?p rdf:type rdf:Property .  
} WHERE {  
    ?s ?p ?o .  
}
```

- Resources *become* properties when they are *used* as predicates in triples!



# Domain and range of properties

- The subjects and objects that occur in triples along with some property belong to certain classes
- Example:
  - *<subject>* `ex:presidentOf` *<object>* .
  - when we see this triple, we may know that:
    - the *<subject>* has `rdf:type dbpedia:President`
    - the *<object>* has `rdf:type dbpedia:Country`
  - this is part of the semantics of `ex:president`
    - in the context of the `ex:` vocabulary
  - ...can be expressed as follows:
    - `ex:presidentOf rdfs:domain dbpedia:President` .
    - `ex:presidentOf rdfs:range dbpedia:Country` .



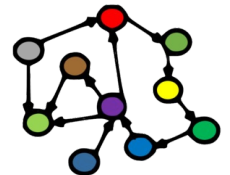
# RDFS entailment [rdfs2]

- The triples `?s ?p ?o .`  
`?p rdfs:domain ?t .`  
entail that `?s rdf:type ?t .`

PREFIX rdf: <...>

PREFIX rdfs: <...>

```
INSERT {  
    ?s rdf:type ?t .  
} WHERE {  
    ?s ?p ?o .  
    ?p rdfs:domain ?t .  
}
```



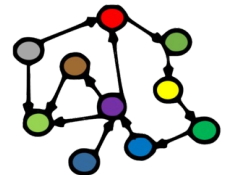
# RDFS entailment [rdfs3]

- The triples `?s ?p ?o .`  
`?p rdfs:range ?t .`  
entail that `?o rdf:type ?t .`

PREFIX rdf: <...>

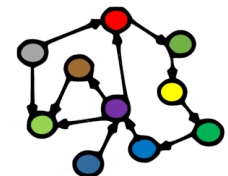
PREFIX rdfs: <...>

```
INSERT {  
    ?o rdf:type ?t .  
} WHERE {  
    ?s ?p ?o .  
    ?p rdfs:range ?t .  
}
```



# RDFS axioms

- RDFS axioms:
  - triples that are “built into” RDFS
  - always there in any RDFS graph
    - whether we have not added them or not
  - essential part of the semantics of RDFS
  - full list at <http://www.w3.org/TR/rdf-mt/>
  - 40 axioms and 3 axiom schemas
- Example *axioms* for *rdf:type*:
  - `rdf:type rdfs:domain rdfs:Resource .`
  - `rdf:type rdfs:range rdfs:Class .`



# RDFS entailment [rdfs3]

- The triples `?s ?p ?o .`  
`?p rdfs:range ?t .`  
entail that `?o rdf:type ?t .`

PREFIX rdf: <...>

PREFIX rdfs: <...>

```
INSERT {  
    ?o rdf:type ?t .  
} WHERE {  
    ?s ?p ?o .  
    ?p rdfs:range ?t .  
}
```



**Remember:**

`rdf:type rdfs:range rdfs:Class .`  
is an axiom in RDFS. This axiom fits  
straight into the rule:

`?p = rdf:type`

`?t = rdfs:Class`

# RDFS entailment [rdfs3 + axiom]

- The triples `?s rdf:type ?o .`  
`rdf:type rdfs:range rdfs:Class .`  
entail that `?o rdf:type rdfs:Class .`



```
PREFIX rdf: <...>  
PREFIX rdfs: <...>
```

```
INSERT {  
    ?o rdf:type rdfs:Class .  
} WHERE {  
    ?s rdf:type ?o .  
    rdf:type rdfs:range rdfs:Class .  
}
```

**Remember:**  
`rdf:type rdfs:range rdfs:Class .`  
is an axiom in RDFS. This axiom fits  
straight into the rule!

*This is an axiom in RDFS!*



# RDFS entailment

- The triples `?s rdf:type ?o .`  
~~`rdf:type rdfs:range rdfs:Class .`~~  
entail that `?o rdf:type rdfs:Class .`

PREFIX rdf: <...>

PREFIX rdfs: <...>

```
INSERT {  
    ?o rdf:type rdfs:Class .  
} WHERE {  
    ?s rdf:type ?o .  
    rdf:type rdfs:range rdfs:Class .  
}
```



Because  
`rdf:type rdfs:range rdfs:Class .`  
is an axiom in RDFS, this rule  
entails that every object in an  
`rdf:type`-triple is an RDFS class.

# RDFS entailment

- The triples `?s rdf:type ?o .`  
entail that `?o rdf:type rdfs:Class .`



```
PREFIX rdf: <...>  
PREFIX rdfs: <...>
```

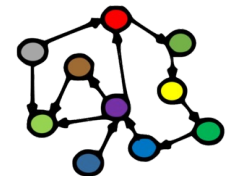
```
INSERT {  
    ?o rdf:type rdfs:Class .  
} WHERE {  
    ?s rdf:type ?o .  
}
```

This rule entails that every object in an `rdf:type`-triple is an RDFS class.

It is not expressed explicitly in RDFS, but is always there in practice, because it is implied by the rule and the axiom we have just shown.

# Subordinate properties (rdfs:subPropertyOf)

- Expresses that: whenever a subject resource and an object resource are related by a particular property, they are necessarily also related by another property, e.g.,
  - whenever this is a fact:  
`ex:Paul_Manafort ex:convictedFor ex:TaxFraud .`
  - then this is necessarily also a fact:  
`ex:Paul_Manafort ex:chargedWith ex:TaxFraud .`
  - we can express this as a *subproperty relationship*:  
`ex:convictedFor rdfs:subPropertyOf ex:chargedWith .`
- Useful for connecting related properties from distinct data sets, e.g.:
  - `rdfs:label`, `dc:title`, `foaf:name`, `skos:prefLabel`, `skos:altLabel`
  - ...just like `rdfs:subClassOf` for properties

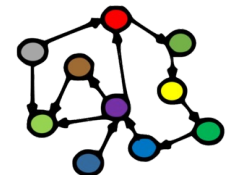


# RDFS entailment [rdfs7]

- “The triples `?s ?p1 ?o .`  
`?p1 rdfs:subPropertyOf ?p2 .`  
entail that `?s ?p2 ?o .`”

PREFIX `rdfs: <...>`

```
INSERT {  
    ?s ?p2 ?o .  
} WHERE {  
    ?s ?p1 ?o .  
    ?p1 rdfs:subPropertyOf ?p2 .  
}
```

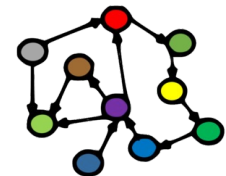


# RDFS entailment [rdfs5]

- `rdfs:subPropertyOf` is *transitive*:
- The triples  
    ?`p1` `rdfs:subPropertyOf` ?`p2` .  
    ?`p2` `rdfs:subPropertyOf` ?`p3` .  
entail that  
    ?`p1` `rdfs:subPropertyOf` ?`p3` .

PREFIX `rdfs:` <...>

```
INSERT {  
    ?p1 rdfs:subPropertyOf ?p3 .  
} WHERE {  
    ?p1 rdfs:subPropertyOf ?p2 .  
    ?p2 rdfs:subPropertyOf ?p3 .  
}
```

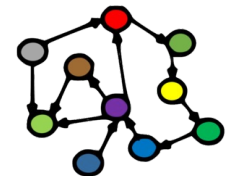


# RDFS entailment [rdfs6]

- `rdfs:subPropertyOf` is *reflexive*:
- The triple `?p rdf:type rdf:Property .`  
entails that `?p rdfs:subPropertyOf ?p .`

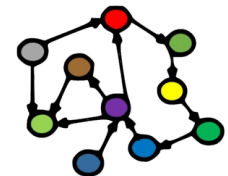
```
PREFIX rdf: <...>  
PREFIX rdfs: <...>
```

```
INSERT {  
    ?p rdfs:subPropertyOf ?p .  
} WHERE {  
    ?p rdf:type rdf:Property .  
}
```



# Additional classes

- RDFS also defines `rdfs:Class`-es for:
  - resources: `rdfs:Resource`
    - the class of all resources
  - literals: `rdfs:Literal`
    - the class of all literals
    - `rdfs:Literal rdfs:subClassOf rdfs:Resource .`
  - datatypes: `rdfs:Datatype`
    - the class of all datatypes
    - `rdfs:Datatype rdfs:subClassOf rdfs:Class .`
  - all of them have `rdf:type rdfs:Class`
  - all of them have entailment rules and axioms



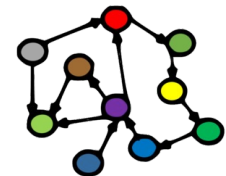
# RDFS entailment [rdfs4a]

- *Every subject in a triple is a resource...*
- The triple `?s ?p ?o .`  
entails that `?s rdf:type rdfs:Resource .`

PREFIX rdf: <...>

PREFIX rdfs: <...>

```
INSERT {  
    ?s rdf:type rdfs:Resource .  
} WHERE {  
    ?s ?p ?o .  
}
```





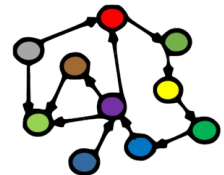
# RDFS entailment [rdfs4b]

- *...and every object too*
- The triple `?s ?p ?o .`  
entails that `?o rdf:type rdfs:Resource .`

PREFIX rdf: <...>

PREFIX rdfs: <...>

```
INSERT {  
    ?o rdf:type rdfs:Resource .  
} WHERE {  
    ?s ?p ?o .  
}
```



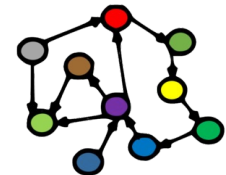
# RDFS entailment [rdfs8]

- Every class corresponds to a set of resources.
  - ...or to a subset of the set of all resources.
- The triple `?c rdf:type rdfs:Class .`  
entails that `?c rdfs:subClassOf rdfs:Resource .`

PREFIX rdf: <...>

PREFIX rdfs: <...>

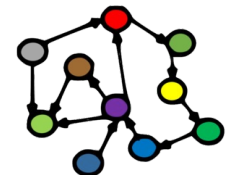
```
INSERT {  
    ?c rdfs:subClassOf rdfs:Resource .  
} WHERE {  
    ?c rdf:type rdfs:Class .  
}
```



# Utility properties in RDFS

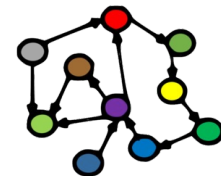
- *Straightforward and much used:*
  - rdfs:label:
    - a human-readable label
  - rdfs:comment:
    - a human-readable comment
  - rdfs:seeAlso:
    - reference to further information
  - rdfs:isDefinedBy:
    - a human-readable definition
    - is a rdfs:subPropertyOf rdfs:seeAlso

*...often take “language-tagged”@en strings as objects*



# What we cannot express...

- RDFS has many limitations, e.g., it cannot say:
  - *“my ancestors' ancestors are also my ancestors”*
  - *“a Person has a unique birth number”*
  - *“a Person has exactly one father”*
  - *“a SoccerTeam has 11 players, but a BasketballTeam has 5”*
  - *“classes with different URIs actually represent the same class”*
  - *“resources with different URIs represent the same resource”*
  - *“properties with different URIs are actually the same”*
  - *“two individuals with different URIs are actually different”*
  - *“two classes cannot share individuals (they are disjoint)”*
  - *“a class is a combination (union or intersection) of other classes”*
  - *“a class is a negation of another class”*
- ***Web Ontology Language (OWL) does all this and more!***



Next week:  
Ontologies (OWL)