An Introduction to OWL 2

Acknowledgement

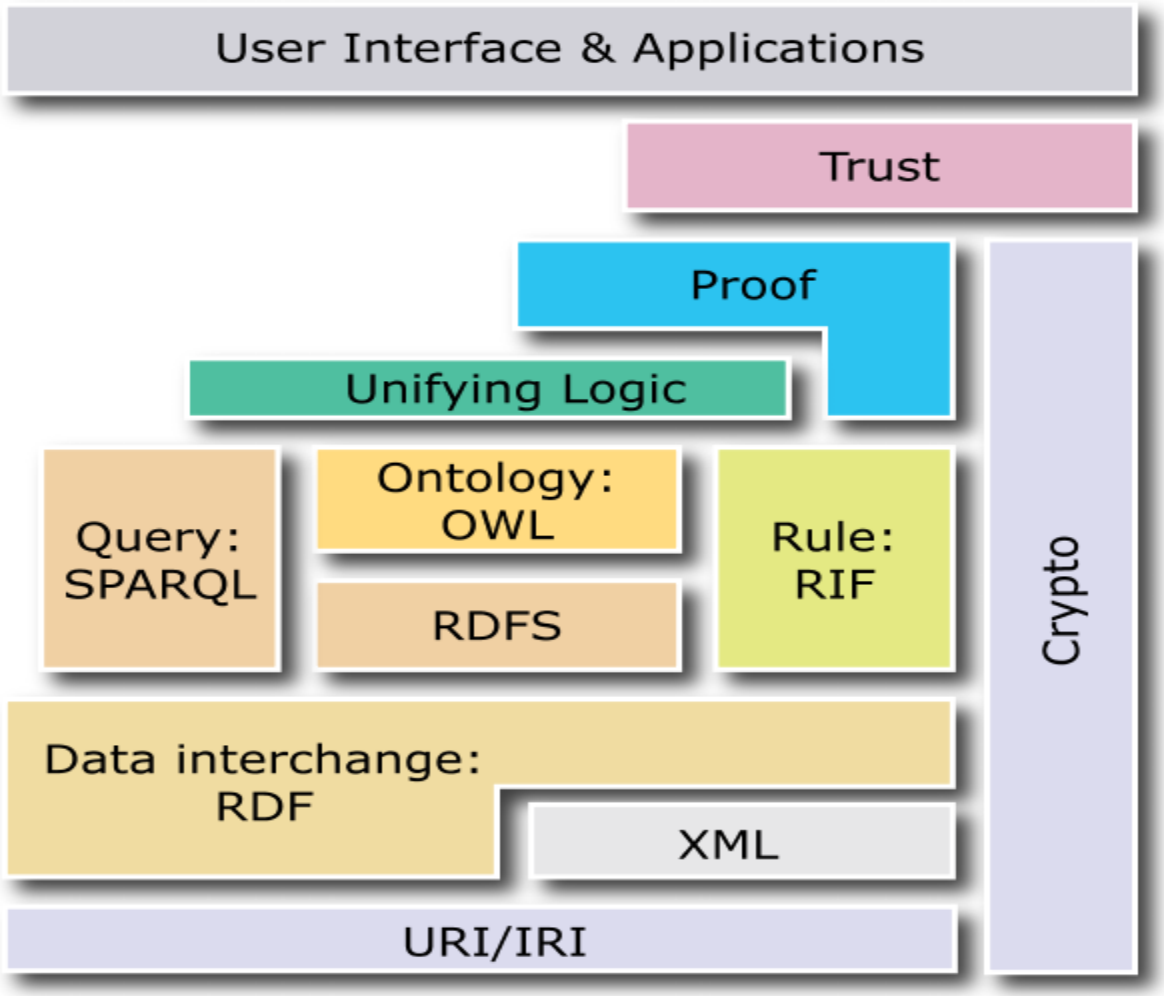
- This presentation is based on the OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax available at http://www.w3.org/TR/owl2-syntax/

- Much of the material in this presentation is verbatim from the above specification.

Outline

- Features of OWL 2
- Structural Specification
- Functional Syntax
- Other Syntaxes
- Examples
- Semantics of OWL 2
- OWL 2 Profiles

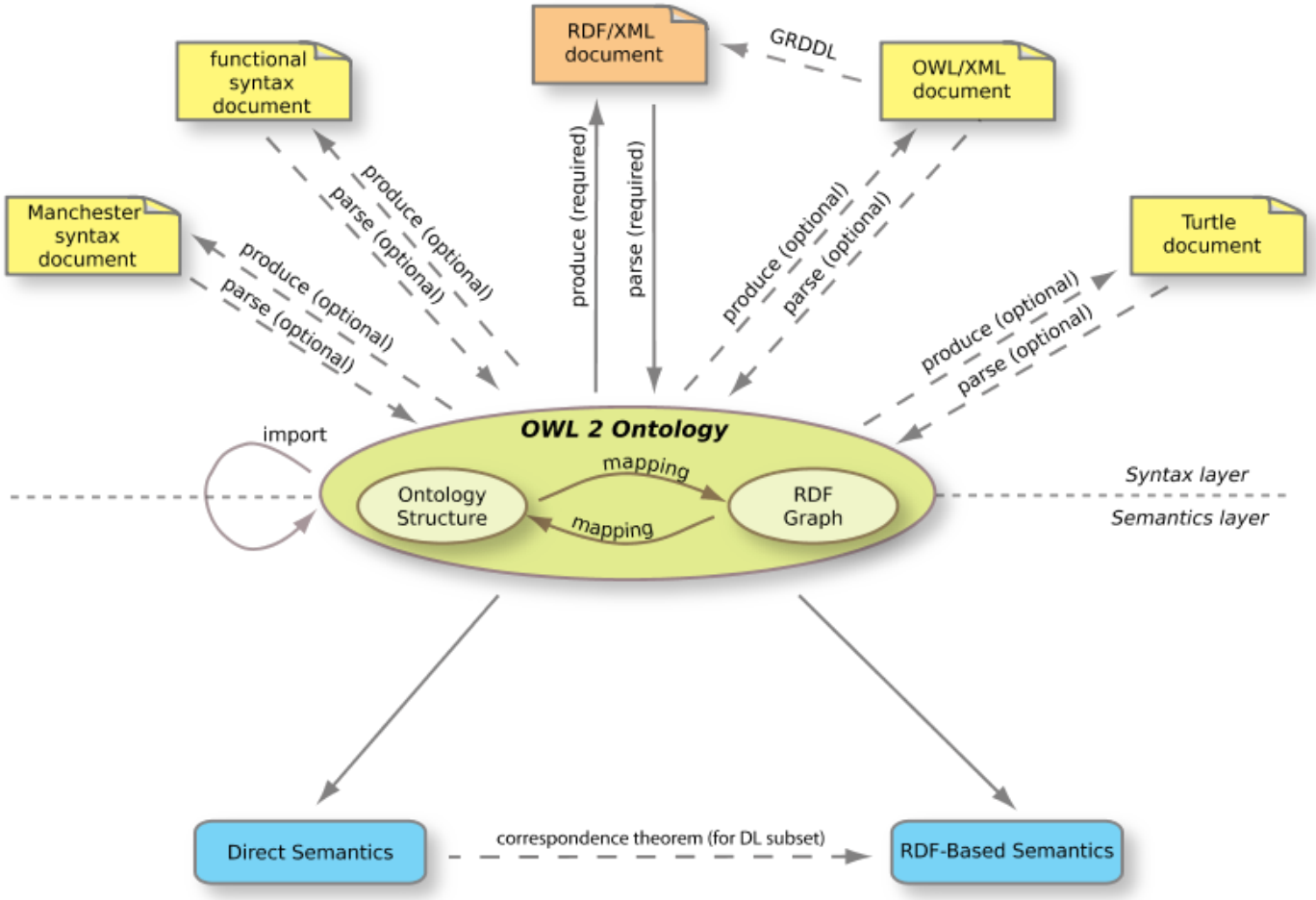
The Semantic Web “Layer Cake”



OWL 2 Basics

- OWL 2 is the current version of the **Web Ontology Language** and a W3C recommendation as of October 2009.
- The previous version of OWL (OWL 1) became a W3C recommendation in 2004.
- All W3C documents about OWL 2 can be found at http://www.w3.org/TR/2009/REC-owl2-overview-20091027/ .

The Structure of OWL 2



OWL 2 Basics (cont'd)

- OWL 2 is a language for writing ontologies for the Web.
- It is based on well-known concepts and results from **description logics**.
- Like DLs, OWL 2 is a language for representing **knowledge about things, groups of things, and relations between things.**

OWL 2 Terminology

- The things or objects about which knowledge is represented (e.g., John, Mary) are called **individuals**.
- Groups of things (e.g., female) are called **classes**.
- Relations between things (e.g., married) are called **properties**.
- Individuals, classes and properties are called **entities**.

OWL 2 Terminology (cont'd)

- As in DLs, entities can be combined using **constructors** to form complex descriptions called **expressions**.

- To represent knowledge in OWL (like in any other KR language), we make statements. These statements are called **axioms.**

Annotations

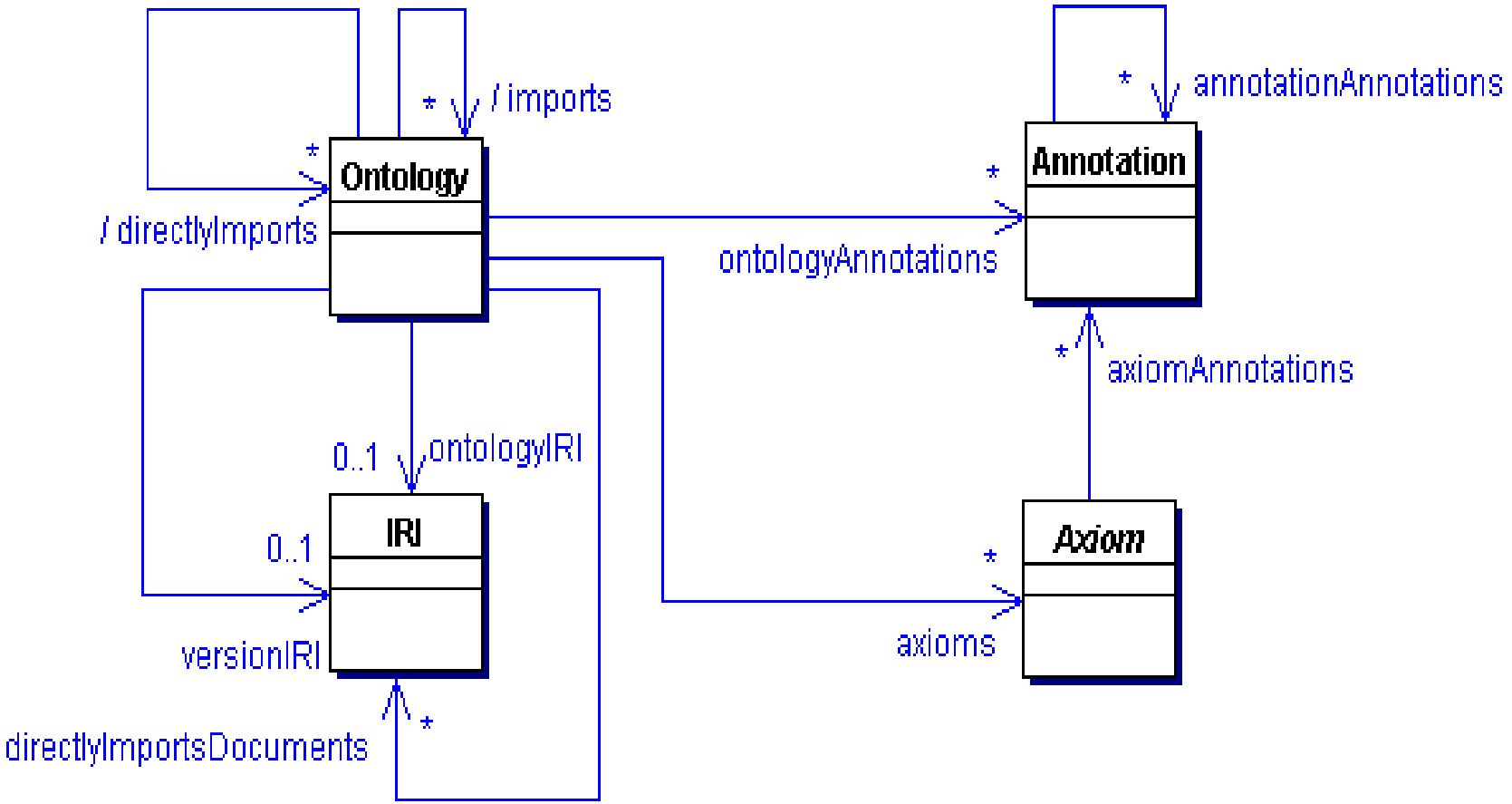
- Entities, expressions and axioms form the **logical part** of OWL 2. They can be given a precise semantics and inferences can be drawn from them.
- In addition, entities, axioms, and ontologies can be **annotated.**
- Example: A class can be given a human-readable label that provides a more descriptive name for the class.
- Annotations have **no effect** on the logical aspects of an ontology. For the purposes of the OWL 2 semantics, annotations are treated as not being present.

IRIs

- Ontologies and their elements are identified using **International Resource Identifiers (IRIs).**

- In OWL 2, an IRI can be written in full or it can be abbreviated as `prefix:lname` as in XML qualified names where `prefix` is a **namespace** and `lname` is the **local name** with respect to the namespace.

The Structure of an Ontology



Ontology IRI and Version IRIs

- An ontology may have an **ontology IRI**, which is used to identify it.
- If an ontology has an ontology IRI, the ontology may additionally have a **version IRI**, which is used to identify the version of the ontology. The version IRI may, but need not be equal to the ontology IRI.
- An **ontology series** is identified using an ontology IRI, and each version in the series is assigned a different version IRI. Only one version of the ontology is the **current one.**
- Example:
  - Ontology IRI: `<http://www.example.com/my>`
  - Version IRIs: `<http://www.example.com/my/1.0>`, `<http://www.example.com/my/2.0>`, …
- An ontology without an ontology IRI must not contain a version IRI.
- Ontology IRIs and version IRIs should satisfy various uniqueness constraints that OWL 2 tools should check, for detecting possible problems.

Ontology Document

- Each ontology is associated with an **ontology document** which physically contains the ontology stored in a particular way (e.g., a text file).

- An ontology document should be accessible via the IRIs determined by the rules defined in the W3C specification.
  - Example: The document of the current version of an ontology should always be accessible via the ontology IRI and the current version IRI.

Imports

- An OWL 2 ontology can **import (directly or indirectly) other ontologies** in order to gain access to their entities, expressions and axioms, thus providing the basic facility for **ontology modularization.**

- **Example:** an ontology of sensors can import a geospatial ontology to specify the location of sensors.

OWL 2 Syntaxes

- **The Functional-Style syntax.** This syntax is designed to be easier for specification purposes and to provide a foundation for the implementation of OWL 2 tools such as APIs and reasoners. **This is the syntax we will use in this presentation.**

- **The RDF/XML syntax**: this is just RDF/XML, with a particular translation for the OWL constructs. Here one can use other popular syntaxes for RDF, e.g., **Turtle syntax.**

- **The Manchester syntax:** this is a frame-based syntax that is designed to be easier for users to read.

- **The OWL XML syntax:** this is an XML syntax for OWL defined by an XML schema.

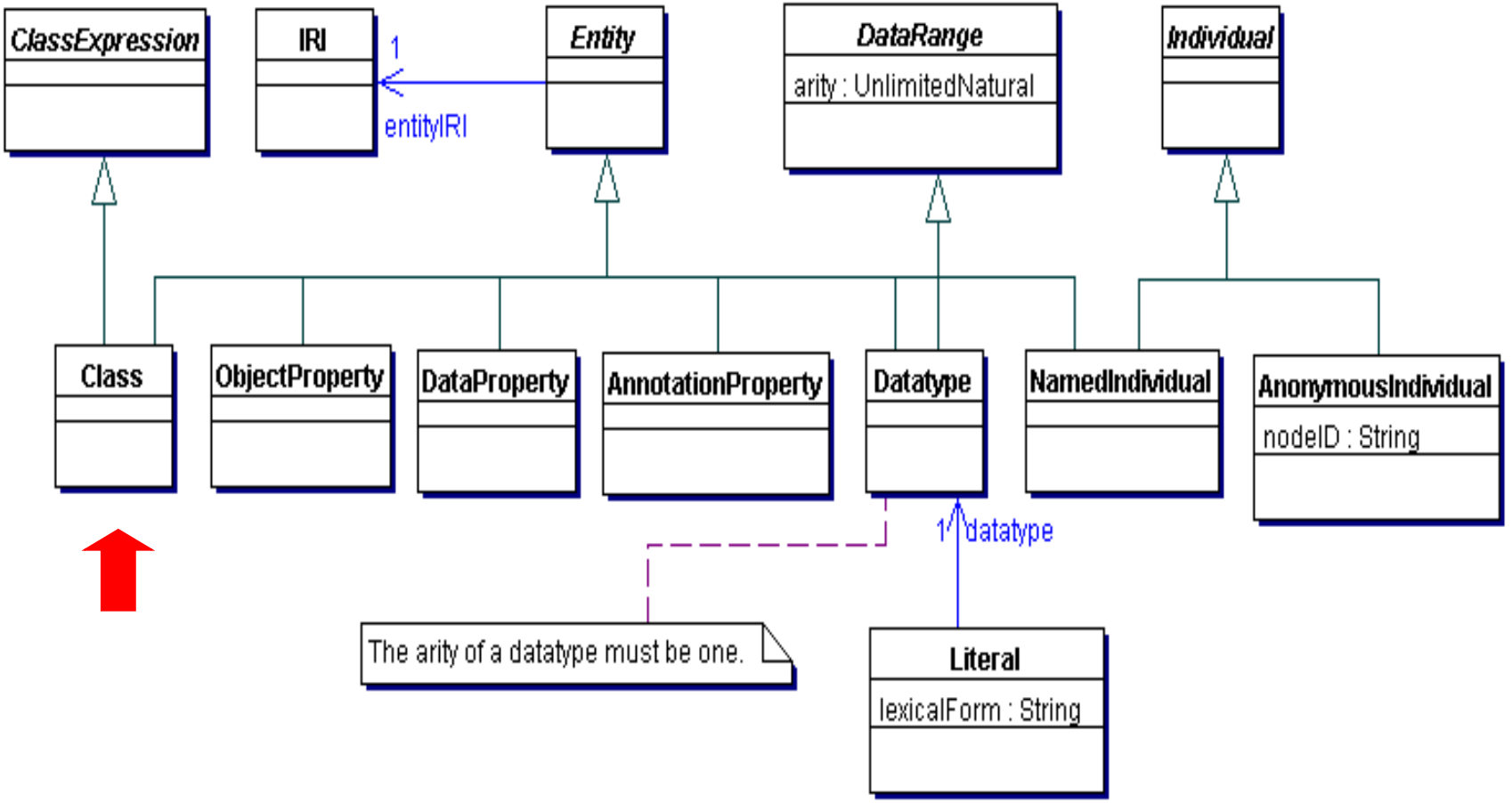
BNF Grammar for the Functional Syntax of OWL 2

ontologyDocument := { prefixDeclaration } Ontology
prefixDeclaration := 'Prefix' '(' prefixName ':=' fullIRI ')'
Ontology :=
'Ontology' '(' [ontologyIRI [versionIRI]]
directlyImportsDocuments
ontologyAnnotations
axioms
')'
ontologyIRI := IRI
versionIRI := IRI
directlyImportsDocuments := { 'Import' '(' IRI ')' }
axioms := { Axiom }

Example

```
Prefix(ex:=<http://www.example.com/ontology1#>)
Prefix(owl:=<http://www.w3.org/2002/07/owl#>)

Ontology(<http://www.example.com/ontology1>

    Import(<http://www.example.com/ontology2>)
    Annotation(rdfs:label "An example ontology")

    SubClassOf(ex:Person owl:Thing)
    SubClassOf(ex:Male ex:Person)
    SubClassOf(ex:Female ex:Person)

)
```

Things One Can Define in OWL 2

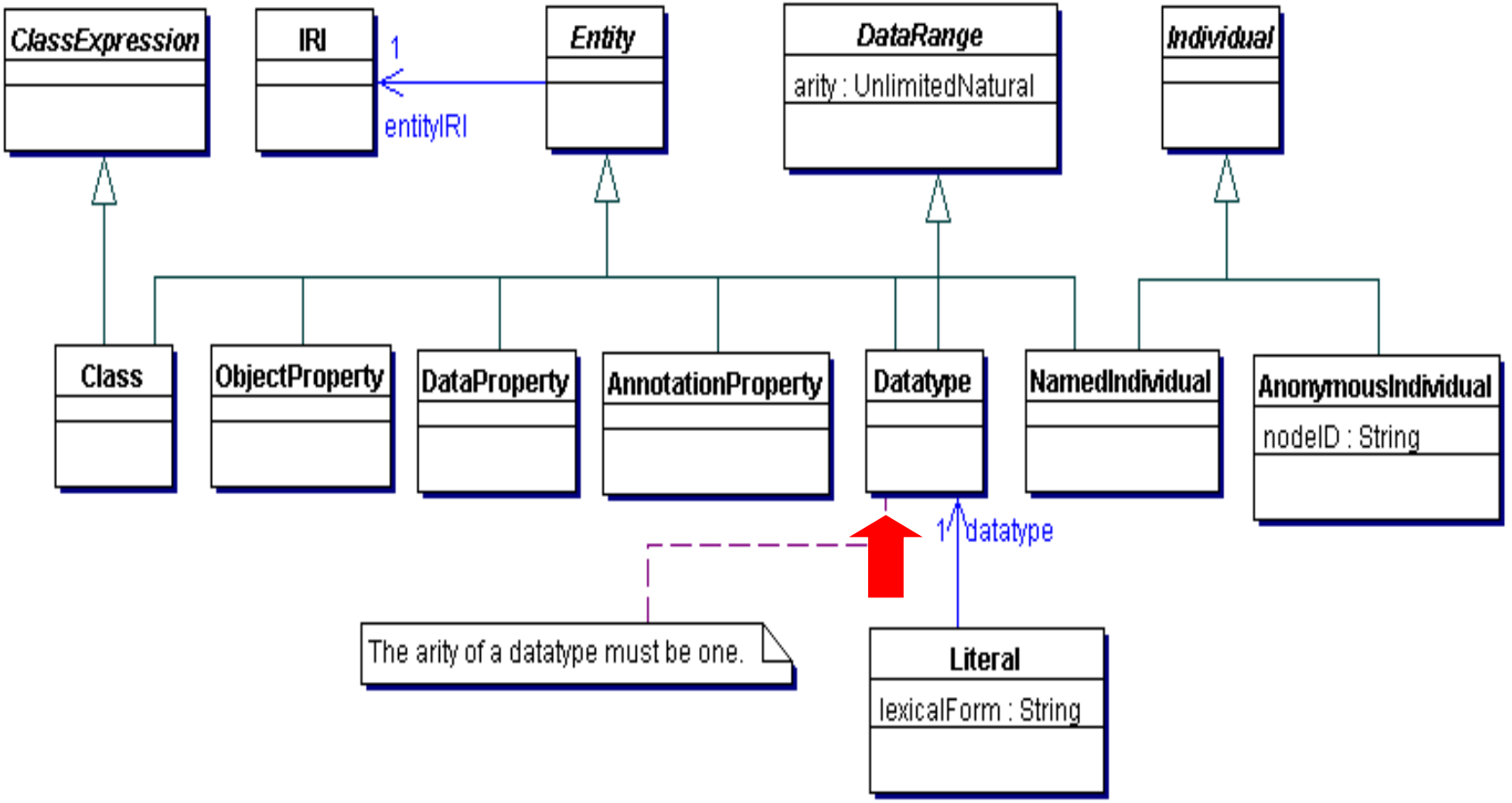


Classes

- **Classes** (e.g., `a:Female`) represent sets of individuals.

- Built-in classes:
  - `owl:Thing`, which represents the set of all individuals.
  - `owl:Nothing`, which represents the empty set.

Things One Can Define in OWL 2 (cont'd)



Datatypes

- **Datatypes** are entities that represent sets of data values.
- OWL 2 offers a rich set of data types: decimal numbers, integers, floating point numbers, rationals, reals, strings, binary data, IRIs and time instants.
- In most cases, these data types are taken from XML schema. From RDF and RDFS, we have `rdf:XMLLiteral, rdf:PlainLiteral` and `rdfs:Literal.`
- `rdfs:Literal` contains all the elements of other data types.
- There are also the OWL datatypes `owl:real` and `owl:rational.`
- Formally, the data types supported are specified in the OWL 2 **datatype map**.

Datatypes (cont'd)

- In a datatype map, each datatype is identified by an IRI and is defined by the following components:
  - The **value space** is the set of values of the datatype. Elements of the value space are called **data values**.
  - The **lexical space** is a set of strings that can be used to refer to data values. Each member of the lexical space is called a **lexical form**, and it is mapped to a particular data value.
  - The **facet space** is a set of pairs of the form *(F,v)* where *F* is an IRI called a **constraining facet**, and *v* is an arbitrary data value called the **constraining value**. Each such pair is mapped to a subset of the value space of the datatype.

Facet Space

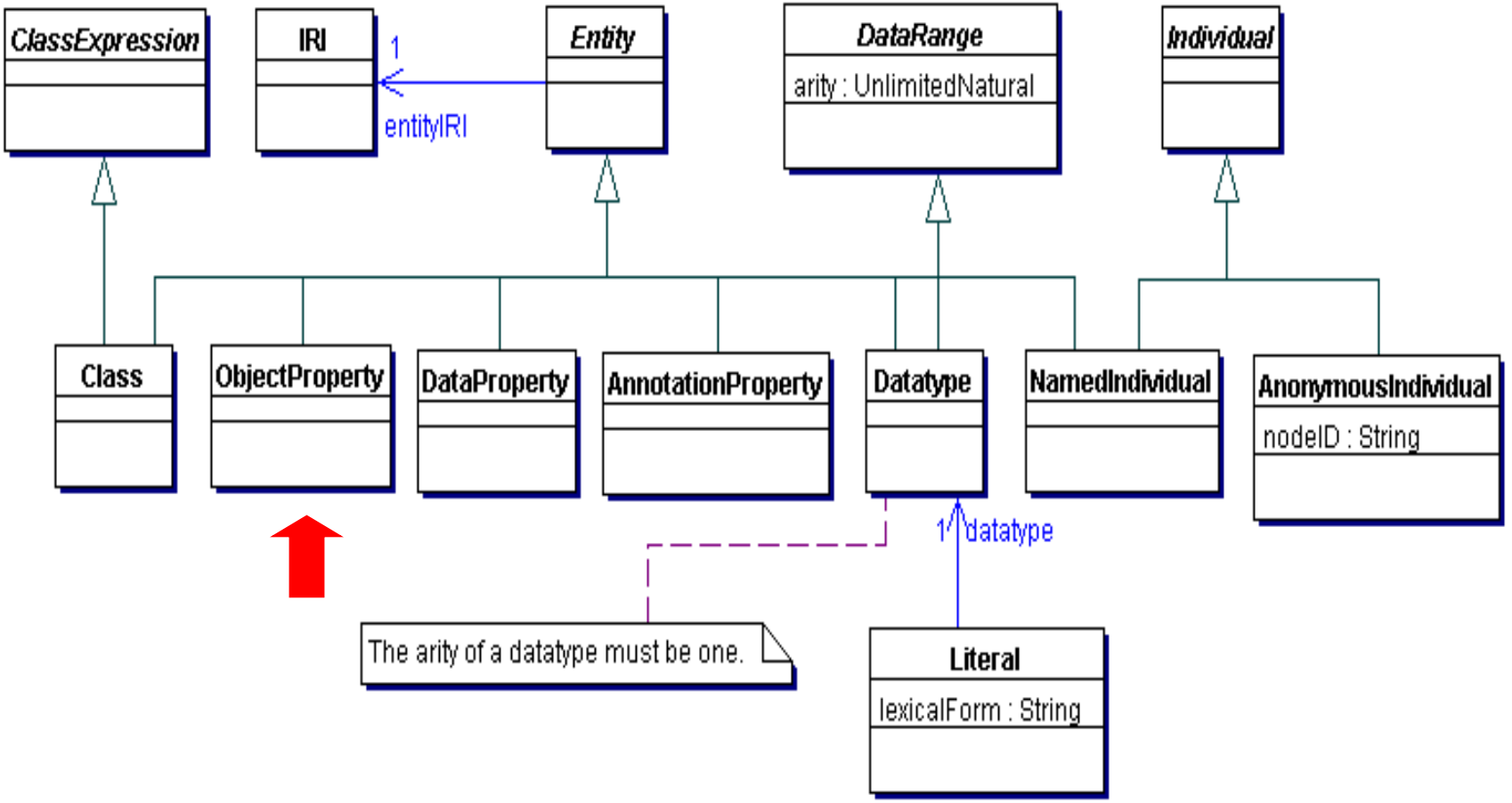
- For the XML Schema datatypes `xsd:double, xsd:float`, and `xsd:decimal`, the **constraining facets** allowed are: `xsd:minInclusive, xsd:maxInclusive, xsd:minExclusive` and `xsd:maxExclusive`.

- Example: The pair `(xsd:minInclusive,v)` of the facet space denotes the set of all numbers *x* from the value space of the datatype such that `x=v` or `x>v`.

- Similarly for other datatypes.

- We will see later how constraining facets can be used to define **data ranges.**

Things One Can Define in OWL 2 (cont'd)



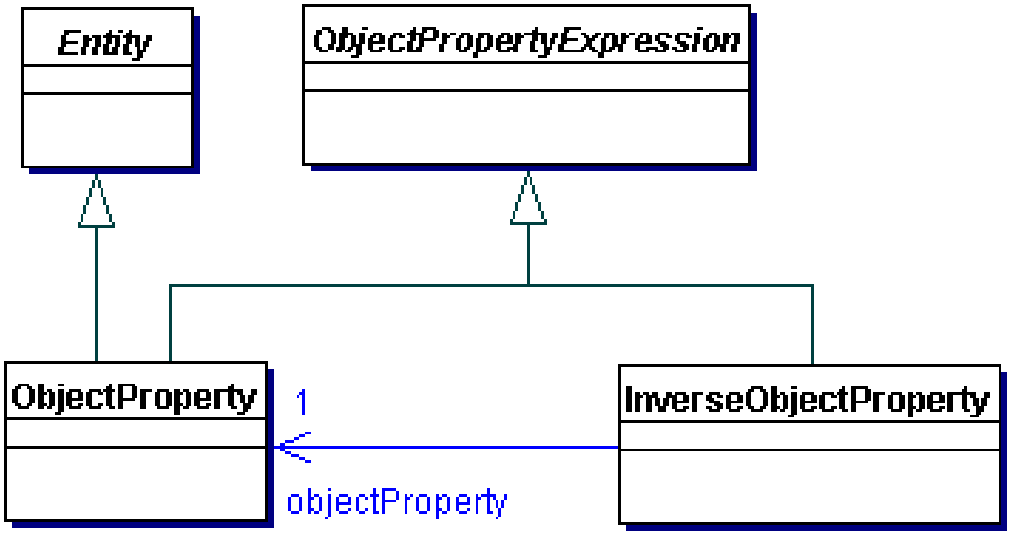
Object Properties

- **Object properties** (e.g., `a:parentOf`) connect pairs of individuals.

- Built-in object properties:
  - `owl:topObjectProperty`, which connects all possible pairs of individuals.
  - `owl:bottomObjectProperty`, which does not connect any pair of individuals.

Object Property Expressions

- Object properties can be used to form **object property expressions.**



Inverse Object Property Expressions

- An **inverse object property expression** `ObjectInverseOf(P)` connects an individual `I1` with `I2` if and only if the object property `P` connects `I2` with `I1`.

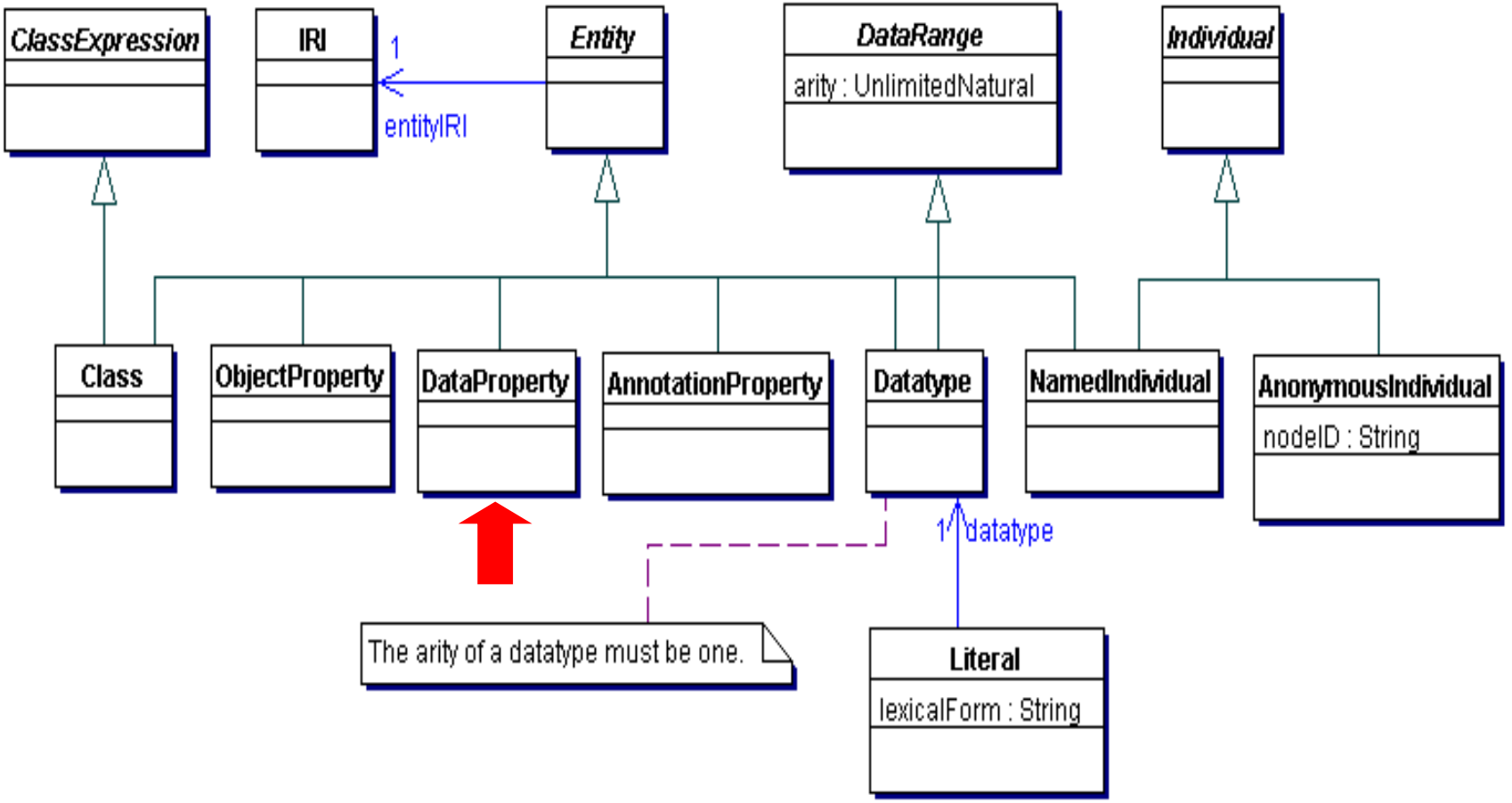
- Example: If an ontology contains the axiom

```
ObjectPropertyAssertion(a:fatherOf a:Peter a:Stewie)
```

then the ontology entails

```
ObjectPropertyAssertion(ObjectInverseOf(a:fatherOf) a:Stewie
                        a:Peter)
```

Things One Can Define in OWL 2 (cont'd)



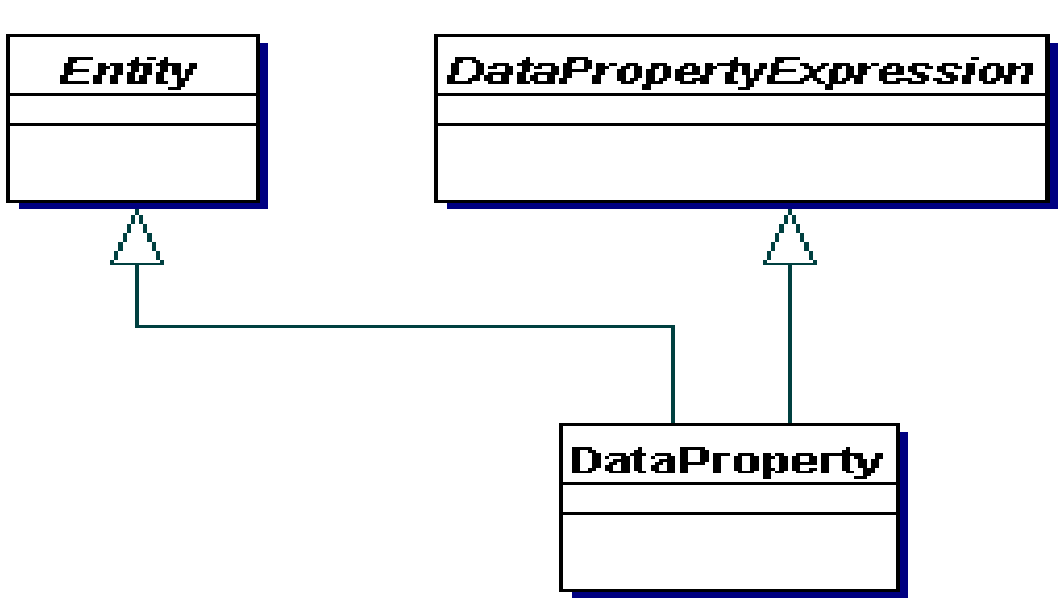
Data Properties

- **Data properties** (e.g., `a:hasAge`) connect individuals with literals.

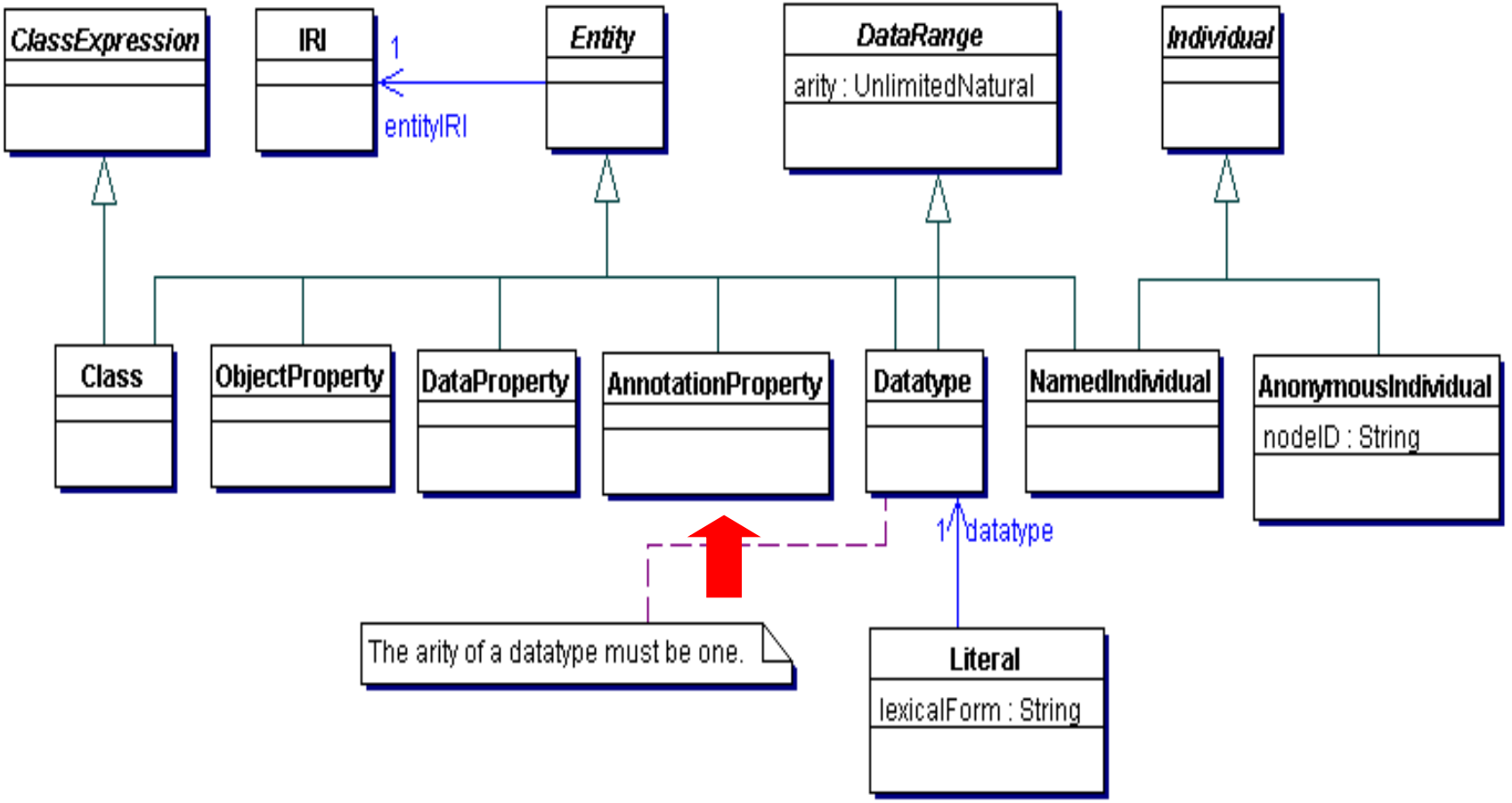
- Built-in properties:
  - `owl:topDataProperty`, which connects all possible individuals with all literals.
  - `owl:bottomDataProperty`, which does not connect any individual with a literal.

Data Property Expressions

- The only allowed **data property expression** is a data property.



Things One Can Define in OWL 2 (cont'd)

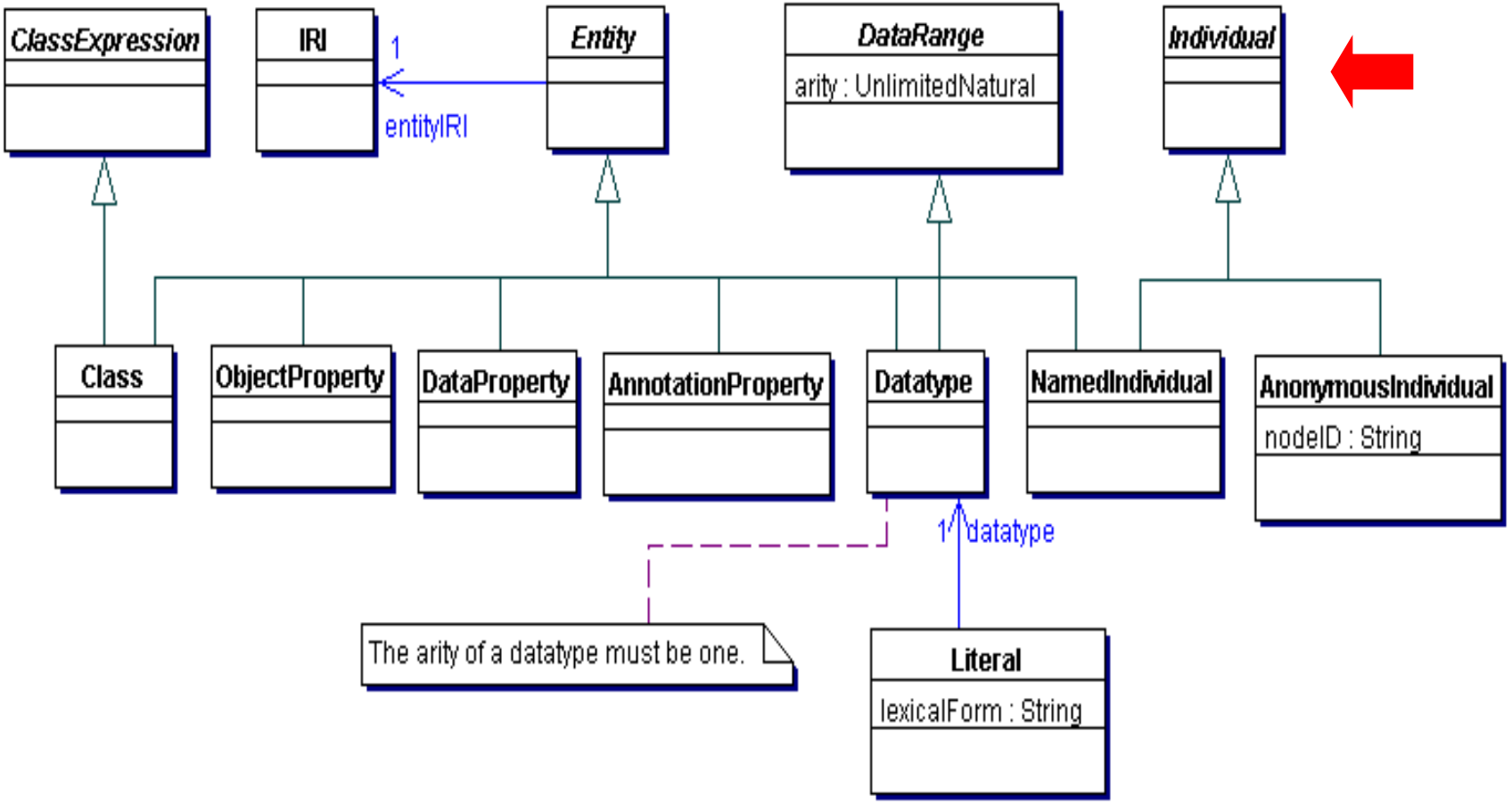


Annotation Properties

- **Annotation properties** can be used to provide an annotation for an ontology, axiom, or an IRI.

- Users can **define their own** annotation properties (we will see how later on) or **use the available** built-in annotation properties:
  - `rdfs:label, rdfs:comment, rdfs:see, rdfs:isDefinedBy`
  - `owl:deprecated, owl:versionInfo, owl:priorVersion, owl:backwardCompatibleWith, owl:incompatibleWith`

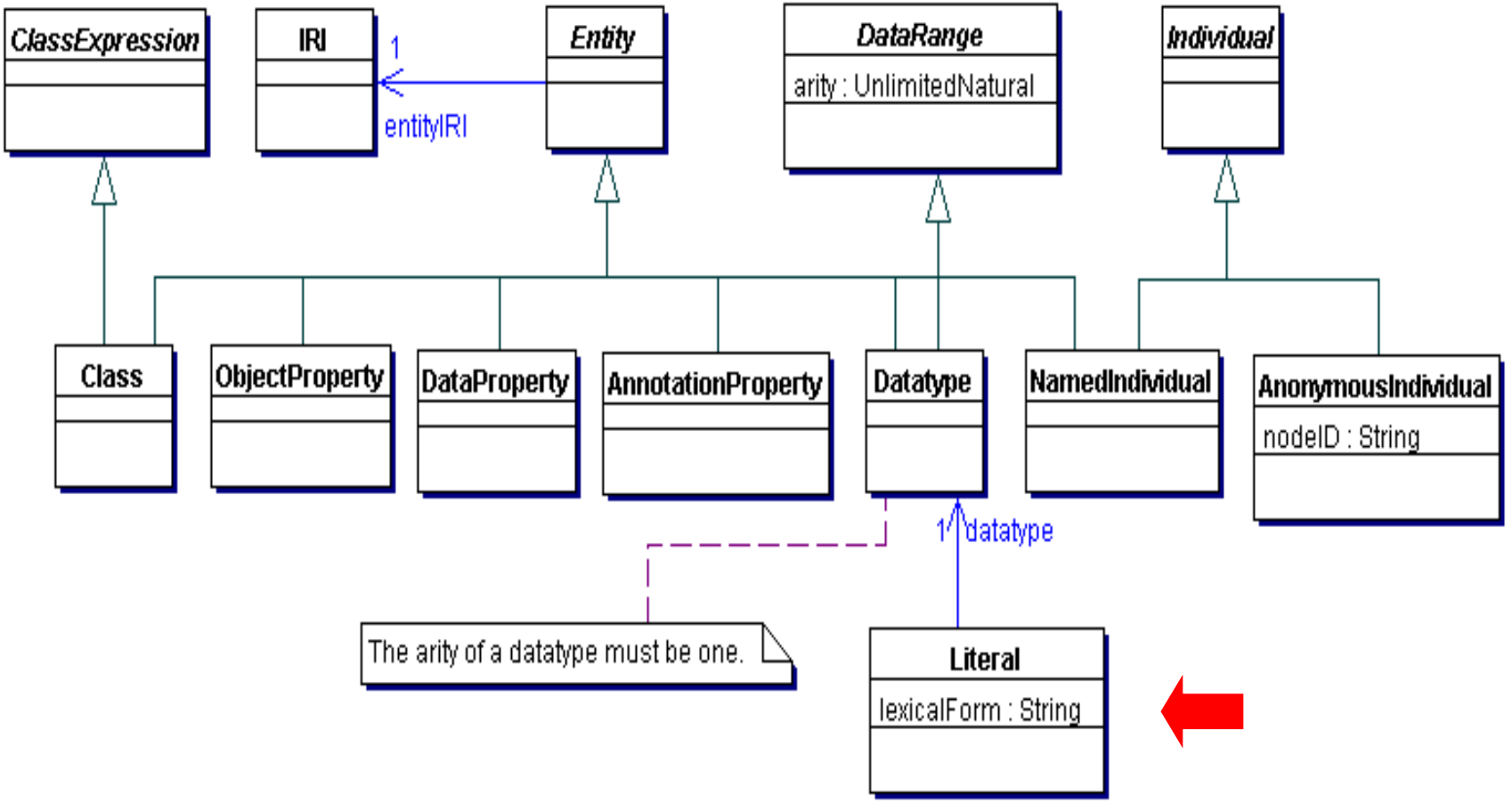
Things One Can Define in OWL 2 (cont'd)



Individuals

- **Individuals** represent actual objects from the domain.
- There are two types of individuals:
  - **Named individuals** are given an explicit name (an IRI e.g., `a:Peter`) that can be used in any ontology to refer to the same object.
  - **Anonymous individuals** do not have a global name. They can be defined using a name (e.g., `_:somebody`) local to the ontology they are contained in. They are like blank nodes in RDF.

Things One Can Define in OWL 2 (cont'd)

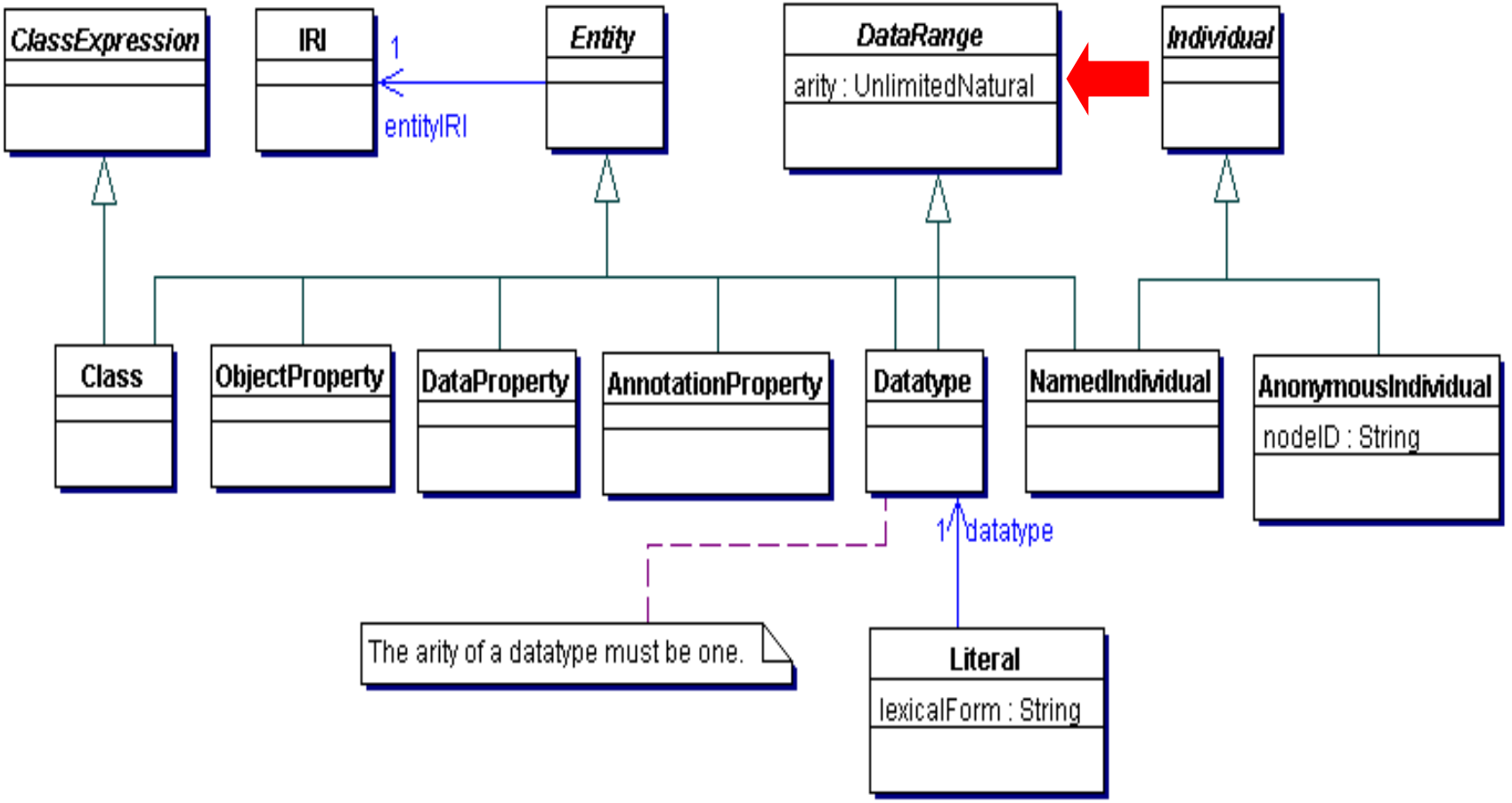


Literals

- **Literals** represent data values such as particular strings or integers. They are analogous to RDF literals.

- Examples:
  - `"1"^^xsd:integer` (**typed literal**)
  - `"Family Guy"` (**plain literal**, an abbreviation for `"Family Guy"^^rdf:PlainLiteral`)
  - `"Padre de familia"@es` (**plain literal with language tag**, an abbreviation for `"Padre de familia@es"^^rdf:PlainLiteral`)

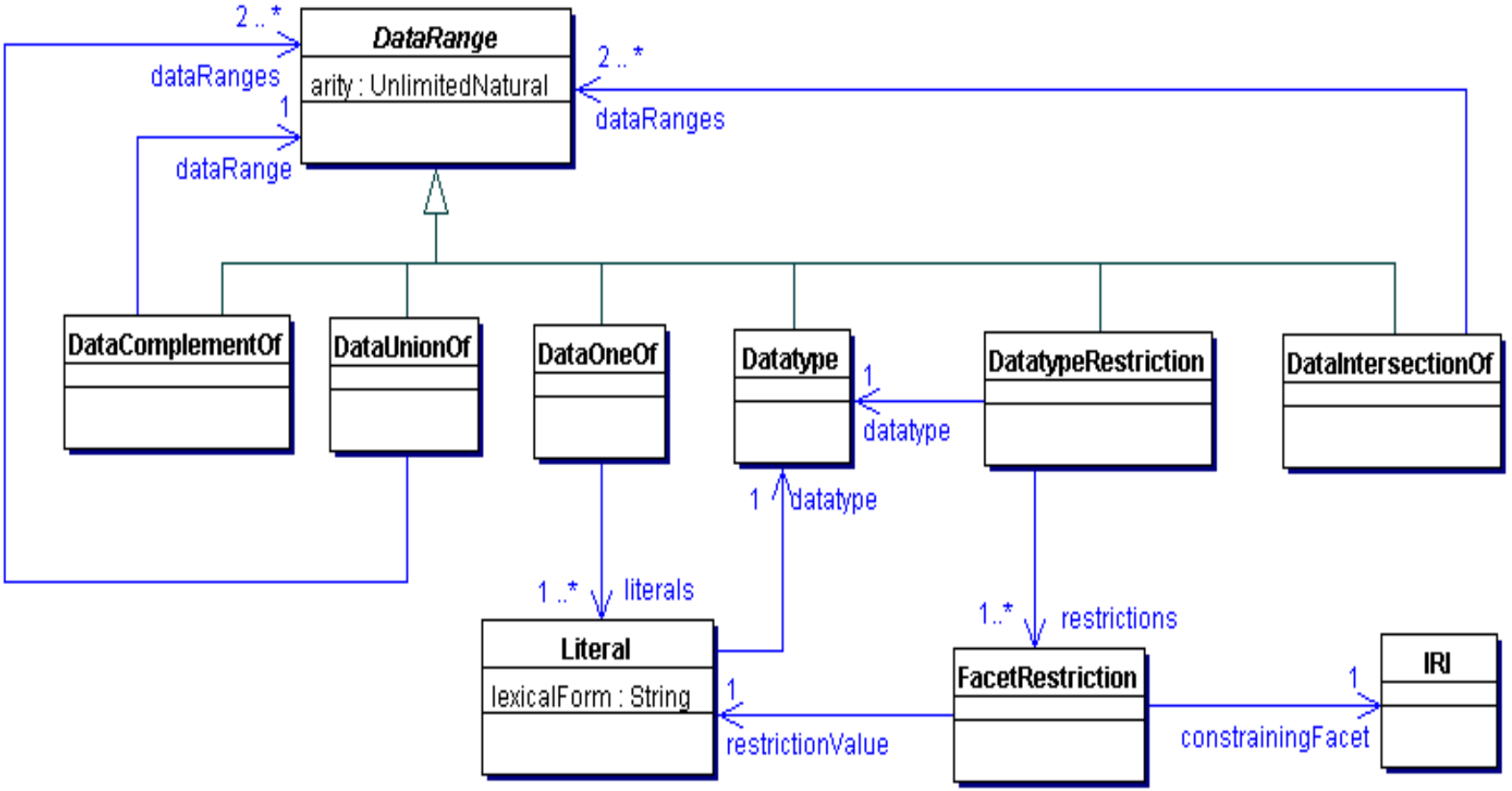
Things One Can Define in OWL 2 (cont'd)



Data Ranges

- Data ranges represent **sets of tuples of literals**. They are defined using datatypes and constraining facets.
- Examples:
  - The set of integers greater than 10.
  - The set of strings that contain “good” as a substring.
  - The set of `(x,y)` such that `x` and `y` are integers and `x < y`.
- Each data range is associated with a positive **arity**, which determines the size of its tuples.
- Datatypes are themselves data ranges of arity 1.
- Data ranges are used in **restrictions on data properties,** as we will see later when we define class expressions.

Data Ranges


DataRange
arity : UnlimitedNatural
2 .. *
dataRanges
1
dataRange
2 .. *
dataRanges
DataComplementOf
DataUnionOf
DataOneOf
Datatype
1
datatype
DatatypeRestriction
DataIntersectionOf
1
datatype
1 ..*
literals
Literal
lexicalForm : String
1
restrictionValue
1..*
restrictions
FacetRestriction
1
constrainingFacet
IRI

BNF for Data Ranges

DataRange :=
Datatype |
DataIntersectionOf |
DataUnionOf |
DataComplementOf |
DataOneOf |
DatatypeRestriction

DataIntersectionOf := 'DataIntersectionOf' '(' DataRange DataRange { DataRange } ')'

DataUnionOf := 'DataUnionOf' '(' DataRange DataRange { DataRange } ')'

DataComplementOf := 'DataComplementOf' '(' DataRange ')'

DataOneOf := 'DataOneOf' '(' Literal { Literal } ')'

Examples

```
DataIntersectionOf(xsd:nonNegativeInteger
                   xsd:nonPositiveInteger)
```

```
DataUnionOf(xsd:string xsd:integer)
```

```
DataComplementOf(xsd:positiveInteger)
```

```
DataOneOf("Peter" "John")
```

Datatype Restrictions

DatatypeRestriction :=
'DatatypeRestriction' '('
Datatype constrainingFacet restrictionValue
{ constrainingFacet restrictionValue } ')'

constrainingFacet := IRI

restrictionValue := Literal

Datatype Restrictions

- A **datatype restriction** `DatatypeRestriction(DT F1 lt1 ... Fn ltn)` consists of a unary datatype `DT` and `n` pairs `(Fi,lti)` where `Fi` is a constraining facet of `DT` and `lti` a literal value.

- The **data range** represented by a datatype restriction is **unary** and is obtained by restricting the value space of `DT` according to the **conjunction** of all `(Fi,lti)`.

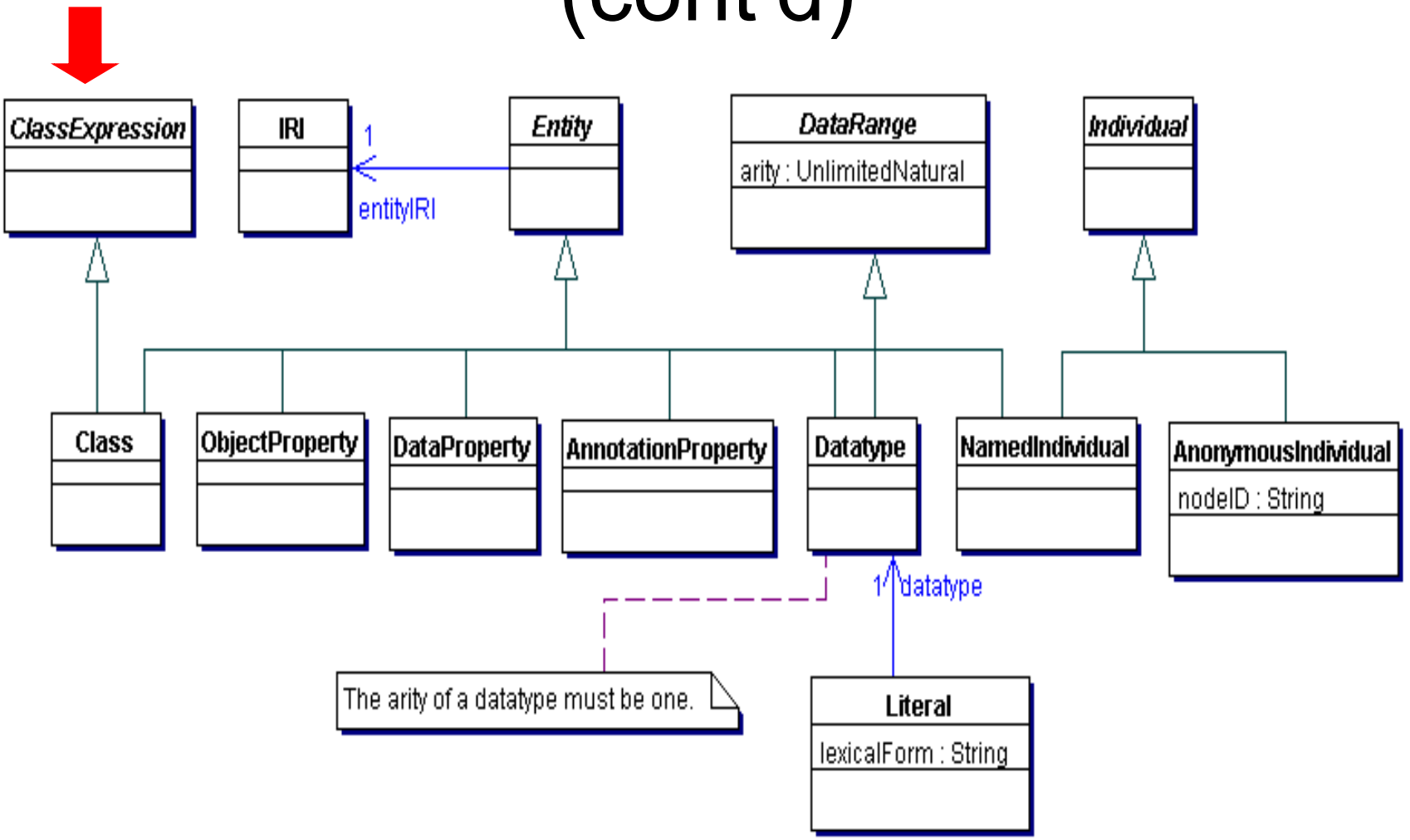
- **Observation:** Thus, although the definition of data range speaks of tuples of any arity, the syntax defined allows only **unary** data ranges.

Example

- The following data type restriction represents the set of integers 5, 6, 7, 8, and 9:

```
DatatypeRestriction(xsd:integer
xsd:minInclusive "5"^^xsd:integer

xsd:maxExclusive "10"^^xsd:integer)
```

Things One Can Define in OWL 2 (cont'd)



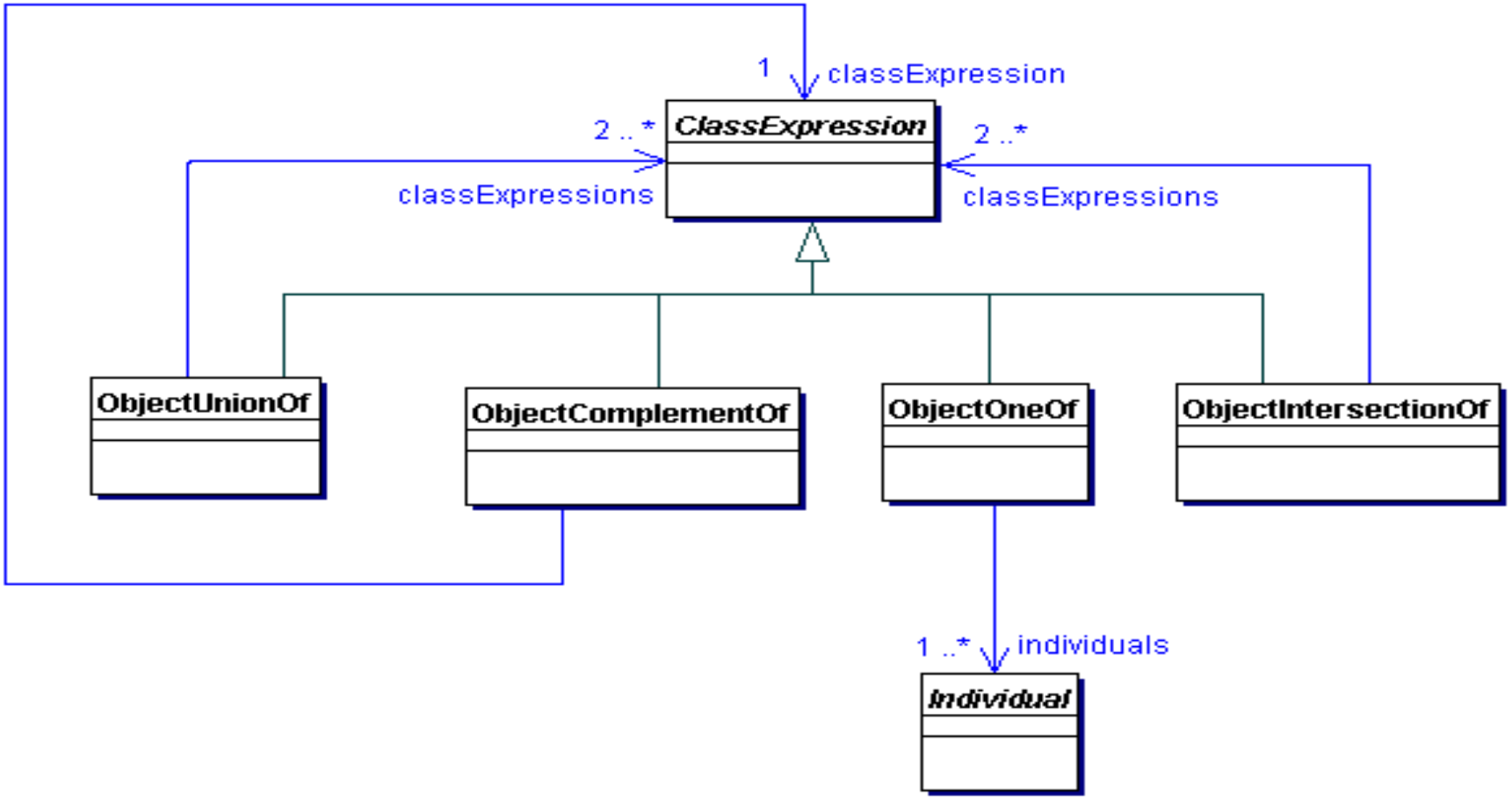
Class Expressions

- Class names and property expressions can be used to construct **class expressions**.
- These are essentially the **complex concepts** or **descriptions** that we can define in DLs.
- Class expressions represent sets of individuals by formally specifying conditions on the individuals' properties; individuals satisfying these conditions are said to be **instances** of the respective class expressions.

Ways to Form Class Expressions

- Class expressions can be formed by:
  - Applying the standard **Boolean connectives** to simpler class expressions or by **enumerating the individuals** that belong to an expression.
  - Placing **restrictions on object property expressions.**
  - Placing **restrictions on the cardinality** of object property expressions.
  - Placing **restrictions on data property expressions.**
  - Placing **restrictions on the cardinality** of data property expressions.

Boolean Connectives and Enumeration of Individuals



Intersection Class Expressions

- An **intersection class expression** `ObjectIntersectionOf(CE1 ... CEn)` contains all individuals that are instances of all class expressions `CEi` for $1 \leq i \leq n$.

- Example:

```
ObjectIntersectionOf(a:Dog a:CanTalk)
```

Union Class Expressions

- A **union class expression** `ObjectUnionOf(CE1 ... CEn)` contains all individuals that are instances of at least one class expression `CEi` for $1 \leq i \leq n$.

- Example:

```
ObjectUnionOf(a:Man a:Woman)
```

Complement Class Expressions

- A **complement class expression** `ObjectComplementOf(CE)` contains all individuals that are not instances of the class expression `CE`.

- Example:

```
ObjectComplementOf(a:Man)
```

Example Inference

- From

```
DisjointClasses(a:Man a:Woman)
 ClassAssertion(a:Woman a:Lois)
```

we can infer

```
ClassAssertion(ObjectComplementOf(a:Man)
                a:Lois)
```

Enumeration of Individuals

- An **enumeration of individuals** `ObjectOneOf(a1 ... an)` contains exactly the individuals `ai` with `1≤i≤n`.

- Example:

```
ObjectOneOf(a:Peter a:Lois
a:Stewie a:Meg a:Chris a:Brian)
```

Example Inference

- From

```
EquivalentClasses(a:GriffinFamilyMember
  ObjectOneOf(a:Peter a:Lois a:Stewie a:Meg
    a:Chris a:Brian))

DifferentIndividuals(a:Quagmire a:Peter a:Lois
    a:Stewie a:Meg a:Chris a:Brian)
```

we can infer

```
ClassAssertion(
ObjectComplementOf(a:GriffinFamilyMember)
    a:Quagmire)
```

Example Inference (con’td)

- From

```
ClassAssertion(a:GriffinFamilyMember a:Peter)
ClassAssertion(a:GriffinFamilyMember a:Lois)
ClassAssertion(a:GriffinFamilyMember a:Stewie)
ClassAssertion(a:GriffinFamilyMember a:Meg)
ClassAssertion(a:GriffinFamilyMember a:Chris)
ClassAssertion(a:GriffinFamilyMember a:Brian)

DifferentIndividuals(a:Quagmire a:Peter a:Lois a:Stewie
                     a:Meg a:Chris a:Brian)
```

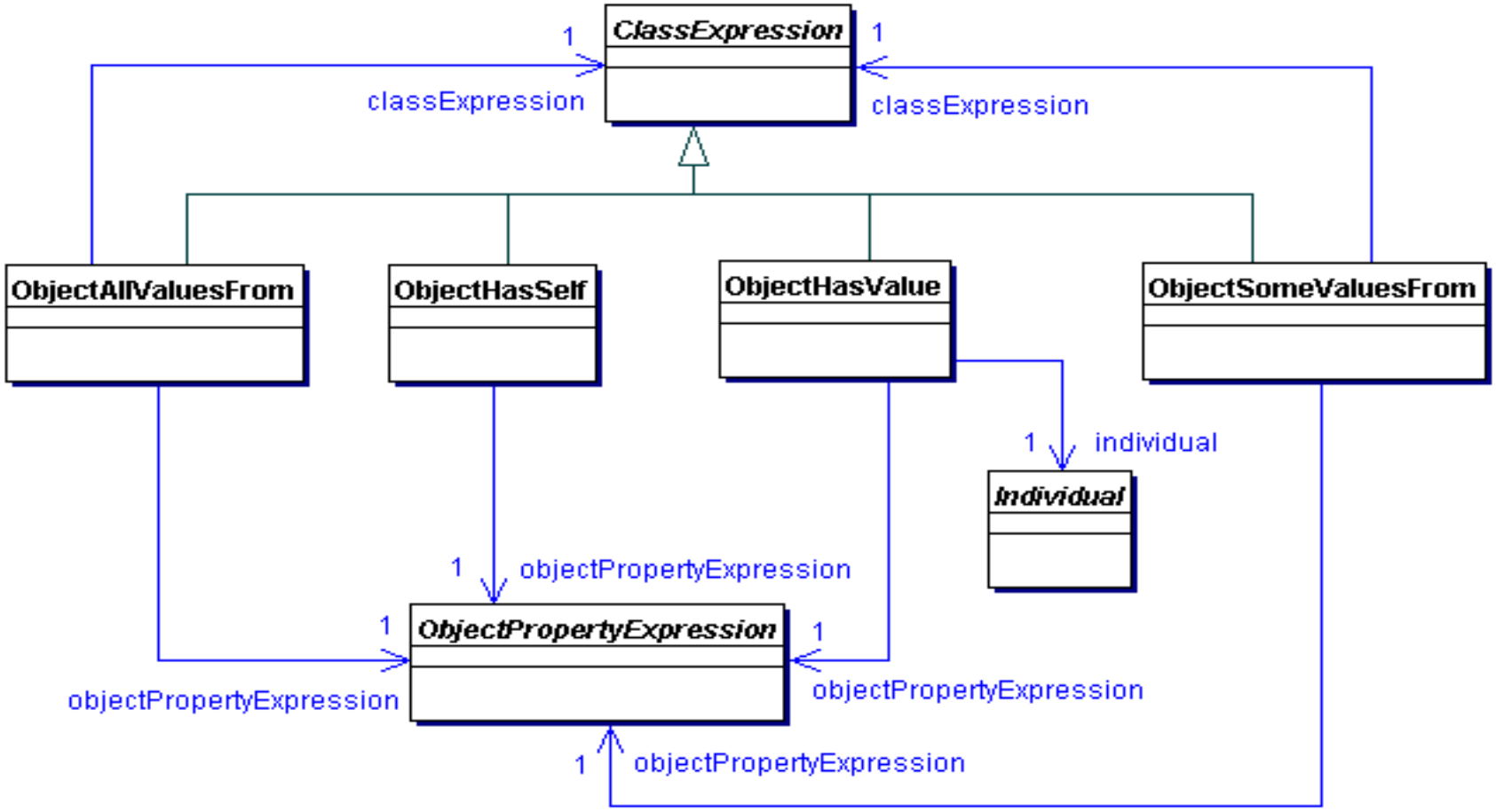
we **cannot** infer

```
ClassAssertion(
ObjectComplementOf(a:GriffinFamilyMember) a:Quagmire)
```

Ways to Form Class Expressions (cont'd)

- Class expressions can be formed by:
  - Applying the standard **Boolean connectives** to simpler class expressions or by **enumerating the individuals** that belong to an expression.
  - Placing **restrictions on object property expressions.**
  - Placing **restrictions on the cardinality** of object property expressions.
  - Placing **restrictions on data property expressions.**
  - Placing **restrictions on the cardinality** of data property expressions.

Object Property Restrictions



Existential Quantification

- An **existential class expression** `ObjectSomeValuesFrom(OPE CE)` consists of an object property expression `OPE` and a class expression `CE`, and it contains all those individuals that are connected by `OPE` to an individual that is an instance of `CE`.

- Example:

```
ObjectSomeValuesFrom(a:fatherOf a:Man)
```

- If OPE is simple, the above class expression is equivalent with the class expression

```
ObjectMinCardinality(1 OPE CE)
```

Example Inference

- From

```
ObjectPropertyAssertion(a:fatherOf
        a:Peter a:Stewie)

ClassAssertion(a:Man a:Stewie)
```

we can infer

```
ClassAssertion(
ObjectSomeValuesFrom(a:fatherOf
        a:Man) a:Peter)
```

Universal Quantification

- A **universal class expression** `ObjectAllValuesFrom(OPE CE)` consists of an object property expression `OPE` and a class expression `CE`, and it contains all those individuals that are connected by `OPE` only to individuals that are instances of `CE`.

- Example:

```
ObjectAllValuesFrom(a:fatherOf a:Man)
```

- If OPE is simple, the above class expression is equivalent with the class expression

```
ObjectMaxCardinality(0 OPE ObjectComplementOf(CE))
```

Example Inference

- From

```
ObjectPropertyAssertion(a:hasPet a:Peter a:Brian)

ClassAssertion(a:Dog a:Brian)

ClassAssertion(
ObjectMaxCardinality(1 a:hasPet) a:Peter)
```

we can infer

```
ClassAssertion(
ObjectAllValuesFrom(a:hasPet a:Dog) a:Peter)
```

Individual Value Restriction

- An **individual value class expression** `ObjectHasValue(OPE a)` consists of an object property expression `OPE` and an individual `a`, and it contains all those individuals that are connected by `OPE` to `a`.

- Example:

```
ObjectHasValue(a:fatherOf a:Stewie)
```

- The above class expression is equivalent to the class expression

```
ObjectSomeValuesFrom(OPE ObjectOneOf(a)).
```

Example Inference

- From

```
ObjectPropertyAssertion(a:fatherOf
        a:Peter a:Stewie)
```

we can infer

```
ClassAssertion(
ObjectHasValue(a:fatherOf a:Stewie)
        a:Peter)
```

Self-Restriction

- A **self-restriction** `ObjectHasSelf(OPE)` consists of an object property expression `OPE`, and it contains all those individuals that are connected by `OPE` to themselves.

- Example:

```
ObjectHasSelf(a:likes)
```

Example Inference

- From

```
ObjectPropertyAssertion(a:likes
        a:Peter a:Peter)
```

we can infer

```
        ClassAssertion(
ObjectHasSelf(a:likes) a:Peter)
```

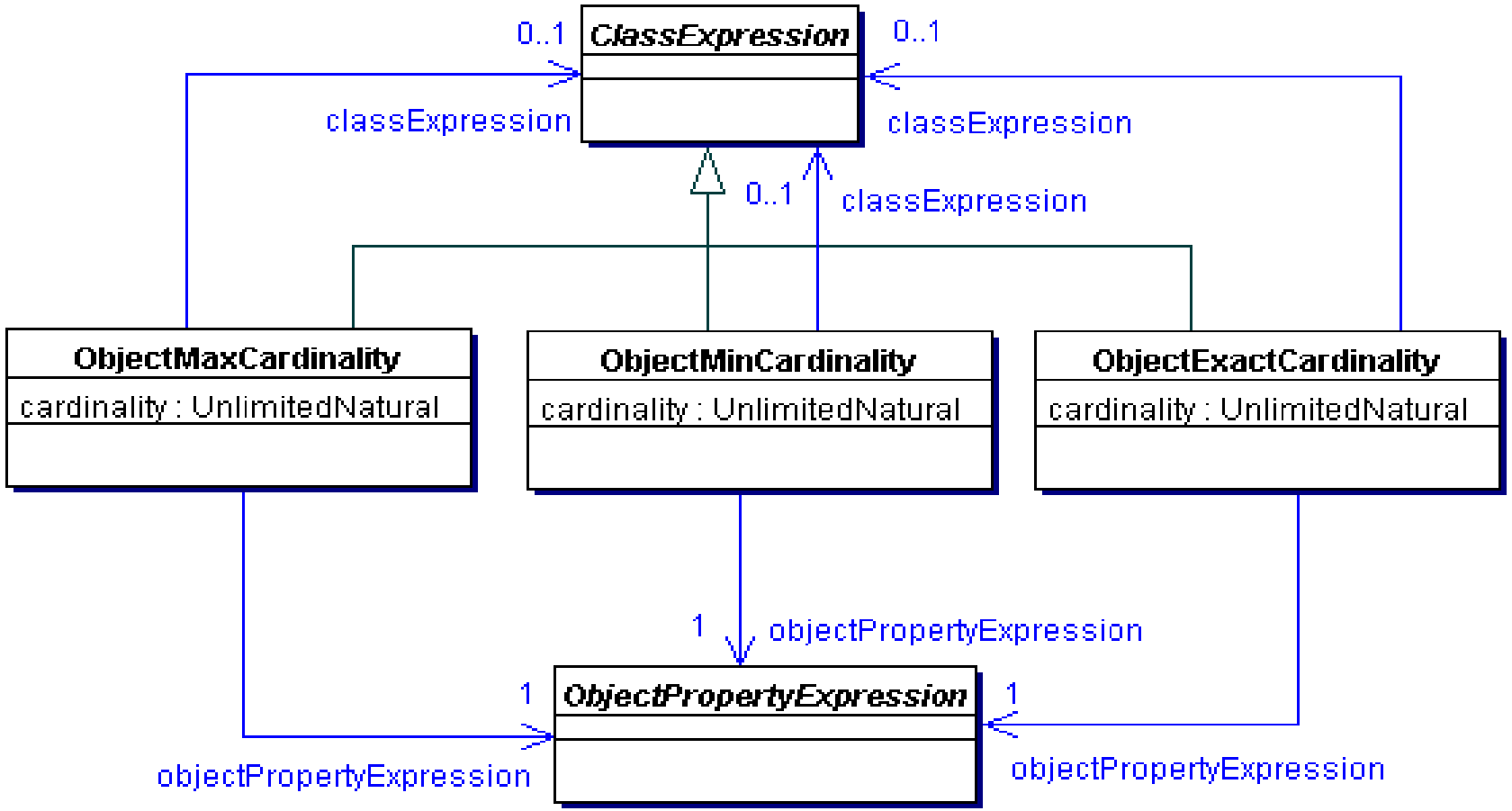
Ways to Form Class Expressions (cont'd)

- Class expressions can be formed by:
  - Applying the standard **Boolean connectives** to simpler class expressions or by **enumerating the individuals** that belong to an expression.
  - Placing **restrictions on object property expressions.**
  - Placing **restrictions on the cardinality** of object property expressions.
  - Placing **restrictions on data property expressions.**
  - Placing **restrictions on the cardinality** of data property expressions.

Object Property Cardinality Restrictions

- Object property cardinality restrictions are distinguished into:
  - **Qualified:** apply only to individuals that are connected by the object property expression and are instances of the qualifying class expression.
  - **Unqualified**: apply to all individuals that are connected by the object property expression (this is equivalent to the qualified case with the qualifying class expression equal to `owl:Thing`).

Object Property Cardinality Restrictions



Minimum Cardinality

- A **minimum cardinality expression** `ObjectMinCardinality(n OPE CE)` consists of a nonnegative integer `n`, an object property expression `OPE`, and a class expression `CE`, and it contains all those individuals that are connected by `OPE` to at least `n` different individuals that are instances of `CE`. If `CE` is missing, it is taken to be `owl:Thing`.

- Example:

```
ObjectMinCardinality(2 a:fatherOf a:Man)
```

Example Inference

- From

```
ObjectPropertyAssertion(a:fatherOf a:Peter a:Stewie)

ClassAssertion(a:Man a:Stewie)

ObjectPropertyAssertion(a:fatherOf a:Peter a:Chris)

ClassAssertion(a:Man a:Chris)

DifferentIndividuals(a:Chris a:Stewie)
```

we can infer

```
ClassAssertion(
ObjectMinCardinality(2 a:fatherOf a:Man) a:Peter)
```

Maximum Cardinality

- A **maximum cardinality expression** `ObjectMaxCardinality(n OPE CE)` consists of a nonnegative integer `n`, an object property expression `OPE`, and a class expression `CE`, and it contains all those individuals that are connected by `OPE` to at most n different individuals that are instances of `CE`. If `CE` is missing, it is taken to be `owl:Thing`.

- Example:

```
ObjectMaxCardinality(2 a:hasPet)
```

Example Inference

- From

```
ObjectPropertyAssertion(a:hasPet
a:Peter a:Brian)
```

```
ClassAssertion(ObjectMaxCardinality(1
a:hasPet) a:Peter)
```

we can infer

```
ClassAssertion(
ObjectMaxCardinality(2 a:hasPet)
a:Peter)
```

Example Inference

- From

```
ObjectPropertyAssertion(a:hasDaughter
        a:Peter a:Meg)

ObjectPropertyAssertion(a:hasDaughter
        a:Peter a:Megan)

ClassAssertion(ObjectMaxCardinality(1
        a:hasDaughter) a:Peter)
```

we can infer

```
SameIndividual(a:Meg a:Megan)
```

Exact Cardinality

- An **exact cardinality expression** `ObjectExactCardinality(n OPE CE)` consists of a nonnegative integer `n`, an object property expression `OPE`, and a class expression `CE`, and it contains all those individuals that are connected by `OPE` to exactly `n` different individuals that are instances of `CE`.

- Example:

```
ObjectExactCardinality(1 a:hasPet a:Dog)
```

- The above expression is equivalent to

```
ObjectIntersectionOf(
ObjectMinCardinality(n OPE CE)
ObjectMaxCardinality(n OPE CE)).
```

Example Inference

- From

```
ObjectPropertyAssertion(a:hasPet a:Peter a:Brian)

ClassAssertion(a:Dog a:Brian)

ClassAssertion(
ObjectAllValuesFrom(a:hasPet
ObjectUnionOf(ObjectOneOf(a:Brian)
ObjectComplementOf(a:Dog)))
a:Peter)
```

we can infer

```
ClassAssertion(ObjectExactCardinality(1 a:hasPet
a:Dog) a:Peter)
```

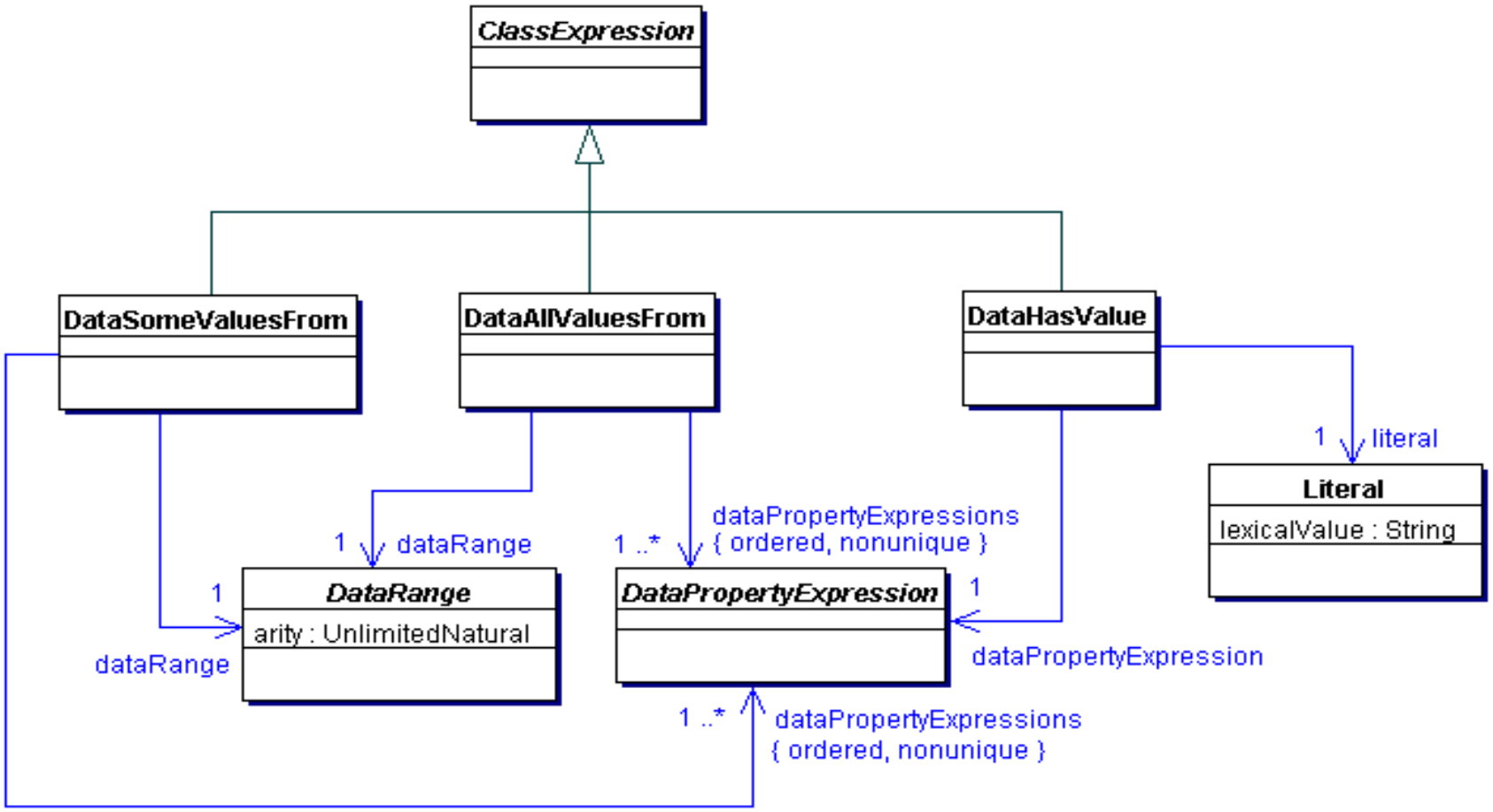
Ways to Form Class Expressions (cont'd)

- Class expressions can be formed by:
  - Applying the standard **Boolean connectives** to simpler class expressions or by **enumerating the individuals** that belong to an expression.
  - Placing **restrictions on object property expressions.**
  - Placing **restrictions on the cardinality** of object property expressions.
  - Placing **restrictions on data property expressions.**
  - Placing **restrictions on the cardinality** of data property expressions.

Data Property Restrictions

- Data property restrictions are similar to the restrictions on object property expressions.
- The main difference is that the expressions for existential and universal quantification allow for **n-ary data ranges**.
- Given the syntax for data ranges given earlier, only **unary** data ranges are supported.
- However, the specification aprovide the syntactic constructs needed to have **n-ary data ranges** e.g., sets of rectangles defined by appropriate geometric constraints.
- The “Data Range Extension: Linear Equations” W3C note proposes an extension to OWL 2 for defining n-ary data ranges in terms of linear (in)equations with rational coefficients. See http://www.w3.org/TR/owl2-dr-linear/ .

Data Property Restrictions



Existential Quantification

- An **existential class expression** `DataSomeValuesFrom(DPE1 ... DPEn DR)` consists of n data property expressions `DPEi,1≤i≤n,` and a data range `DR` whose arity must be `n`.

- Such a class expression contains all those individuals that are connected by `DPEi` to literals `lti,1≤i≤n`, such that the tuple `(lt1 ,...,ltn)` is in `DR`.

- Example:

```
DataSomeValuesFrom(a:hasAge
DatatypeRestriction(xsd:integer xsd:maxExclusive
"20"^^xsd:integer))
```

- A class expression of the form `DataSomeValuesFrom(DPE DR)` is equivalent to the class expression `DataMinCardinality(1 DPE DR)`.

Example Inference

- From

```
DataPropertyAssertion(a:hasAge a:Meg
        "17"^^xsd:integer)
```

we can infer

```
ClassAssertion(
DataSomeValuesFrom(a:hasAge
  DatatypeRestriction(xsd:integer
xsd:maxExclusive "20"^^xsd:integer))
a:Meg)
```

Universal Quantification

- A **universal class expression** `DataAllValuesFrom(DPE1 ... DPEn DR)` consists of `n` data property expressions `DPEi,1≤i≤n,` and a data range `DR` whose arity must be `n`.
- Such a class expression contains all those individuals that are connected by `DPEi` only to literals `lti,1≤i≤n,` such that each tuple `(lt1,...,ltn)` is in `DR`.
- Example:

```
DataAllValuesFrom(a:hasZIP xsd:integer)
```

- A class expression of the form `DataAllValuesFrom(DPE DR)` can be seen as a syntactic shortcut for the class expression `DataMaxCardinality(0 DPE DataComplementOf(DR))`.

Example Inference

- From

```
DataPropertyAssertion(a:hasZIP _:a1
        "02903"^^xsd:integer)

FunctionalDataProperty(a:hasZIP)
```

we can infer

```
ClassAssertion(
DataAllValuesFrom(a:hasZIP xsd:integer)
        _:a1)
```

Literal Value Restriction

- A **literal value class restriction** `DataHasValue(DPE lt)` consists of a data property expression `DPE` and a literal `lt`, and it contains all those individuals that are connected by `DPE` to `lt`.

- Example:

```
DataHasValue(a:hasAge "17"^^xsd:integer)
```

- Each such class expression is equivalent to the class expression

```
DataSomeValuesFrom(DPE DataOneOf(lt)).
```

Ways to Form Class Expressions (cont'd)

- Class expressions can be formed by:
  - Applying the standard **Boolean connectives** to simpler class expressions or by **enumerating the individuals** that belong to an expression.
  - Placing **restrictions on object property expressions.**
  - Placing **restrictions on the cardinality** of object property expressions.
  - Placing **restrictions on data property expressions.**
  - Placing **restrictions on the cardinality** of data property expressions.

Data Property Cardinality Restrictions

- Data property cardinality restrictions can be distinguished into:
  - **Qualified:** they only apply to literals that are connected by the data property expression and are in the qualifying data range.
  - **Unqualified:** they apply to all literals that are connected by the data property expression. This is equivalent to the qualified case with the qualifying data range equal to `rdfs:Literal`.

Minimum Cardinality

- A **minimum cardinality expression** `DataMinCardinality(n DPE DR)` consists of a nonnegative integer `n`, a data property expression `DPE`, and a unary data range `DR`, and it contains all those individuals that are connected by `DPE` to at least `n` different literals in `DR`. If `DR` is not present, it is taken to be `rdfs:Literal`.

- Example:

```
DataMinCardinality(2 a:hasName)
```

- There are similar definitions for `DataMaxCardinality(n DPE DR)` and `DataExactCardinality(n DPE DR)`.

Example Inference

- From

```
DataPropertyAssertion(a:hasName a:Meg
                      "Meg Griffin")

DataPropertyAssertion(a:hasName a:Meg
                      "Megan Griffin")
```

we can infer

```
ClassAssertion(
DataMinCardinality(2 a:hasName)
                   a:Meg)
```

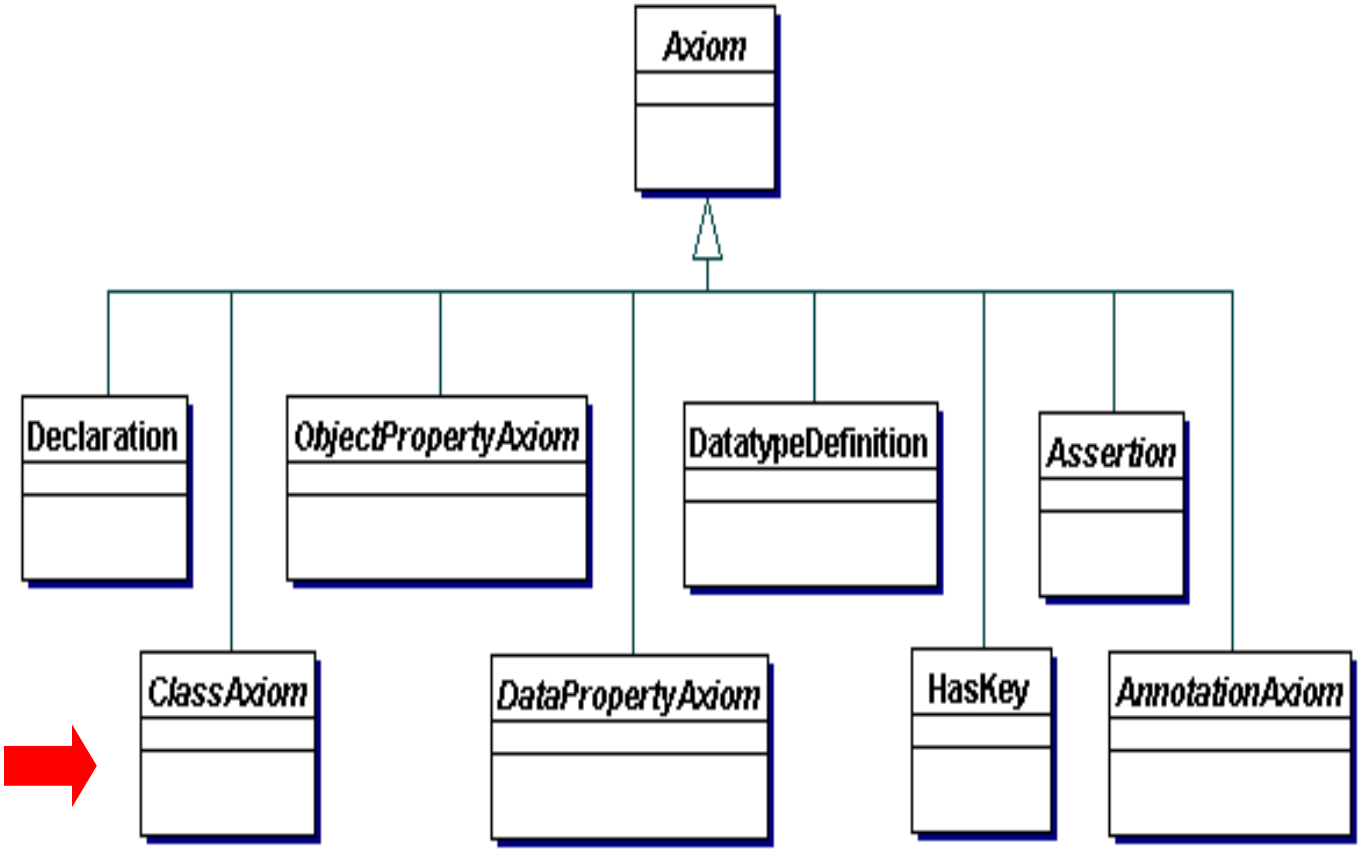
Maximum/Exact Cardinality

- Defined similarly.

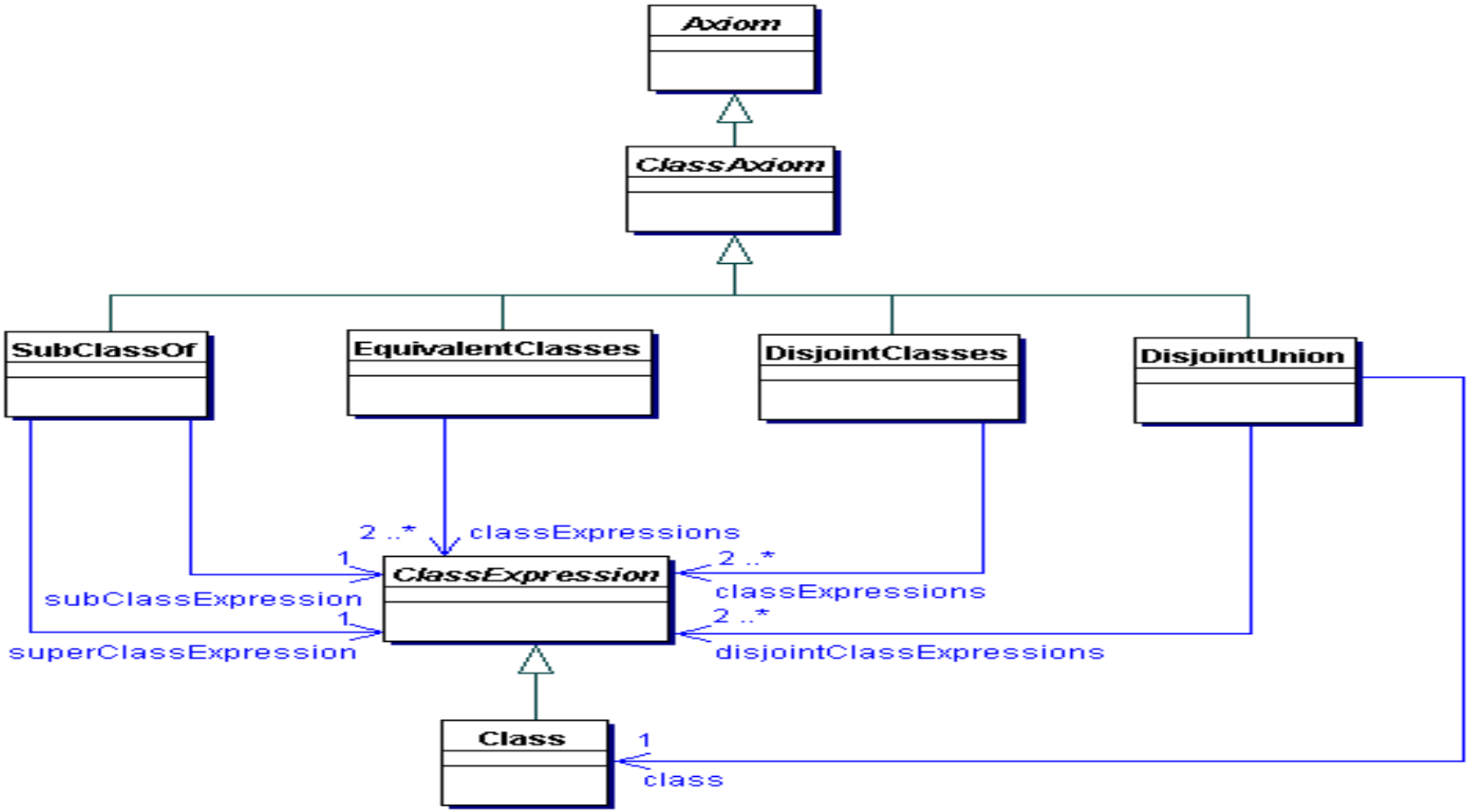
What Have we Achieved so far?

- We have explained what **the “things” that one can define** in OWL 2 are.
- Now let us see how to use these “things” to **represent knowledge about a domain.**
- In OWL 2 knowledge is represented by **axioms**: statements that say what is true in the domain of interest.

Axioms


Axiom
Declaration
ObjectPropertyAxiom
DatatypeDefinition
Assertion
ClassAxiom
DataPropertyAxiom
HasKey
AnnotationAxiom

Class Expression Axioms



Subclass Axioms

- A **subclass axiom** `SubClassOf(CE1 CE2)` states that the class expression `CE1` is a subclass of the class expression `CE2`.

- Example:

```
SubClassOf(a:Child a:Person)
```

- The properties known from RDFS for `SubClassOf` hold here as well:
  - Reflexivity
  - Transitivity
  - If `x` is an instance of class `A` and class `A` is a subclass of class `B`, then `x` is an instance of `B` as well.

Example Inferences

- From

```
SubClassOf(a:Baby a:Child)
SubClassOf(a:Child a:Person)

ClassAssertion(a:Baby a:Stewie)
```

we can infer

```
SubClassOf(a:Baby a:Person)

ClassAssertion(a:Child a:Stewie)
ClassAssertion(a:Person a:Stewie)
```

Example Inferences

- From

```
SubClassOf(a:PersonWithChild
ObjectSomeValuesFrom(a:hasChild
ObjectUnionOf(a:Boy a:Girl)))

SubClassOf(a:Boy a:Child)

SubClassOf(a:Girl a:Child)

SubClassOf(ObjectSomeValuesFrom(a:hasChild a:Child)
a:Parent)
```

we can infer

```
SubClassOf(a:PersonWithChild a:Parent)
```

Equivalent Classes

- An **equivalent classes** axiom `EquivalentClasses(CE1 ... CEn)` states that all of the class expressions `CEi,1≤i≤n,` are semantically equivalent to each other.

- Example:

```
EquivalentClasses(a:Boy
ObjectIntersectionOf(a:Child a:Male))
```

- An axiom `EquivalentClasses(CE1 CE2)` is equivalent to the conjunction of the following two axioms:

```
SubClassOf(CE1 CE2)
SubClassOf(CE2 CE1)
```

Example Inferences

- From

```
EquivalentClasses(a:Boy
ObjectIntersectionOf(a:Child a:Male))

ClassAssertion(a:Child a:Chris)

ClassAssertion(a:Male a:Chris)
```

we can infer

```
ClassAssertion(a:Boy a:Chris)
```

Example Inferences

- From

```
EquivalentClasses(a:MongrelOwner
ObjectSomeValuesFrom(a:hasPet a:Mongrel))

EquivalentClasses(a:DogOwner ObjectSomeValuesFrom(a:hasPet
a:Dog))

SubClassOf(a:Mongrel a:Dog)

ClassAssertion(a:MongrelOwner a:Peter)
```

we can infer

```
SubClassOf(a:MongrelOwner a:DogOwner)

ClassAssertion(a:DogOwner a:Peter)
```

Disjoint Classes

- A **disjoint classes axiom** `DisjointClasses(CE1 ... CEn)` states that all of the class expressions `CEi`, `1≤i≤n`, are pairwise disjoint.

- Example:

```
DisjointClasses(a:Boy a:Girl)
```

- An axiom `DisjointClasses(CE1 CE2)` is equivalent to the following axiom:

```
SubClassOf(CE1 ObjectComplementOf(CE2))
```

Disjoint Union of Classes

- A **disjoint union axiom** `DisjointUnion(C CE1 ... CEn)` states that a class `C` is a disjoint union of the class expressions `CEi, 1≤i≤ n,` all of which are pairwise disjoint.

- Such axioms are sometimes referred to as **covering axioms**, as they state that the extensions of all `CEi` exactly cover the extension of `C`.

- Example:

```
DisjointUnion(a:Child a:Boy a:Girl)
```

- Each such axiom is equivalent to the conjunction of the following two axioms:

```
EquivalentClasses(C ObjectUnionOf(CE1 ... CEn))

DisjointClasses(CE1 ... CEn)
```

Example Inferences

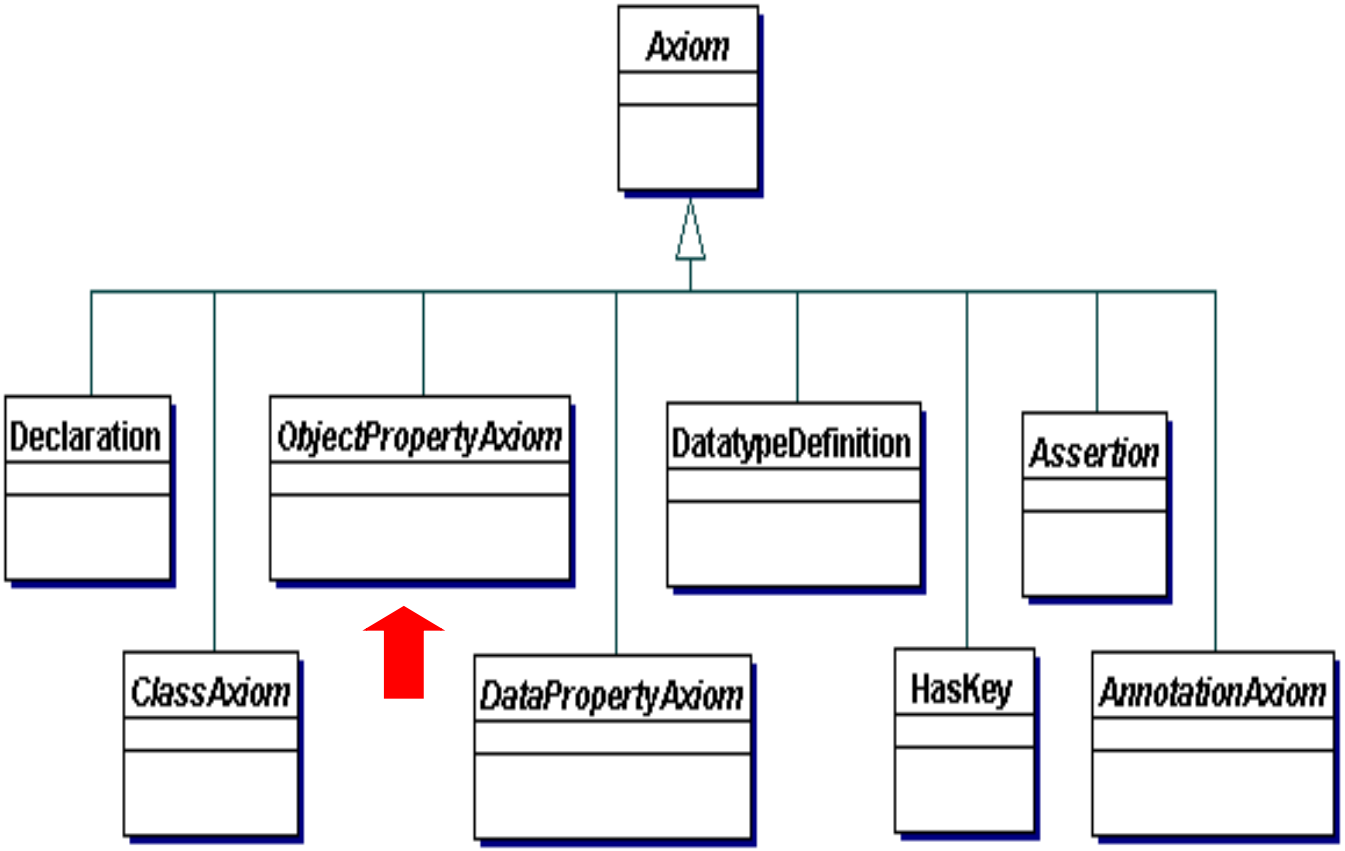
- From

```
DisjointUnion(a:Child a:Boy a:Girl)

ClassAssertion(a:Child a:Stewie)

ClassAssertion(ObjectComplementOf(a:Girl)
a:Stewie)
```

we can infer

```
ClassAssertion(a:Boy a:Stewie)
```

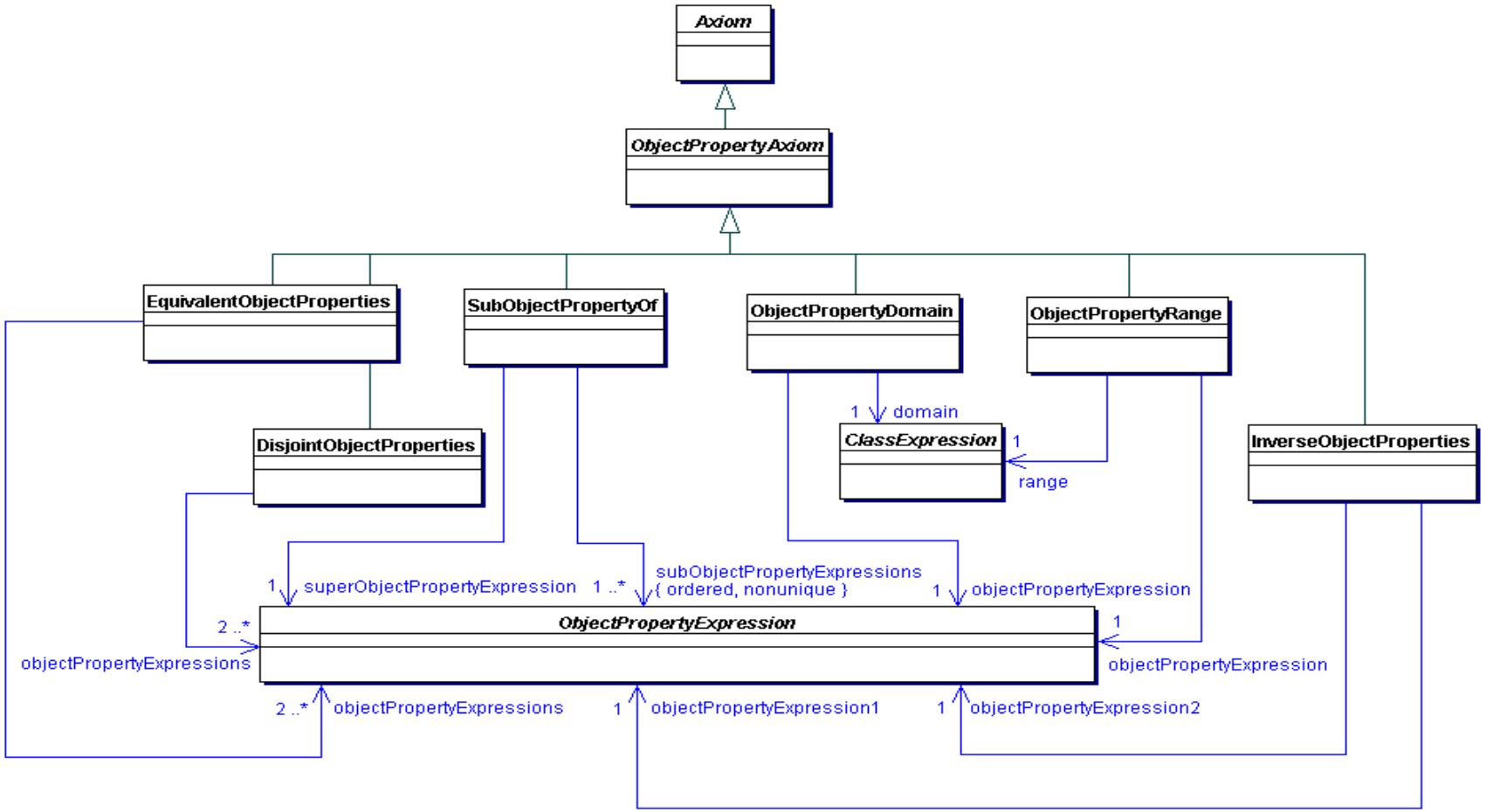
Axioms (cont'd)



Object Property Axioms

- OWL 2 provides axioms that can be used to **characterize and establish relationships between object property expressions.**

Object Property Axioms



Object Subproperty Axioms

- Object subproperty axioms are analogous to subclass axioms.
- The **basic form** of an object subproperty axiom is `SubObjectPropertyOf(OPE1 OPE2)`.
- This axiom states that the object property expression `OPE1` is a subproperty of the object property expression `OPE2` — that is, if an individual `x` is connected by `OPE1` to an individual `y`, then `x` is also connected by `OPE2` to `y`.
- `SubObjectPropertyOf` is a reflexive and transitive relation.

Example Inferences

- From

```
SubObjectPropertyOf(a:hasDog a:hasPet)

ObjectPropertyAssertion(a:hasDog a:Peter
                        a:Brian)
```

we can infer

```
ObjectPropertyAssertion(a:hasPet a:Peter
                        a:Brian)
```

Object Subproperty Axioms: Inclusions with Property Chains

- If `OPE1, …, OPEn` are object properties then
  `OPE1 … OPEn` is called an **object property chain**.

- The more complex form of object subproperty axioms is

```
SubObjectPropertyOf(
ObjectPropertyChain(OPE1 ... OPEn) OPE).
```

- This axiom states that, if an individual `x` is connected by a sequence of object property expressions `OPE1, ..., OPEn` with an individual `y`, then `x` is also connected with `y` by the object property expression `OPE`.

- These axioms are known as **complex role inclusions** in the DL literature.

Example Inferences

- From

```
SubObjectPropertyOf(
ObjectPropertyChain(a:hasMother a:hasSister)
a:hasAunt)

ObjectPropertyAssertion(a:hasMother a:Stewie a:Lois)

ObjectPropertyAssertion(a:hasSister a:Lois
a:Carol)
```

we can infer

```
ObjectPropertyAssertion(a:hasAunt a:Stewie a:Carol)
```

Equivalent Object Properties

- An **equivalent object properties** axiom `EquivalentObjectProperties(OPE1 ... OPEn)` states that all of the object property expressions `OPEi,1≤i≤n,` are semantically equivalent to each other.

- The axiom `EquivalentObjectProperties(OPE1 OPE2)` is equivalent to the following two axioms:

```
SubObjectPropertyOf(OPE1 OPE2)
SubObjectPropertyOf(OPE2 OPE1)
```

Example Inferences

- From

```
EquivalentObjectProperties(a:hasBrother a:hasMaleSibling)

    ObjectPropertyAssertion(a:hasBrother a:Chris a:Stewie)

ObjectPropertyAssertion(a:hasMaleSibling a:Stewie a:Chris)
```

we can infer

```
ObjectPropertyAssertion(a:hasMaleSibling a:Chris a:Stewie)

  ObjectPropertyAssertion(a:hasBrother a:Stewie a:Chris)
```

Disjoint Object Properties

- A **disjoint object properties** axiom `DisjointObjectProperties(OPE1 ... OPEn)` states that all of the object property expressions `OPEi,1≤i≤n`, are pairwise disjoint.

- Example:

```
DisjointObjectProperties(a:hasFather
                         a:hasMother)
```

Inverse Object Properties

- An **inverse object properties** axiom `InverseObjectProperties(OPE1 OPE2)` states that the object property expression `OPE1` is an inverse of the object property expression `OPE2`.

- Each such axiom is equivalent with the following:

```
EquivalentObjectProperties(OPE1
    ObjectInverseOf(OPE2))
```

Example Inferences

- From

```
InverseObjectProperties(a:hasFather a:fatherOf)

ObjectPropertyAssertion(a:hasFather a:Stewie
                        a:Peter)

ObjectPropertyAssertion(a:fatherOf a:Peter a:Chris)
```

we can infer

```
ObjectPropertyAssertion(a:fatherOf a:Peter a:Stewie)

ObjectPropertyAssertion(a:hasFather a:Chris a:Peter)
```

Object Property Domain Axioms

- An **object property domain** axiom `ObjectPropertyDomain(OPE CE)` states that the domain of the object property expression `OPE` is the class expression `CE` — that is, if an individual `x` is connected by `OPE` with some other individual, then `x` is an instance of `CE`.

- Each such axiom is equivalent to the following axiom:

```
SubClassOf(ObjectSomeValuesFrom(OPE
           owl:Thing) CE)
```

Example Inferences

- From

```
ObjectPropertyDomain(a:hasDog a:Person)

ObjectPropertyAssertion(a:hasDog a:Peter
                        a:Brian)
```

we can infer

```
ClassAssertion(a:Person a:Peter)
```

Object Property Range Axioms

- An **object property range axiom** `ObjectPropertyRange(OPE CE)` states that the range of the object property expression `OPE` is the class expression `CE` — that is, if some individual is connected by `OPE` with an individual `x`, then `x` is an instance of `CE`.

- Each such axiom is equivalent to the following axiom:

```
SubClassOf(owl:Thing ObjectAllValuesFrom(OPE
                     CE))
```

Example Inferences

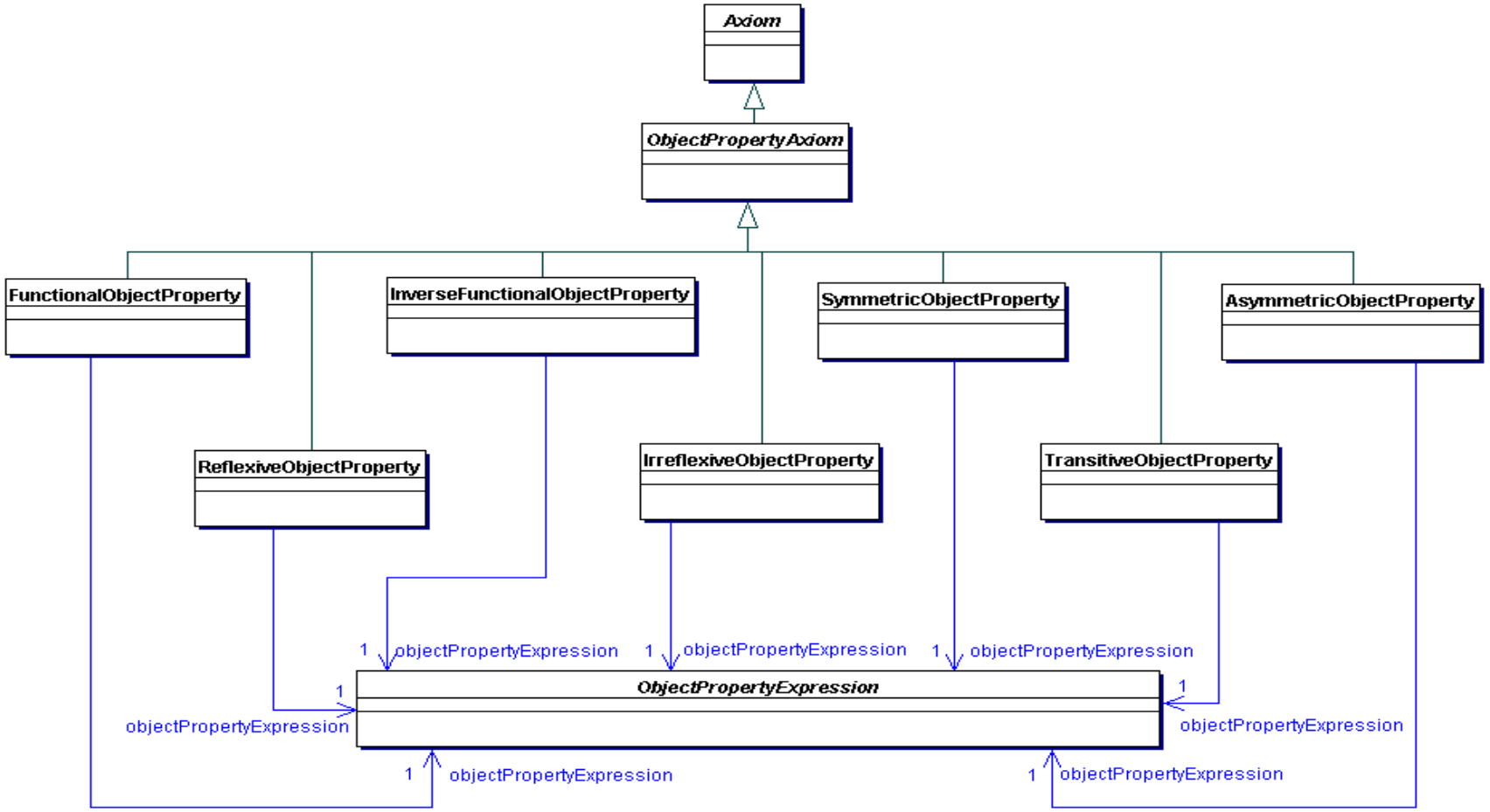
- From

```
ObjectPropertyRange(a:hasDog a:Dog)

ObjectPropertyAssertion(a:hasDog
a:Peter a:Brian)
```

we can infer

```
ClassAssertion(a:Dog a:Brian)
```

Object Property Axioms (cont'd)



Functional Object Properties

- An object property functionality axiom `FunctionalObjectProperty(OPE)` states that the object property expression `OPE` is **functional** — that is, for each individual `x`, there can be at most one distinct individual `y` such that `x` is connected by `OPE` to `y`.

- Each such axiom is equivalent to the following axiom:

```
SubClassOf(owl:Thing
ObjectMaxCardinality(1 OPE))
```

Example Inferences

- From

```
FunctionalObjectProperty(a:hasFather)

ObjectPropertyAssertion(a:hasFather a:Stewie
                        a:Peter)

ObjectPropertyAssertion(a:hasFather a:Stewie
                        a:Peter_Griffin)
```

we can infer

```
SameIndividual(a:Peter a:Peter_Griffin)
```

Inverse-Functional Object Properties

- An object property inverse functionality axiom `InverseFunctionalObjectProperty(OPE)` states that the object property expression `OPE` is i**nverse-functional** — that is, for each individual `x`, there can be at most one individual `y` such that `y` is connected by `OPE` with `x`.

- Each such axiom is equivalent to the following axiom:

```
SubClassOf(owl:Thing ObjectMaxCardinality(1
                ObjectInverseOf(OPE)))
```

Example Inferences

- From

```
InverseFunctionalObjectProperty(a:fatherOf)

ObjectPropertyAssertion(a:fatherOf a:Peter a:Stewie)

ObjectPropertyAssertion(a:fatherOf a:Peter_Griffin
a:Stewie)
```

we can infer

```
SameIndividual(a:Peter a:Peter_Griffin)
```

Reflexive Object Properties

- An object property reflexivity axiom `ReflexiveObjectProperty(OPE)` states that the object property expression `OPE` is **reflexive** — that is, each individual is connected by `OPE` to itself.

- Each such axiom is equivalent to the following axiom:

```
SubClassOf(owl:Thing ObjectHasSelf(
                        OPE))
```

Example Inferences

- From

```
ReflexiveObjectProperty(a:knows)
  ClassAssertion(a:Person a:Peter)
```

we can infer

```
ObjectPropertyAssertion(a:knows
        a:Peter a:Peter)
```

Irreflexive Object Properties

- An object property irreflexivity axiom `IrreflexiveObjectProperty(OPE)` states that the object property expression `OPE` is **irreflexive** — that is, no individual is connected by `OPE` to itself.

- Each such axiom is equivalent to the following axiom:

```
SubClassOf(ObjectHasSelf(OPE)
           owl:Nothing)
```

Symmetric Object Properties

- An object property symmetry axiom `SymmetricObjectProperty(OPE)` states that the object property expression `OPE` is **symmetric** — that is, if an individual `x` is connected by `OPE` to an individual `y`, then `y` is also connected by `OPE` to `x`.

- Example:

```
SymmetricObjectProperty(a:friend)
```

- Each such axiom is equivalent to the following axiom:

```
SubObjectPropertyOf(OPE
  ObjectInverseOf(OPE))
```

Asymmetric Object Properties

- An object property asymmetry axiom `AsymmetricObjectProperty(OPE)` states that the object property expression `OPE` is **asymmetric** — that is, if an individual `x` is connected by `OPE` to an individual `y`, then `y` cannot be connected by OPE to `x`.

- Example

```
AsymmetricObjectProperty(a:parentOf)
```

Transitive Object Properties

- An object property transitivity axiom `TransitiveObjectProperty(OPE)` states that the object property expression `OPE` is **transitive** — that is, if an individual `x` is connected by `OPE` to an individual `y` that is connected by `OPE` to an individual `z`, then `x` is also connected by `OPE` to `z`.

- Each such axiom is equivalent to the following axiom:

```
SubObjectPropertyOf(ObjectPropertyChain(OPE
                    OPE) OPE)
```

Example Inferences

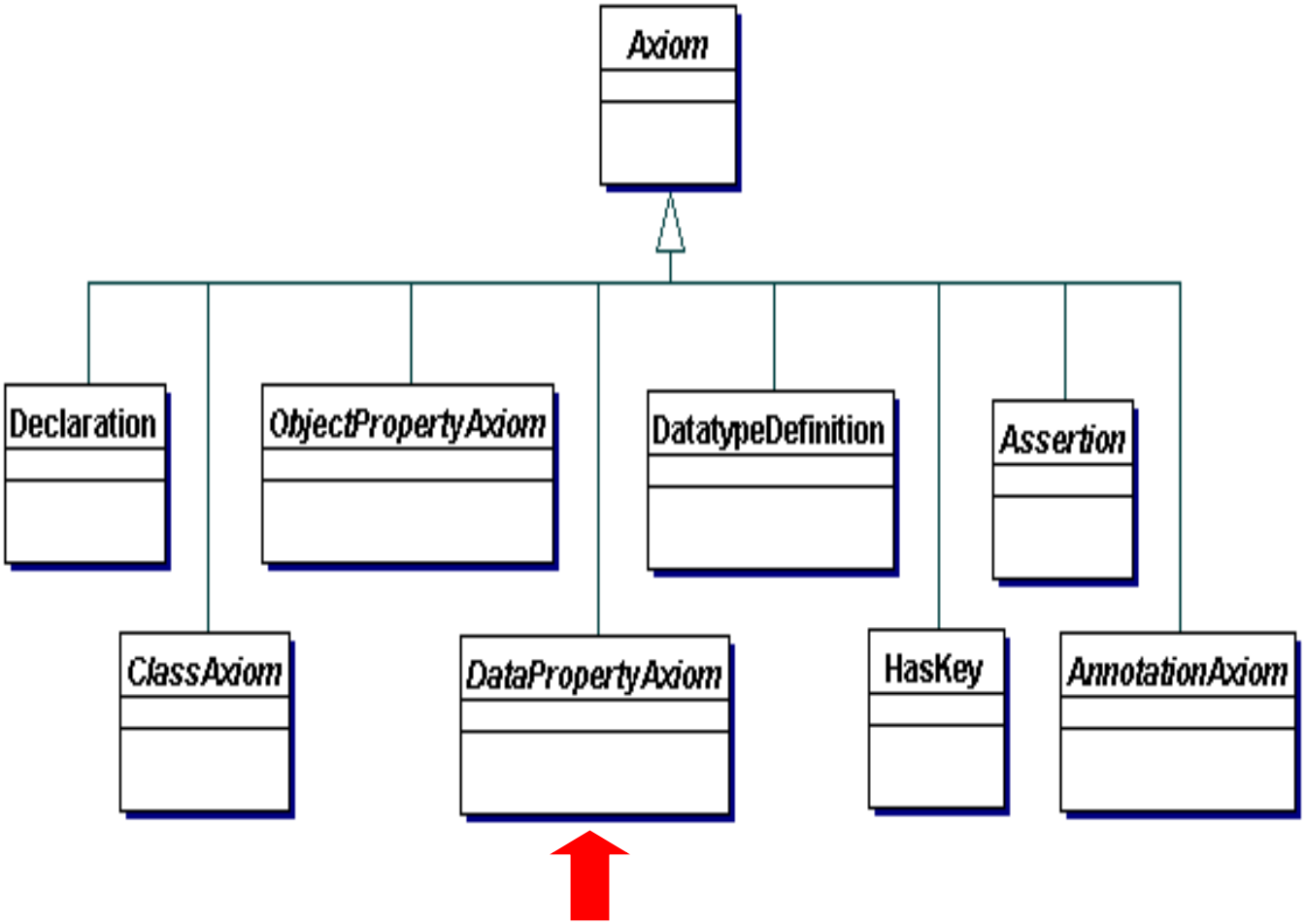
- From

```
TransitiveObjectProperty(a:ancestorOf)

ObjectPropertyAssertion(a:ancestorOf a:Carter
a:Lois)

ObjectPropertyAssertion(a:ancestorOf a:Lois a:Meg)
```

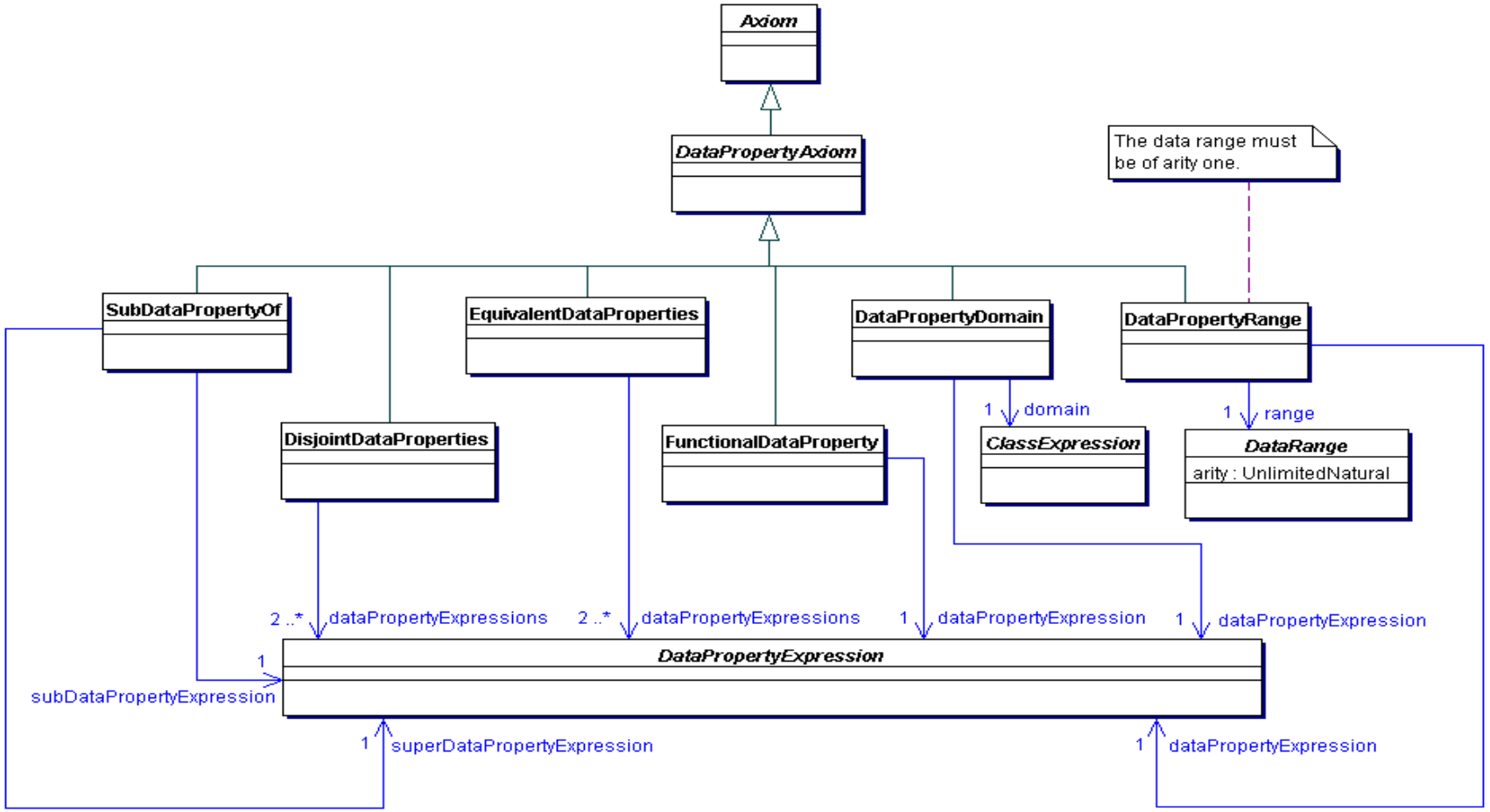
we can infer

```
ObjectPropertyAssertion(a:ancestorOf a:Carter a:Meg)
```

Axioms (cont'd)



Data Property Axioms



Data Property Axioms (cont'd)

- OWL 2 also provides for data property axioms. Their structure and semantics is similar to the corresponding object property axioms.

- We will not present data property axioms in detail. We will only give some examples.

Examples

- From

```
SubDataPropertyOf(a:hasLastName a:hasName)

DataPropertyAssertion(a:hasLastName a:Peter
                      "Griffin")
```

we can infer

```
DataPropertyAssertion(a:hasName a:Peter
                      "Griffin")
```

Examples (cont'd)

- The ontology

```
FunctionalDataProperty(a:hasAge)

DataPropertyAssertion(a:hasAge a:Meg
        "17"^^xsd:integer)

DataPropertyAssertion(a:hasAge a:Meg
        "17.0"^^xsd:decimal)

DataPropertyAssertion(a:hasAge a:Meg "+17"^^xsd:int)
```

is consistent because the different age literals given map to the same value.

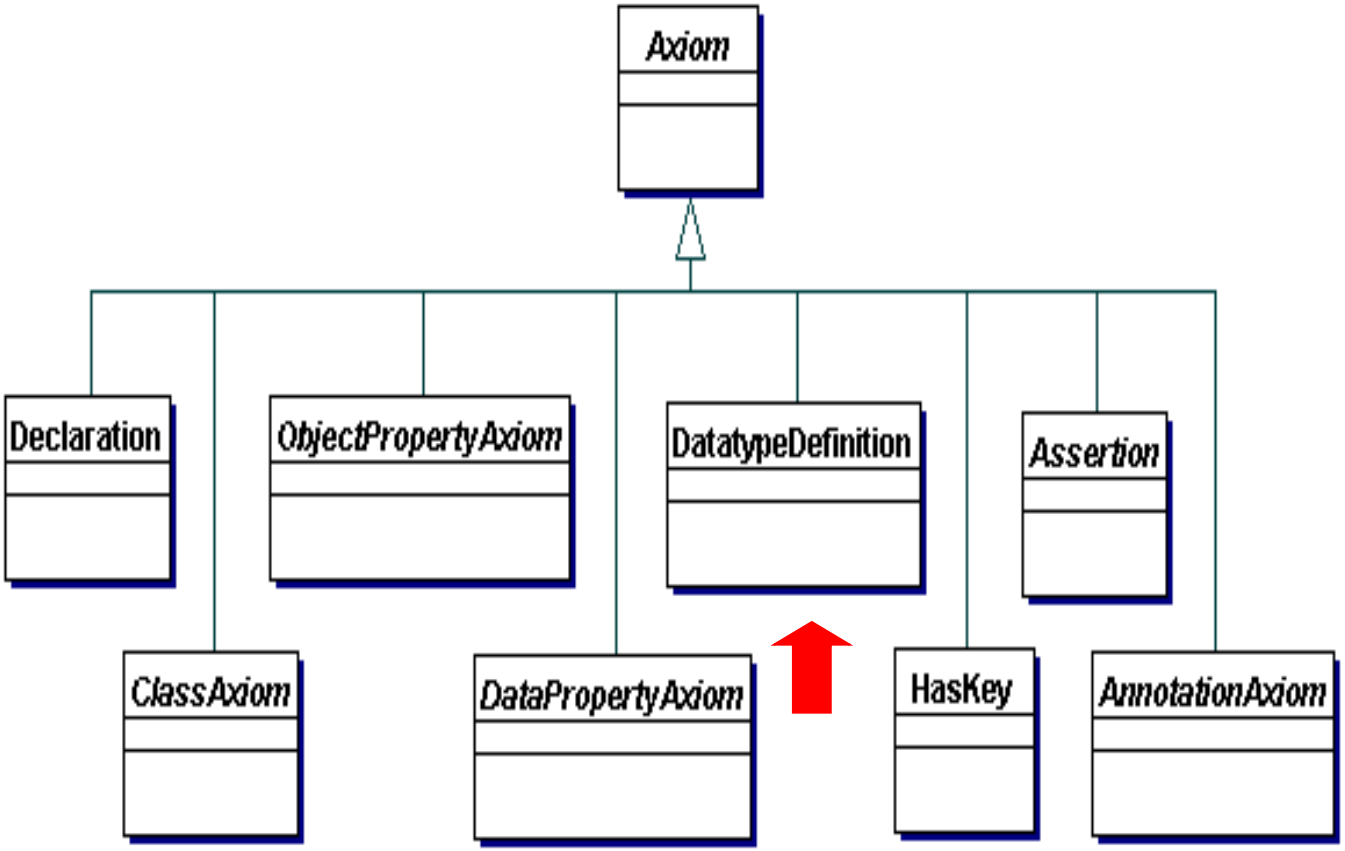
Examples (cont'd)

- The ontology

```
FunctionalDataProperty(a:numberOfChildren)

DataPropertyAssertion(a:numberOfChildren
          a:Meg "+0"^^xsd:float)

DataPropertyAssertion(a:numberOfChildren
          a:Meg "-0"^^xsd:float)
```

is unsatisfiable because literals `"+0"^^xsd:float` and `"-0"^^xsd:float` are mapped to distinct data values `+0` and `-0` in the value space of `xsd:float`; these data values are equal, but not identical.

Axioms (cont'd)



Datatype Definitions

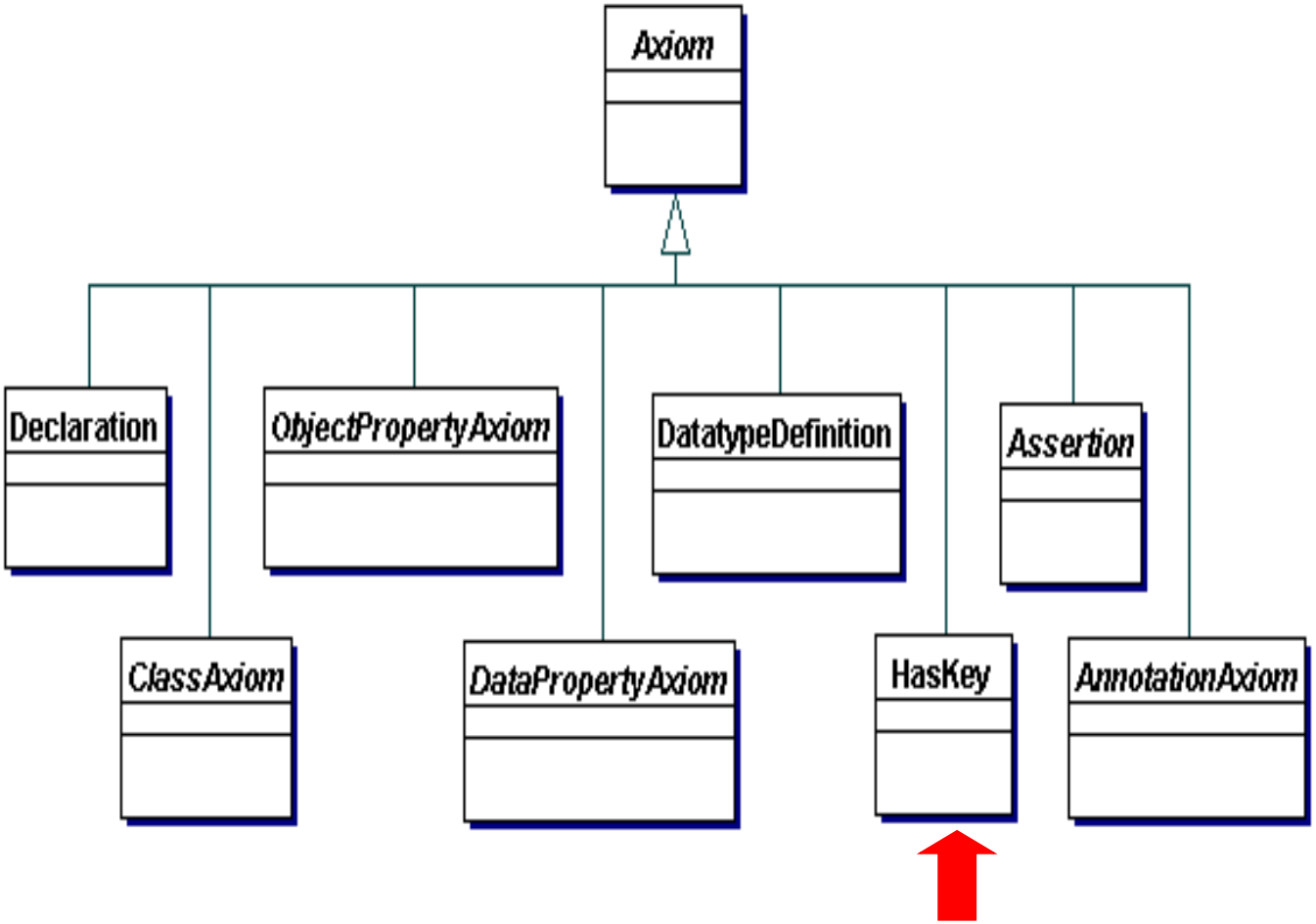
- A **datatype definition** `DatatypeDefinition(DT DR)` defines a new datatype `DT` as being semantically equivalent to the data range `DR`; the latter must be a unary data range.

- The datatypes defined by datatype definition axioms **support no facets** so they must not occur in datatype restrictions.

Example

```
DatatypeDefinition(a:SSN
DatatypeRestriction(xsd:string
xsd:pattern "[0-9]{3}-[0-9]{2}-
[0-9]{4}"))

DataPropertyRange(a:hasSSN
a:SSN)
```

Axioms (cont'd)



Keys

- A **key axiom**

  `HasKey(CE (OPE1 ... OPEm) (DPE1 ... DPEn))`

  states that each **named** instance of the class expression `CE` is uniquely identified by the object property expressions `OPEi` and/or the data property expressions `DPEj`.

- In this case, no two **distinct**, **named** instances of `CE` can coincide on the values of all object property expressions `OPEi` and all data property expressions `DPEj`.

- A key axiom of the form `HasKey(owl:Thing (OPE) ())` is similar to the axiom `InverseFunctionalObjectProperty(OPE)`. Their main difference is that the former axiom is applicable only to individuals that are explicitly named in an ontology, while the latter axiom is also applicable to unnamed individuals.

Example Inferences

- From

```
HasKey(owl:Thing () ( a:hasSSN))

DataPropertyAssertion(a:hasSSN a:Peter "123-45-6789")

DataPropertyAssertion(a:hasSSN a:Peter_Griffin "123-45-6789")
```

we can infer

```
SameIndividual(a:Peter a:Peter_Griffin)
```

Example Inferences

- From

```
HasKey(a:GriffinFamilyMember () (a:hasName))

DataPropertyAssertion(a:hasName a:Peter "Peter")

ClassAssertion(a:GriffinFamilyMember a:Peter)

DataPropertyAssertion(a:hasName a:Peter_Griffin "Peter")

ClassAssertion(a:GriffinFamilyMember a:Peter_Griffin)

DataPropertyAssertion(a:hasName a:StPeter "Peter")
```

we can infer

```
SameIndividual(a:Peter a:Peter_Griffin)
```

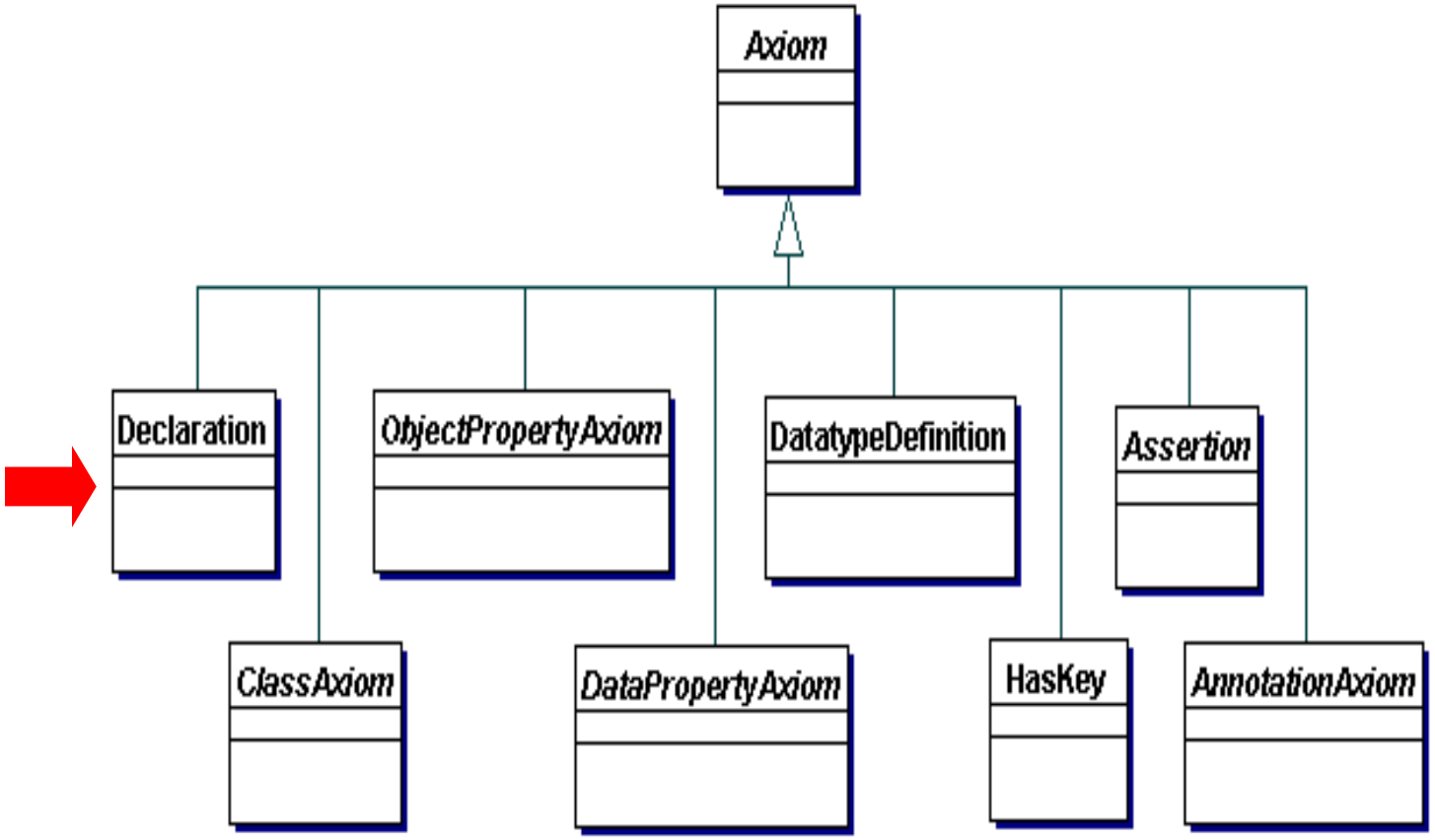
Example

- The ontology

```
HasKey(a:GriffinFamilyMember () (a:hasName))

DataPropertyAssertion(a:hasName a:Peter "Peter")

DataPropertyAssertion(a:hasName a:Peter "Kichwa-
                      Tembo")

ClassAssertion(a:GriffinFamilyMember a:Peter)
```

is consistent because a key axiom does not make all the properties used in it functional.

Axioms (cont'd)



Declarations

- In an OWL 2 ontology, the entities (individuals, classes, properties) used can be, and sometimes even needs to be, declared.

- Declarations are **nonlogical axioms**. They have no semantics but can allow OWL 2 tools to catch errors.

- Declarations are **optional**. But in OWL DL classes, datatypes and properties of various kinds **need to be declared** as such.

BNF for Entity Declarations

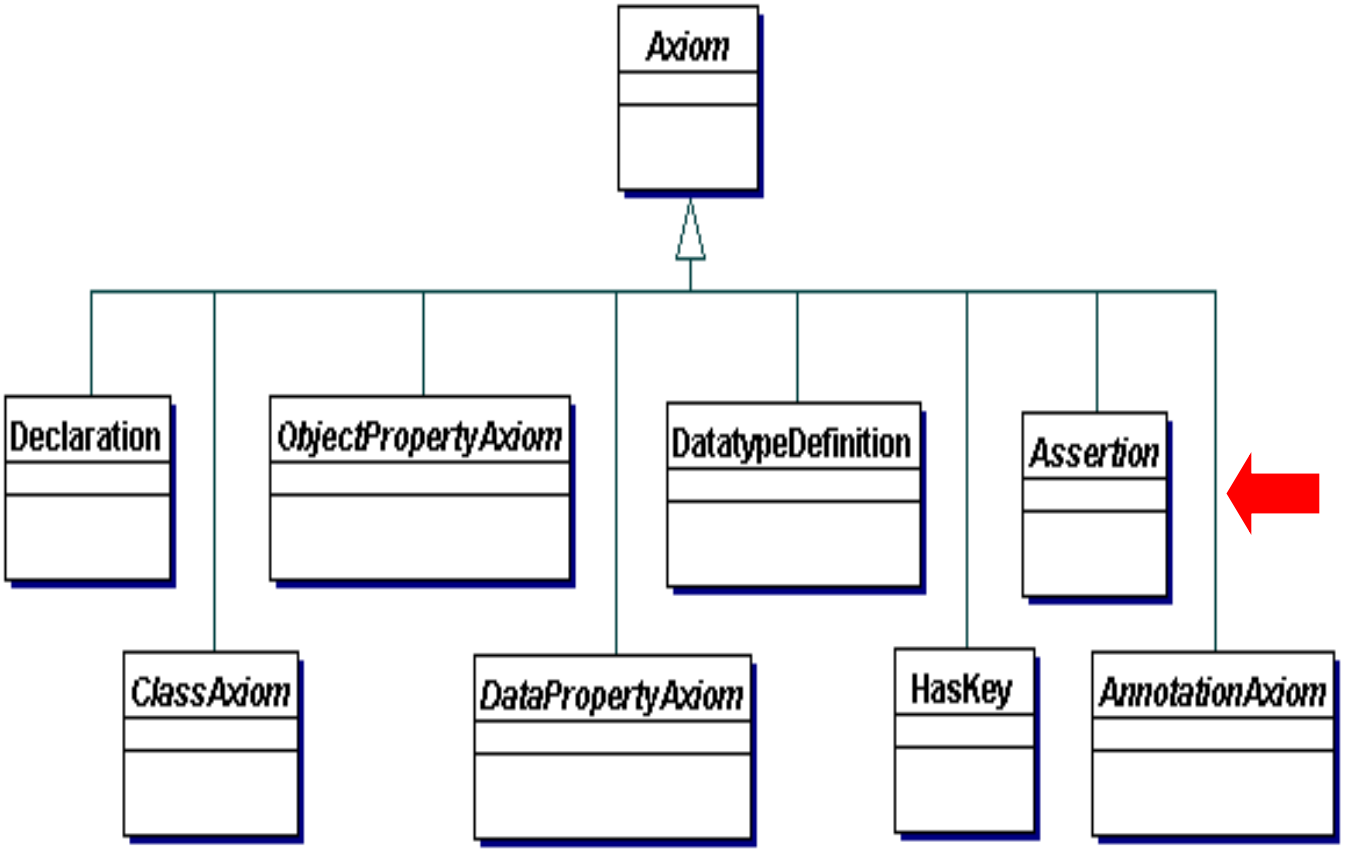
Declaration := 'Declaration' '(' axiomAnnotations
Entity ')‘

Entity :=
'Class' '(' Class ')' |
'Datatype' '(' Datatype ')' |
'ObjectProperty' '(' ObjectProperty ')' |
'DataProperty' '(' DataProperty ')' |
'AnnotationProperty' '(' AnnotationProperty
')' |
'NamedIndividual' '(' NamedIndividual ')'

Example

```
Declaration(Class(a:Person))
Declaration(NamedIndividual(a:Peter))

ClassAssertion(a:Person a:Peter)
```

Axioms (cont'd)



Assertions

- OWL 2 supports a rich set of axioms for stating **assertions about individuals**:
  - Individual equality
  - Individual inequality
  - Class assertion
  - Positive object property assertion
  - Negative object property assertion
  - Positive data property assertion
  - Negative data property assertion

- Assertions are often also called **facts**. They are part of the ABox in DLs.

Individual Equality Axiom

- An **individual equality axiom** `SameIndividual(a1 ... an)` states that all of the individuals `ai, 1≤i≤n,` are equal to each other.

Example Inference

- From

```
SameIndividual(a:Meg a:Megan)

ObjectPropertyAssertion(a:hasBrother a:Meg
a:Stewie)
```

we can infer

```
ObjectPropertyAssertion(a:hasBrother a:Megan
a:Stewie)
```

Individual Inequality Axiom

- An **individual inequality axiom** `DifferentIndividuals(a1 ... an)` states that all of the individuals `ai`, `1≤i≤n`, are different from each other.

- Example:

```
DifferentIndividuals(a:Peter a:Meg
        a:Chris a:Stewie)
```

Class Assertions

- A **class assertion** `ClassAssertion(CE a)` states that the individual `a` is an instance of the class expression `CE`.

- Example:

```
ClassAssertion(a:Dog a:Brian)
```

Object Property Assertions

- A **positive object property assertion** `ObjectPropertyAssertion(OPE a1 a2)` states that the individual `a1` is connected by the object property expression `OPE` to the individual `a2`.

- A **negative object property assertion** `NegativeObjectPropertyAssertion(OPE a1 a2)` states that the individual `a1` is not connected by the object property expression `OPE` to the individual `a2`.

Examples

```
ObjectPropertyAssertion(a:hasDog
a:Peter a:Brian)
```

```
NegativeObjectPropertyAssertion(
a:hasSon a:Peter a:Meg)
```

Data Property Assertions

- A **positive data property assertion** `DataPropertyAssertion(DPE a lt)` states that the individual `a` is connected by the data property expression `DPE` to the literal `lt`.

- A **negative data property** assertion `NegativeDataPropertyAssertion(DPE a lt)` states that the individual `a` is not connected by the data property expression `DPE` to the literal `lt`.

Example Inference

- From

```
DataPropertyAssertion(a:hasAge a:Meg "17"^^xsd:integer)

SubClassOf(
       DataSomeValuesFrom(a:hasAge
          DatatypeRestriction(xsd:integer
             xsd:minInclusive "13"^^xsd:integer
             xsd:maxInclusive "19"^^xsd:integer
          )
        )
        a:Teenager
   )
```

we can infer

```
ClassAssertion(a:Teenager a:Meg)
```

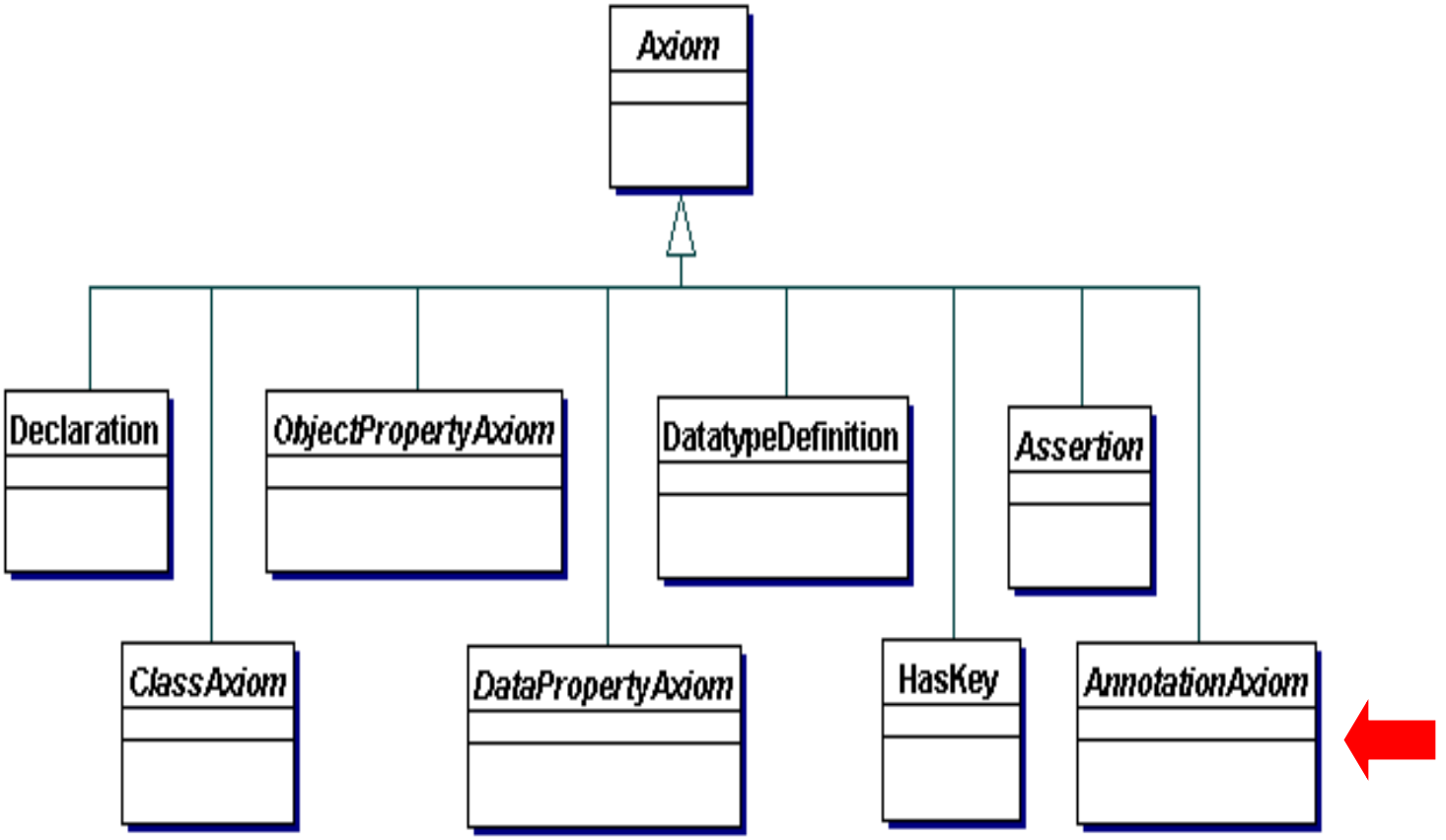
Annotations

- OWL 2 applications often need **ways to associate additional information** with ontologies, entities, and axioms. To this end, OWL 2 provides for **annotations** on ontologies, axioms, and entities.

- Annotations are **first-class citizens** in OWL 2; their structure is independent of the underlying syntax and they are different than comments that a syntax (e.g., OWL XML) might allow.

- Annotations have no formal semantics, thus they do not participate in the meaning of an ontology (under the OWL 2 direct semantics).

Axioms (cont'd)



Annotation of Entities and Anonymous Individuals

- The axiom `AnnotationAssertion(AP as av)` states that the annotation subject `as` is annotated with the annotation property `AP` (user defined or built-in) and the annotation value `av`.

- `as` can be an **entity** (i.e., individual, class or property) or an **anonymous individual**.

- Example:

```
AnnotationAssertion(rdfs:label a:Person
   "Represents the set of all people.")
```

Annotations of Axioms, Annotations and Ontologies

- OWL 2 also provides the construct `Annotation({A} AP v)` where `AP` is an annotation property (user defined or built-in), `v` is a literal, an IRI, or an anonymous individual and `{A}` are 0 or more annotations.

- The above construct can be used for annotations of **axioms** and **ontologies**. It can also be used for **annotations of annotations themselves.**

Examples

```
SubClassOf(
Annotation(rdfs:comment "Persons
  are humans.") a:Person a:Human)
```

Examples (cont'd)

```
Prefix(ex:=<http://www.example.com/ontology1#>)
Prefix(owl:=<http://www.w3.org/2002/07/owl#>)

Ontology(<http://www.example.com/ontology1>

   Import(<http://www.example.com/ontology2>)
    Annotation(rdfs:label "An example ontology")

   SubClassOf(ex:Child owl:Thing)

)
```

Annotation Properties

- Various annotation properties can be defined by users (e.g., an integer ID in the Foundational Model of Anatomy ontology; see http://sig.biostr.washington.edu/projects/fm/AboutFM.html).

- To help users in their modeling, OWL 2 also offers the constructs:
  - `SubAnnotationPropertyOf(AP1 AP2)` states that the annotation property `AP1` is a subproperty of the annotation property `AP2`.
  - `AnnotationPropertyDomain(AP U)` states that the domain of the annotation property `AP` is the IRI `U`.
  - `AnnotationPropertyRange(AP U)` states that the range of the annotation property `AP` is the IRI `U`.

Metamodeling

- OWL 2 enables **metamodeling** by allowing the same IRI `I` to refer to more than one type of entity (e.g., an individual and a class). This is called "**punning**" in the literature.

- Example:

```
ClassAssertion(a:Father a:John)
ClassAssertion(a:SocialRole a:Father)
```

- In the above example, IRI `a:Father` is first used as a class and then as an individual.

- The direct model-theoretic semantics of OWL 2 accommodates this by understanding the class `a:Father` and the individual `a:Father` as **two different views on the same IRI**, i.e. they are interpreted semantically as if they were distinct.

Semantics

- There are two alternative ways of assigning meaning to ontologies in OWL 2:
  - The **direct model-theoretic semantics**. This provides a meaning for OWL 2 in a DL style by understanding OWL 2 constructs as constructs of the DL SROIQ (with the exception of datatypes and punning that are not included in SROIQ but still covered by this semantics). See http://www.w3.org/TR/owl2-direct-semantics/ .
  - The **RDF-based semantics.** This is an extension of the semantics of RDFS (D-entailment in particular) and is based on viewing OWL 2 ontologies as RDF graphs. For the exact relationship of the two semantics, see http://www.w3.org/TR/owl2-rdf-based-semantics/.

OWL 2 DL and OWL 2 Full

- Informally, the notion **"OWL 2 DL"** is used to refer to OWL 2 ontologies interpreted using the direct semantics, and the notion **"OWL 2 Full"** is used when considering the RDF-based semantics.

- Formally, there are certain **additional conditions** which must be met by an OWL 2 ontology to qualify as OWL 2 DL.

- See the OWL 2 Structural Specification and Functional-Style Syntax for details.

Examples of Additional Conditions in OWL 2 DL

- **Reserved vocabulary** (e.g., `owl:Thing`) should only be used for its intended purpose.
- Classes, datatypes and properties (object, datatype and annotation) **need to be declared**.
- **Strict typing**: the sets of IRIs used as object, data, and annotation properties in *O* are disjoint and that, similarly, the sets of IRIs used as classes and datatypes in *O* are disjoint as well.
- Some **global restrictions on axiom closure** from SROIQ to ensure decidability (e.g., only **simple properties** in certain kinds of axioms).

Axiom Closure of an Ontology

- The **import closure** of an ontology *O* is a set containing *O* and all the ontologies that *O* imports.

- The **axiom closure** of an ontology *O* is the smallest set that contains all the axioms from each ontology *O′* in the import closure of *O* with all anonymous individuals **standardized apart** (i.e., the anonymous individuals from different ontologies in the import closure of *O* are treated as being different).

Example (not OWL 2 DL!)

```
Prefix(ex:=<http://www.example.com/ontology1#>)
Prefix(owl:=<http://www.w3.org/2002/07/owl#>)

Ontology(<http://www.example.com/ontology1>

SubObjectPropertyOf(
ObjectPropertyChain(ex:hasFather ex:hasBrother)
                                        ex:hasUncle)

EquivalentClasses(ex:PersonWithThreeUncles
   ObjectExactCardinality(3 ex:hasUncle ex:Person))
)
```

Note: Property `hasUncle` is **not simple** (it is defined by a property chain) so it **cannot be used in cardinality restrictions**.

OWL 2 DL and OWL 2 Full (cont'd)

- We can think of the difference between **OWL 2 DL** and **OWL 2 Full** in two ways:
  - OWL 2 DL is a syntactically restricted version of OWL 2 Full. OWL 2 Full is undecidable while OWL 2 DL is not. There are several production quality reasoners that cover the entire OWL 2 DL language (e.g., Pellet and HermiT).
  - OWL 2 Full is an extension of RDFS. As such, the RDF-Based Semantics for OWL 2 Full follows the RDFS semantics and general syntactic philosophy (i.e., everything is a triple and the language is fully reflective).

OWL 2 Profiles

- In addition to OWL 2 DL and OWL 2 Full, OWL 2 specifies **three profiles: OWL 2 EL, OWL QL and OWL RL.**
- These profiles are designed to be **subsets of OWL 2 sufficient for a variety of applications.**
- **Computational considerations** are a major requirement of these profiles; they are all much easier to implement with robust scalability given existing technology.
- There are many subsets of OWL 2 that have good computational properties. The selected OWL 2 profiles were identified as having **substantial user communities** already.
- The OWL 2 Profiles document provides a clear template for specifying additional profiles.

OWL 2 EL

- The OWL 2 EL profile is a subset of OWL 2 that
  - is particularly suitable for applications employing ontologies that define **very large numbers of classes and/or properties,**
  - captures the expressive power used by many such ontologies, and
  - for which **ontology consistency, class expression subsumption, and instance checking can be decided in polynomial time**.

- Example: OWL 2 EL is sufficient to express the very large biomedical ontology SNOMED CT. The specialized reasoner ELK can classify all 400,000 classes of this ontology in 5 seconds using 4 cores (http://www.cs.ox.ac.uk/isg/tools/ELK/).

- The acronym EL comes from the fact that the profile is based on the DL family of languages EL. See the relevant paper
  - *Pushing the EL Envelope.* Franz Baader, Sebastian Brandt, and Carsten Lutz. In Proc. of the 19th Joint Int. Conf. on Artificial Intelligence (IJCAI 2005), 2005 . Available from http://lat.inf.tu-dresden.de/research/papers/2005/BaaderBrandtLutz-IJCAI-05.pdf

OWL 2 EL Specification

- Types of **class restrictions** allowed:
  - existential quantification to a class expression (`ObjectSomeValuesFrom`) or a data range (`DataSomeValuesFrom`)
  - existential quantification to an individual (`ObjectHasValue`) or a literal (`DataHasValue`)
  - self-restriction (`ObjectHasSelf`)
  - enumerations involving a single individual (`ObjectOneOf`) or a single literal (`DataOneOf`)
  - intersection of classes (`ObjectIntersectionOf`) and data ranges (`DataIntersectionOf`)

OWL 2 EL Specification (cont'd)

- Types of **axioms** allowed:
  - class inclusion (`SubClassOf`)
  - class equivalence (`EquivalentClasses`)
  - class disjointness (`DisjointClasses`)
  - object property inclusion (`SubObjectPropertyOf`) with or without property chains, and data property inclusion (`SubDataPropertyOf`)
  - property equivalence (`EquivalentObjectProperties` and `EquivalentDataProperties`)

OWL 2 EL Specification (cont'd)

- transitive object properties (`TransitiveObjectProperty`)
- reflexive object properties (`ReflexiveObjectProperty`)
- domain restrictions (`ObjectPropertyDomain` and `DataPropertyDomain`)
- range restrictions (`ObjectPropertyRange` and `DataPropertyRange`)
- assertions (`SameIndividual, DifferentIndividuals, ClassAssertion, ObjectPropertyAssertion, DataPropertyAssertion, NegativeObjectPropertyAssertion,` and `NegativeDataPropertyAssertion`)
- functional data properties (`FunctionalDataProperty`)
- keys (`HasKey`)

OWL 2 EL Specification (cont'd)

- Constructs **not supported**:
  - universal quantification to a class expression (`ObjectAllValuesFrom`) or a data range (`DataAllValuesFrom`)
  - cardinality restrictions (`ObjectMaxCardinality`, `ObjectMinCardinality`, `ObjectExactCardinality`, `DataMaxCardinality`, `DataMinCardinality`, and `DataExactCardinality`)
  - disjunction (`ObjectUnionOf`, `DisjointUnion`, and `DataUnionOf`)
  - class negation (`ObjectComplementOf`)
  - enumerations involving more than one individual (`ObjectOneOf` and `DataOneOf`)

OWL 2 EL Specification (cont'd)

- disjoint properties (`DisjointObjectProperties` and `DisjointDataProperties`)
- irreflexive object properties (`IrreflexiveObjectProperty`)
- inverse object properties (`InverseObjectProperties`)
- functional and inverse-functional object properties (`FunctionalObjectProperty` and `InverseFunctionalObjectProperty`)
- symmetric object properties (`SymmetricObjectProperty`)
- asymmetric object properties (`AsymmetricObjectProperty`)

OWL 2 QL

- The OWL 2 QL profile is a subset of OWL 2 that provides a useful language for writing ontologies that have computational properties similar to the ones that one finds in **relational databases**.
- In this profile sound and complete **query answering can be done with LOGSPACE computational complexity** with respect to the size of the data (assertions), while providing many of the main features necessary to express conceptual models such as UML class diagrams and ER diagrams.
- This profile contains the intersection of RDFS and OWL 2 DL.
- This profile is designed so that data (assertions) that is stored in a standard relational database system can be queried through an ontology via a simple rewriting mechanism, i.e., by rewriting the query into an SQL query that is then answered by the RDBMS system, without any changes to the data.
- OWL 2 QL is based on the **DL-Lite** family of description logics.
- See the OWL 2 Language Profiles document for more details.

OWL 2 RL

- The OWL 2 RL profile is aimed at **applications that require scalable reasoning without sacrificing too much expressive power.**
- It is designed to accommodate both OWL 2 applications that can trade the full expressivity of the language for efficiency, and RDF(S) applications that need some added expressivity from OWL 2.
- This is achieved by defining **a syntactic subset of OWL 2 which is amenable to implementation using rule-based technologies** and **presenting a partial axiomatization of the OWL 2 RDF-based semantics in the form of first-order implications that can be used as the basis for such an implementation.**
- The design of OWL 2 RL was inspired by **Description Logic Programs** and **pD*.**
- See the OWL 2 Language Profiles document for more details.

OWL Syntaxes (cont'd)

- **The Functional-Style syntax** (used so far in these slides).
- **The RDF/XML syntax**: this is just RDF/XML, with a particular translation for the OWL constructs. Here one can use other popular syntaxes for RDF, e.g., **Turtle syntax.**
- **The Manchester syntax:** this is a frame-based syntax that is designed to be easier for users to read.
- **The OWL XML syntax:** this is an XML syntax for OWL defined by an XML schema.

Example

- Jack is a person but not a parent.

Functional-Style Syntax

```
ClassAssertion(
    ObjectIntersectionOf(:Person
            ObjectComplementOf(:Parent))
    :Jack
)
```

RDF/XML Syntax

```
<rdf:Description rdf:about="Jack">
  <rdf:type>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="Person"/>
        <owl:Class>
          <owl:complementOf rdf:resource="Parent"/>
        </owl:Class>
     </owl:intersectionOf>
    </owl:Class>
  </rdf:type>
</rdf:Description>
```

Turtle Syntax

```
:Jack rdf:type [
  rdf:type owl:Class;
  owl:intersectionOf ( :Person
                       [ rdf:type owl:Class;
                         owl:complementOf  :Parent ]
                     )
] .
```

Manchester Syntax

```
Individual: Jack
Types: Person and not Parent
```

OWL/XML Syntax

```
<ClassAssertion>
  <ObjectIntersectionOf>
    <Class IRI="Person"/>
    <ObjectComplementOf>
      <Class IRI="Parent"/>
    </ObjectComplementOf>
   </ObjectIntersectionOf>
   <NamedIndividual IRI="Jack"/>
</ClassAssertion>
```

Readings

- The document http://www.w3.org/TR/2009/REC-owl2-overview-20091027/ gives an overview of the OWL 2 specification of the W3C OWL Working Group. In the documents referenced there, you will find all the information that you may need.

- You should read at least the Primer (http://www.w3.org/TR/owl2-primer/) and Structural Specification and Functional Style Syntax (http://www.w3.org/TR/owl2-syntax/) .

- The following survey paper introduces OWL 2, explains its relationship with DL *SROIQ,* and discusses various OWL tools and applications:
  - Ian Horrocks and Peter F. Patel-Schneider. KR and Reasoning on the Semantic Web: OWL. In Handbook of Semantic Web Technologies, chapter 9. Springer, 2010. Available from http://www.cs.ox.ac.uk/people/ian.horrocks/Publications/download/2010/HoPa10a.pdf

Readings (cont'd)

- The DL *SROIQ* on which OWL 2 is based is fully described in the paper
  - The Even More Irresistible SROIQ. Ian Horrocks, Oliver Kutz, and Uli Sattler. In Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2006). AAAI Press, 2006. Available from http://www.cs.manchester.ac.uk/~sattler/publications/sroiq-TR.pdf.