

## Gestione di dati RDF con Eclipse RDF4J

**Manuel Fiorelli**

***<fiorelli@info.uniroma2.it>***

# Eclipse RDF4J (1/3)

**Eclipse RDF4J** (precedentemente noto come OpenRDF Sesame) è un framework Java per l'*elaborazione di dati RDF*:

- **rappresentare** *termini e statement* RDF all'interno di programmi Java
- **leggere/scrivere** dati RDF data/in sintassi concrete
- **accedere a database RDF** (*triplestore*) sia integrati nell'applicazione sia remoti
- **storage** di dati RDF
- **query** su dati RDF
- **inferenza** su dati RDF
- **indicizzazione** di dati RDF

I **potenziali utenti** di Eclipse RDF4J includono:

- *Sviluppatori di applicazioni per il Web Semantico* che hanno bisogno di memorizzazione ed elaborare dati RDF
- *Implementatori di triplestore* che possono sfruttare l'architettura stratificata di RDF4J ed i suoi componenti per leggere/scrivere le comuni sintassi RDF, effettuare il parsing e la valutazione di query, etc.

Eclipse RDF4J include **due implementazioni di triplestore**:

- *Memory store*: dataset piccoli, che si adattano alla RAM disponibile; persistenza su disco opzionale
- *Native store*: dataset di medie dimensioni (100 milioni triple); memorizzazione ed elaborazione basata su disco

In aggiunta, è possibile usare triplestore (commerciali) di terze parti che si conformano in misura diversa all'architettura di RDF4J.

# Setup di Eclipse RDF4J

Gli **artefatti** (*artifact*) di Eclipse RDF4J sono *pubblicati su Maven Central* rendendone immediato l'uso in **progetti basati su Apache Maven**.

- Aggiungere le dipendenze agli artefatti richiesti
- Oppure, aggiungere soltanto una dipendenza al runtime di RDF4J (che a sua volta importa gli altri moduli)

```
<dependency>
```

```
<groupId>org.eclipse.rdf4j</groupId>
```

```
<artifactId>rdf4j-runtime</artifactId>
```

```
<version>2.4.0</version>
```

```
</dependency>
```

Proprietà che può essere utile aggiungere al POM

```
<properties>
```

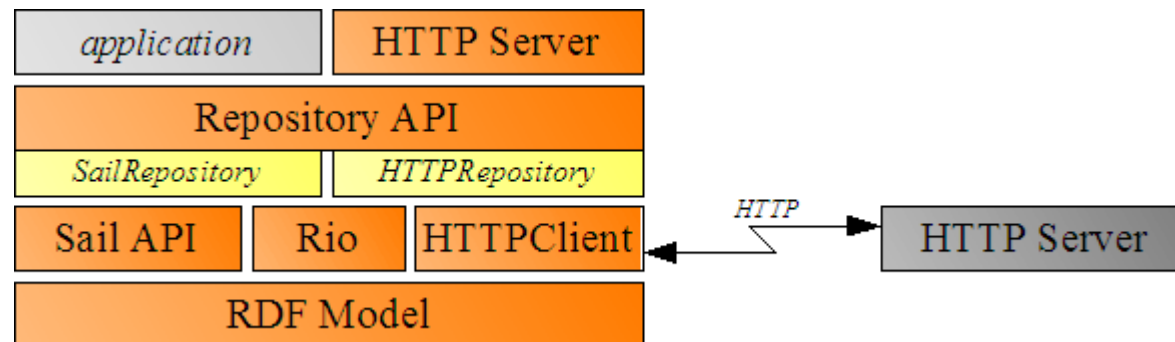
```
<maven.compiler.source>1.8</maven.compiler.source>
```

```
<maven.compiler.target>1.8</maven.compiler.target>
```

```
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
```

```
</properties>
```

# Architettura di RDF4J



# Nota sulle eccezioni

In OpenRDF Sesame la maggior parte delle eccezioni erano di tipo checked:

- Se non vengono catturate nel corpo di un metodo, occorre inserirle nella clausola throws del metodo stesso

In Eclipse RDF4J la maggior parte delle eccezioni (specifiche del framework) sono di tipo unchecked:

- Non occorre elencarle nella clausola throws nel caso non vengano gestite internamente.

Questo cambiamento è stato motivato – tra l'altro – dalla volontà di migliorare la compatibilità con le lambda expression di Java 8.

# PACKAGE MODEL

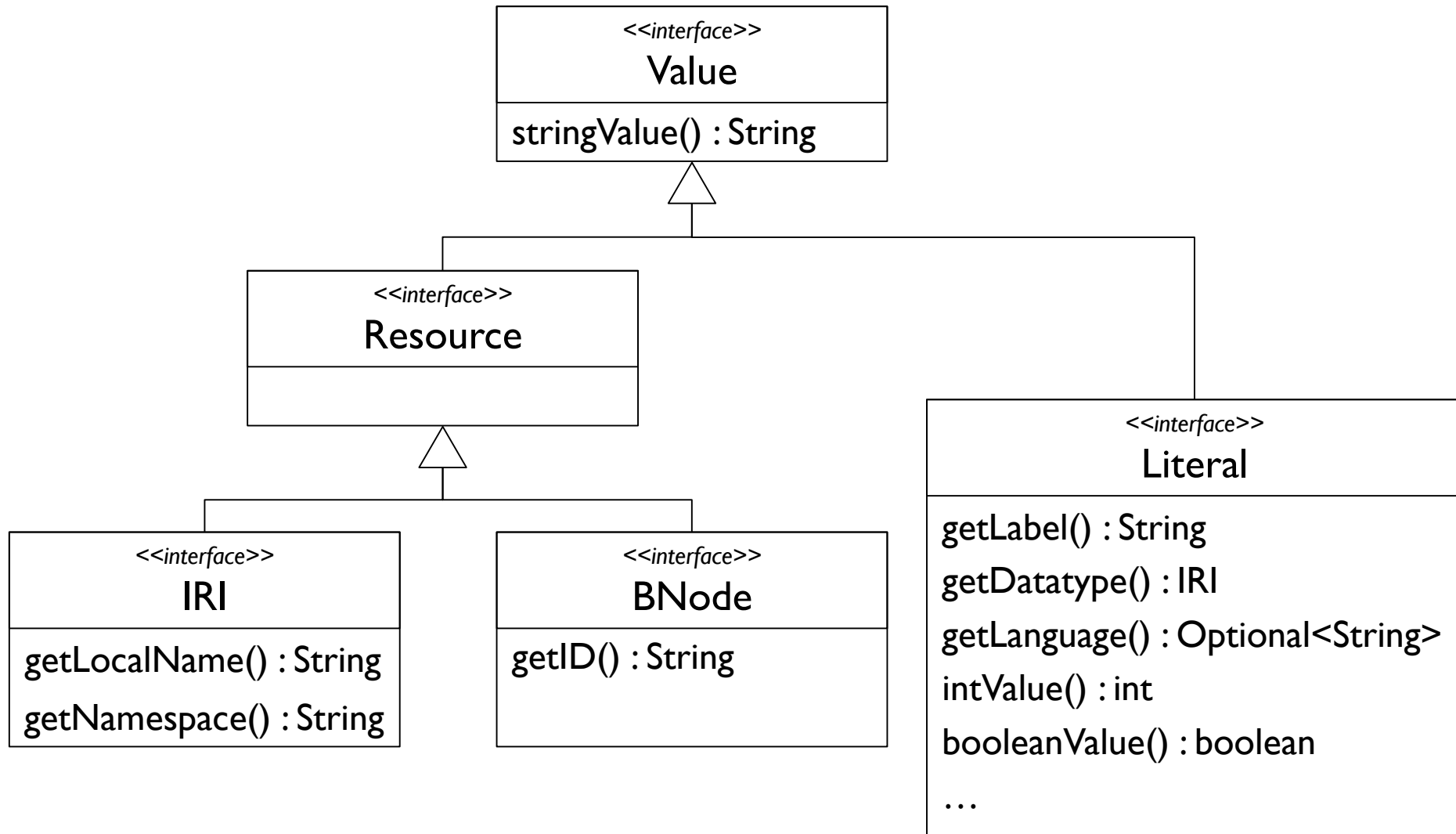


# Il package Model

Il **package Model** fornisce interfacce per *rappresentare termini* RDF, *statement* e collezioni di ciò – chiamate *modelli*.

Ci sono *implementazioni di modelli* (*LinkedHashModel* e *TreeModel*) che supportano la manipolazione di statement RDF in memoria.

# II Package Model – Termini RDF (1/2)



# Il Package Model – Termini RDF (2/2)

I *Value* di RDF4J sovrascrivono (*override*) il metodo `Object.equal(Object o)` (come pure `Object.hashCode()`) per implementare l'*uguaglianza tra value* come segue:

- Due *IRI* sono uguali, se essi hanno lo stesso *stringValue*
- Due *Literal* sono uguali, se hanno la stessa *label*, lo stesso *datatype*, e (se presente) lo stesso *language*
- Due *BNode* sono uguali, se hanno lo stesso *identifier*

# Il Package Model – Statement

Uno **statement** è composto da un *soggetto* (*subject*), un *predicato* (*predicate*) e uno *oggetto* (*object*).

Opzionalmente, un quarto componente – chiamato *contesto* (*context*) – può essere usato per raggruppare un insieme di statement

I principali usi del contesto includono:

- Tracciare la provenienza (*provenance*) degli statement (es. usando il contesto per indicare l'origine degli statement)
- Valutare una query su sottoinsieme degli statement (in particolare su determinati contesti)
- Alternativa alla reificazione RDF standard per fornire metadati su uno o più statement

<<interface>> Statement
getSubject() : Resource
getPredicate() : IRI
getObject() : Value
getContext() : Resource

Due statement sono uguali se e solo se tutte le componenti corrispondenti sono uguali

# Il Package Model – ValueFactory

I tipi descritti finora sono interfacce: in Java non possono essere istanziati direttamente.

Come otteniamo un oggetto di tipo *IRI*, *BNode*,..., *Statement*?

Eclipse RDF4J usa il *design pattern della Factory Astratta*

- L'interfaccia *ValueFactory* definisce un'API per istanziare i vari tipi definiti nel package Model
- Una *factory concreta* può essere ottenuta da una connessione ad un repository (maggiori informazioni a seguire) o semplicemente come *SimpleValueFactory.getInstance()*

# II Package Model – ValueFactory – IRI (1/2)

ValueFactory vf = ... ;

IRI per esteso

IRI socrates = vf.createIRI("http://example.org/Socrates");

IRI aristotle = vf.createIRI("http://example.org/Aristotle");

ValueFactory vf = ... ;

IRI = namespace + local name

String ns = "http://example.org/"

IRI socrates = vf.createIRI(ns + "Socrates");

IRI aristotle = vf.createIRI(ns + "Aristotle");

ValueFactory vf = ... ;

Delega la concatenazione del namespace a  
RDF4J

String ns = "http://example.org/"

IRI socrates = vf.createIRI(ns, "Socrates");

IRI aristotle = vf.createIRI(ns, "Aristotle");

## II Package Model – ValueFactory – IRI (2/2)

```
ValueFactory vf = ... ;
```

```
IRI type = vf.createIRI(“http://www.w3.org/1999/02/22-rdf-syntax-ns#type”);
```

Constanti per vocabolari comuni

```
import org.eclipse.rdf4j.model.vocabulary.RDF
```

```
...
```

```
IRI type = RDF.TYPE;
```

# II Package Model – ValueFactory – BNode

```
ValueFactory vf = ... ;  
BNode bnode = vf.createBNode();
```

```
ValueFactory vf = ... ;  
String nodeID = "abc...:";  
BNode bnode = vf.createBNode(nodeID);
```

Nell'ambito dell'API di RDF4J l'identità di un bnode è determinata da un *identificatore di nodo* (**locale**).

Tuttavia, questo identificatore non esiste in ambito globale: durante l'*importazione dei dati* (es. lettura di un file), i **bnode** sono rinominati **consistentemente** (per evitare collisioni con identificatori già in uso localmente)



# II Package Model – ValueFactory – Literal

```
ValueFactory vf = ... ;
```

Language-tagged literal

```
Literal literal = vf.createLiteral("dog", "en")
```

```
ValueFactory vf = ... ;
```

```
Literal literal = vf.createLiteral("2018-09-26T16:24:28+00:00",  
XMLSchema.DATETIME);
```

Datatype

Label

```
ValueFactory vf = ... ;
```

```
GregorianCalendar c = new GregorianCalendar();
```

```
c.setTime(yourDate);
```

```
XMLGregorianCalendar date =
```

```
DatatypeFactory.newInstance().newXMLGregorianCalendar(c);
```

```
Literal literal = vf.createLiteral(date);
```

Creazione di un literal da un oggetto Java

## II Package Model – ValueFactory – Statement

```
ValueFactory vf = ... ;  
String ns = "http://example.org/";  
IRI socrates = vf.createIRI(ns, "Socrates");  
IRI mortal = vf.createIRI(ns, "Mortal");  
Statement stmt = vf.createStatement(socrates, RDF.TYPE, mortal);
```

# Il Package Model – i Model (1/8)

Un *Model* è un insieme di *Statement*.

```
public interface Model extends Graph, Set<Statement>,
    Serializable, NamespaceAware {
    ...
}
```

Le implementazioni elencate qui sotto possono essere usate per gestire (piccole) collezioni di statement in memoria:

- *LinkedHashModel*: usa hash-table
- *TreeModel*: usa alberi Red-Black, ordinando i dati secondo l'ordine lessicale dei termini RDF

## II Package Model – i Model (2/8)

Creazione di una collection

```
LinkedHashSet<Statement> set = new LinkedHashSet<>();
```

Creazione di un model

```
Model model = new LinkedHashModel ();
```

## II Package Model – i Model (3/8)

Aggiunta di uno statement ad una collection

```
set.add(vf.createStatement(  
    vf.createIRI(ns, "Socrates"),  
    RDF.TYPE,  
    vf.createIRI(ns, "Mortal")  
));
```

Aggiunta di uno statement ad un model

```
model.add(vf.createIRI(ns, "Socrates"), RDF.TYPE, vf.createIRI(ns, "Mortal"))
```

Un quarto parametro è un *vararg* che può essere usato per indicare i contesti in cui inserire lo statement. Se non viene specificato alcun contesto, lo statement non ha contesto associato.

## II Package Model – i Model (4/8)

```
Statement stmt = vf.createStatement(
    vf.createIRI(ns, "Socrates"),
    RDF.TYPE,
    vf.createIRI(ns, "Mortal")
);
set.contains(stmt)
```

Testare presenza di uno statement in una collection

```
model.contains(vf.createIRI(ns, "Socrates"),
    RDF.TYPE, vf.createIRI(ns, "Mortal"))
```

Testare presenza di uno statement in un model

*null* gioca il ruolo di wildcard; in aggiunta dai Javadoc:

model.contains(sI, null, null) è true se qualche statement ha soggetto sI,  
 model.contains(null, null, null, cI) è true se qualche statement ha contesto cI,  
 model.contains(null, null, null, (Resource)null) è true se qualche statement non ha un contesto associato

model.contains(null, null, null, cI, c2, c3) è true se qualche statement ha contesto cI, c2 o c3 .

# Il Package Model – i Model (5/8)

## Dai Javadoc:

`model.remove(sI, null, null)` rimuove ogni statement che ha soggetto `sI`,

`model.remove(null, null, null, cI)` rimuove ogni statement che ha contesto `cI`,

`model.remove(null, null, null, (Resource)null)` rimuove ogni statement che non ha contesto associato,

`model.remove(null, null, null, cI, c2, c3)` rimuove ogni statement che ha contesto `cI`, `c2` o `c3`.

## Il Package Model – i Model (6/8)

È possibile ottenere una **vista** sugli statement con un soggetto, *predicato* e oggetto (e opzionalmente, *contesto*) specificati.

```
Model filteredModel = model.filter(subject, predicate,  
object, contexts...);
```

Modifiche al modello di partenza si rifletteranno sulla vista, e viceversa.



## Il Package Model – i Model (7/8)

Possiamo chiedere ad un model una **vista** sui soggetti, predicati e oggetti che contiene.

```
Set<Resource> subjects = model.subjects();
```

```
Set<IRI> predicates = model.predicates();
```

```
Set<Value> objects = model.objects();
```

Usando la classe di utilità *Models* possiamo richiedere componenti di un certo tipo, es. *Models.objectLiterals(model)* restituisce tutti e soli gli oggetti di tipo literal.

## Il Package Model – i Model (8/8)

La classe di utilità *Models* supporta anche un approccio basato sulle proprietà anziché sugli statement.

```
Optional<String> aStringValue = Models.getPropertyString(model, subject, property);
```

Anziché usare le viste e l'utility per filtrare gli oggetti:

```
Optional<String> aStringValue = Models.objectString(model.filter(subject, property, null));
```

Similmente, possiamo settare il valore di una proprietà e cancellare tutti i suoi valori precedenti (nei contesti indicati, oppure in tutto il modello se non ne viene indicato nessuno).

```
Models.setProperty (model, subject, property, value);
```

RDF Input/Output

**RIO**

**RIO** (RDF Input/Output) il modulo di occupazione dell'*input/output di dati RDF*.

- *RDFParser*: legge un documento RDF come oggetti conformi alle interfacce definite dal package model
- *RDFWriter*: scrive statement RDF (rappresentati come oggetti conformi alle interfacce definite dal package model) come documenti RDF conformi ad una sintassi concreta

# RDFParser (1/5)

```
RDFParser p = RDFParserRegistry.getInstance()  
    .get(RDFFormat.TURTLE)  
    .orElseThrow(Rio.unsupportedFormat(RDFFormat.TURTLE))  
    .getParser();
```

Un registro contenente gli  
RDFParser trovati sul classpath

La factory del parser per il  
format TURTLE, se disponibile:  
*Optional<RDFParserFactory>*

Lancia un'eccezione se il parser  
non è stato trovato

Istanza il parser

## Scorciatoia

```
RDFParser p = Rio.createParser(rdfFormat [, valueFactory]);
```

# RDFParser (2/5)

OPZIONALI

p.setValueFactory(...)

Setta la *ValueFactory* da usare per creare istanze delle interfacce del modello

p.setParseErrorListener(...)

Setta il listener da notificare di ogni errore durante il parsing

p.setParseLocationListener(...)

Setta il listener da notificare del progresso del parser

p.setRDFHandler(...)

Setta l'*RDFHandler* cui saranno forniti gli statement letti

```
ParserConfig config = new ParserConfig();
```

```
config.set(setting, value);
```

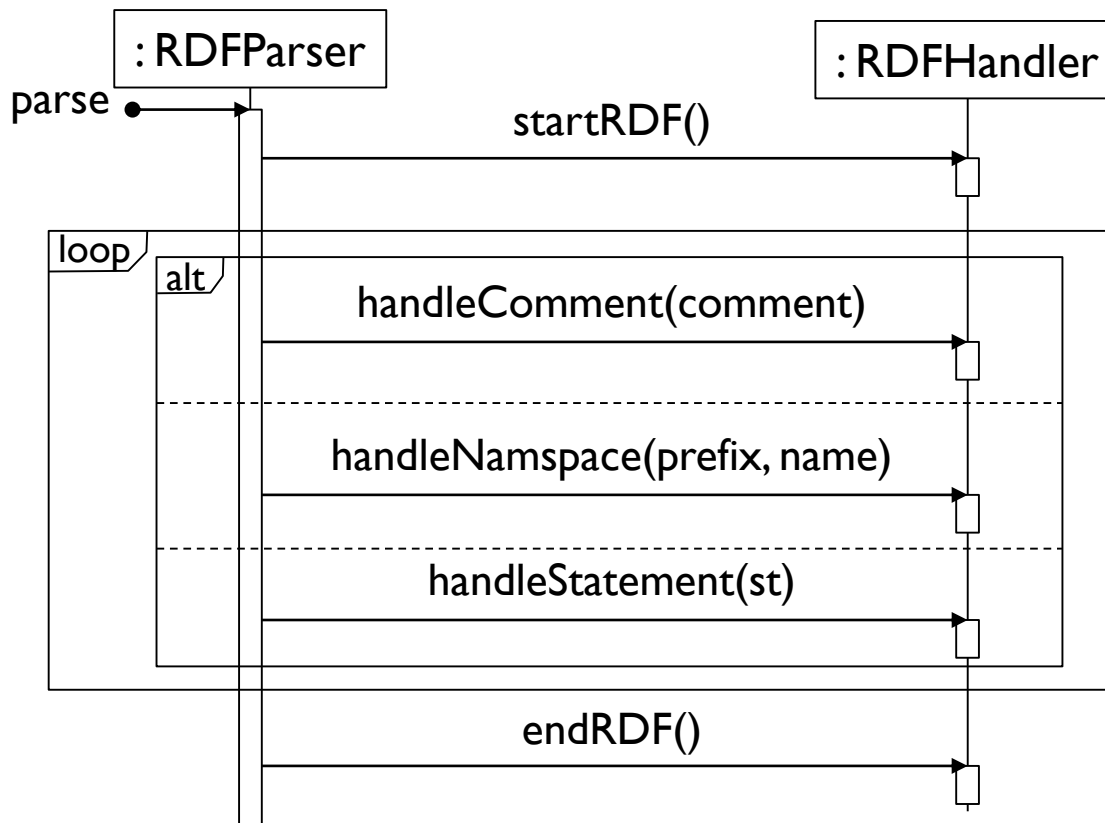
```
p.setParserConfig(config)
```

*RDFParser.getSupportedSettings()* restituisce i setting riconosciuti dal parser

Un *RioSetting<T>* che rappresenta un parametro di configurazione

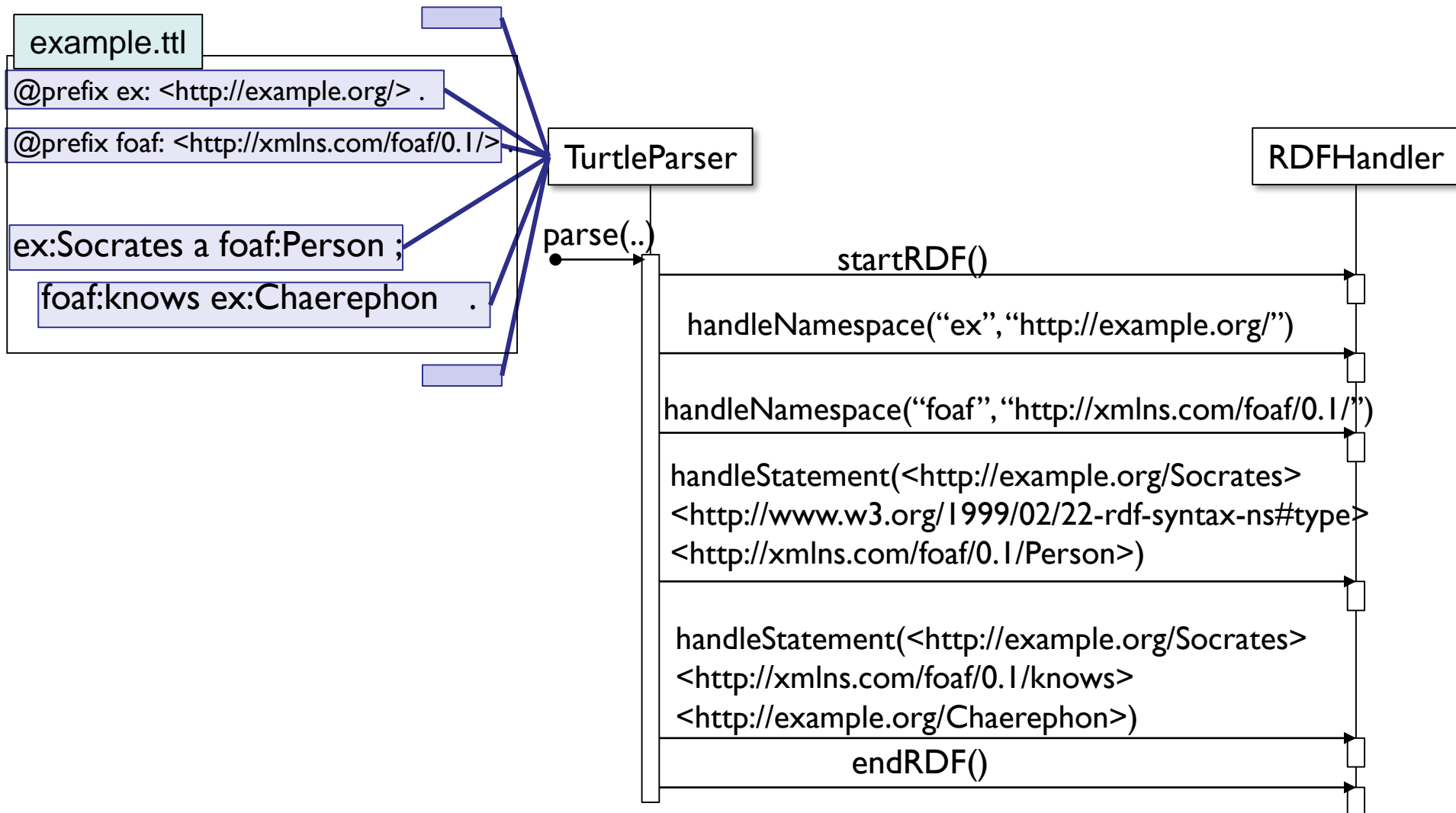
# RDFParser (3/5)

```
p.parse(inputStreamOrReader, baseURI);
```



**Approccio push al parsing** alla maniera dall'API SAX per il parsing XML

# RDFParser (4/5)





Uno **StatementCollector** è un *RDFHandler* che può essere usato per inserire gli statement letti in una collezione (oppure un model).

## Scorciatoia

```
Model model = Rio.parse(inputStreamOrReader,baseURI,rdfFormat, [settings,  
valueFactory, errorListener], contexts...)
```

# RDFWriter (1/2)

```
RDFWriter w = RDFWriterRegistry.getInstance()
    .get(RDFFormat.TURTLE)
    .orElseThrow(Rio.unsupportedFormat(RDFFormat.TURTLE))
    .getWriter(outputStreamOrWriter [, baseURI]);
```

Un registro contenente gli RDFWriter trovati nel classpath

La factory del writer per il format TURTLE, se disponibile:  
*Optional<RDFWriterFactory>*

Lancia un'eccezione se il writer non è stato trovato

Istanza il writer

## Scorciatoia

```
RDFWriter w = Rio.createWriter(rdfFormat, outputStreamOrWriter [, base URI]);
```

# RDFWriter (2/2)

```
WriterConfig config = new WriterConfig();
config.set(setting, value);
p.setWriterConfig(config)
```

RDFWriter.getSupportedSettings()  
restituisce i setting riconosciuti dal  
writer

Un *RioSetting*<T> che  
rappresenta un parametro  
di configurazione

```
Rio.write(iterable, w);
```

## Shorthand<sup>2</sup>

```
Rio.write(iterable, outputStreamOrWriter [, base URI], format, [writerConfig]);
```

Non c'è mai bisogno di avere tutti gli statement caricati in memoria

- I *reader* inviano all'*handler* uno statement alla volta
- Similmente, un *writer* riceve uno statement alla volta

Accedere a triplestore

# REPOSITORY

# Repository (1/2)

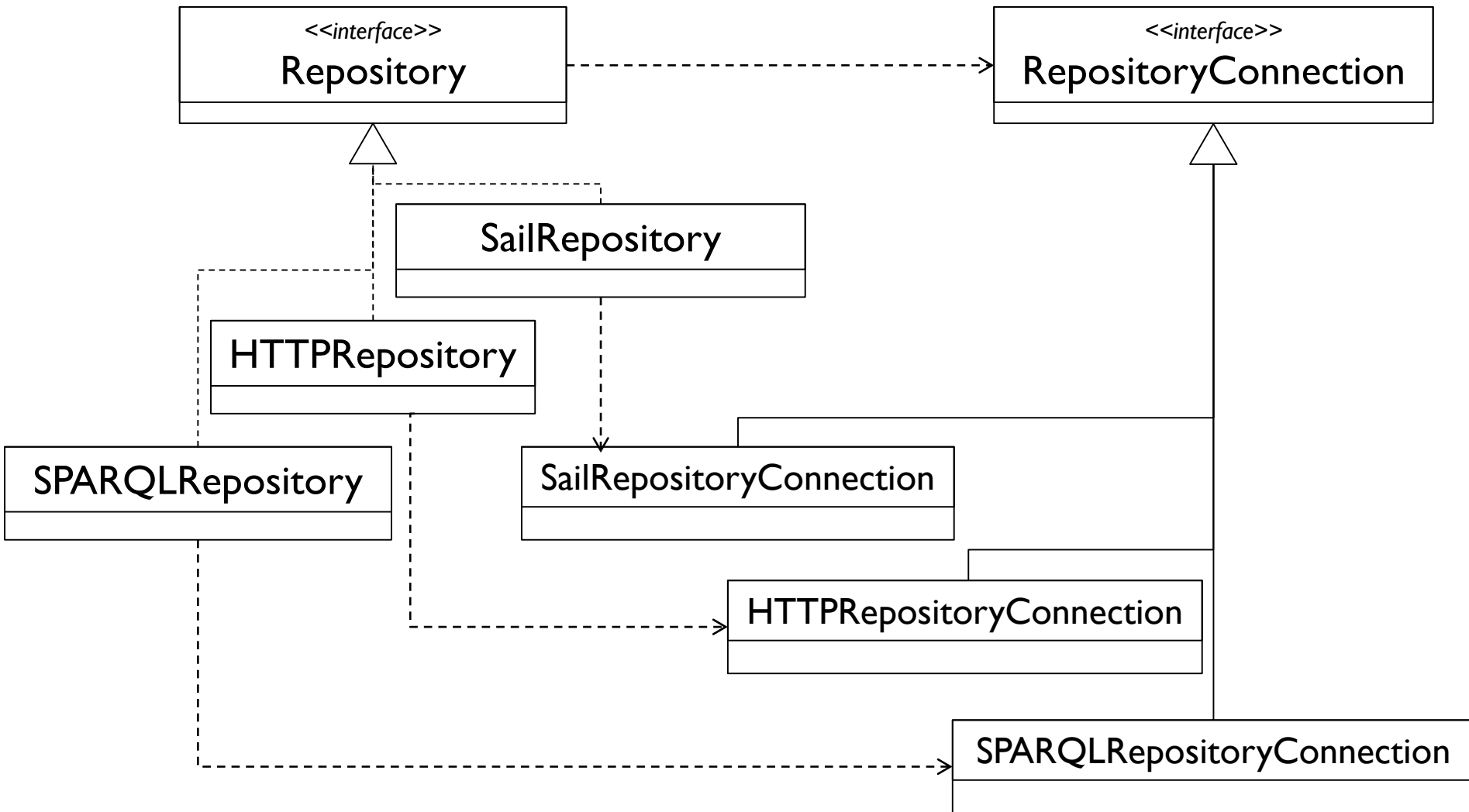
Un **Repository** è il punto di accesso ad un *database RDF* (*triplestore*).

Un utente interagisce con un *Repository* attraverso una **RepositoryConnection**, ottenuta dal repository stesso.

Rispetto ai modelli, i *Repository* generalmente supportano:

- Connessione concorrente di molteplici utenti
- Transazioni (commit o rollback di gruppi di operazioni, diversi livelli di isolamento tra transazioni concorrenti)
- Supporto per linguaggi di interrogazione e/o modifica (spesso SPARQL)

# Repository (2/2)



# SPARQLRepository

Usato per **accedere ad un endpoint SPARQL** attraverso l'interfaccia *Repository*.  
Presenta, tuttavia, delle limitazioni e/o non conformità dovute alle limitazioni del sottostante linguaggio/protocollo SPARQL (es. gestione delle transazioni e bnode come input). È preferibile usarlo per singole operazioni SPARQL.

```
Repository rep = new SPARQLRepository("http://dbpedia.org/sparql");
rep.initialize();
try {
    // use the repository
}
finally {
    rep.shutdown();
}
```

Utilizzare  
l'overload con due  
parametri per  
indicare l'endpoint  
di modifica



# HTTPRepository

Usato per ***accedere ad un repository remoto*** che è stato esposto attraverso l'API REST di RDF4J (maggiori informazioni in seguito).

```
Repository rep = new HTTPRepository("http://localhost:7200/");
r.setUsernameAndPassword(username, password); // se necessario
rep.initialize();
try {
    // use the repository
}
finally {
    rep.shutdown();
}
```

# SailRepository (1/2)

```
Sail sail = new MemoryStore();  
Repository rep = new SailRepository(sail);  
rep.initialize();  
try {  
    // use the repository  
}  
finally {  
    rep.shutdown();  
}
```

Un Sail (Storage and Inference Layer) implementa le funzionalità di storage, query e inferenza.

L'interfaccia Sail è stata introdotta per disaccoppiare le implementazioni di triplestore dai moduli funzionali del framework (parsers, query engines, end-user API access, etc).

La configurazione sulla sinistra usa un Sail operante esclusivamente in memoria, ed è pertanto detto non persistente

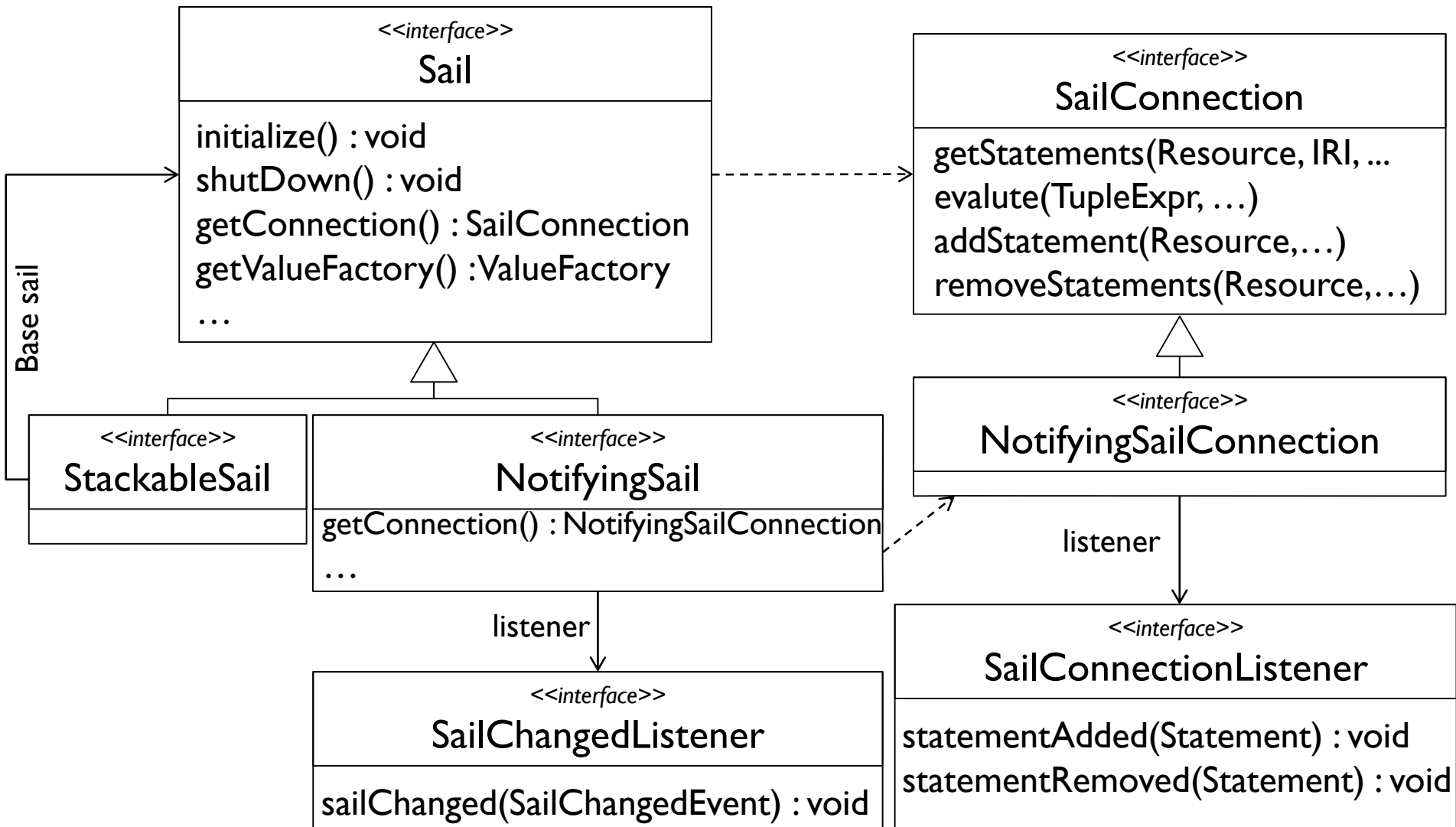
# SailRepository (2/2)

```
MemoryStore sail = new MemoryStore();
sail.setDataDir(...)
sail.setPersist(true);
sail.setSyncDelay(0);
Repository rep = new SailRepository(sail);
rep.initialize();
try {
    // use the repository
}
finally {
    rep.shutdown();
}
```

Il contenuto del repository è scritto (*setPersist(true)*) in un file, chiamato *memorystore.data*.

- *syncDelay = 0* il file è scritto in maniera sincrona quando viene fatto il commit, massimizzando la durabilità dei dati; tuttavia, una transazione in scrittura deve attendere una scrittura su file (più lenta dell'accesso alla memoria). In aggiunta, transazioni in scrittura ravvicinate, determinano scritture su file ravvicinate
- *syncDelay > 0* rende il riversamento dei dati nel file un task asincrono eseguito dopo un certo lasso di tempo. Una transazione in scrittura non dovrà più attendere la scrittura del file
- *syncDelay < 0* i dati sono scritti nel file solo quando il sail viene spento

# Sail – Architettura (1/2)



## Sail – Architettura (2/2)

Uno **stackable sail** opera al di sopra di un altro sail (*base*, o *delegate*):

- Può restituire un *wrapper* intorno ad una connessione al base sail (utile per eseguire logica speciale al commit o rollback, o per intercettare ogni lettura/scrittura ed esecuzione di query)
- Può sottoscrivere alla ricezione di eventi generati dal base (*notifying*) sail
  - Questi eventi dicono se qualche statement è stato aggiunto o rimosso
- In modo simile, il wrapper intorno alla connessione può sottoscrivere alla ricezione di eventi generati dalla connessione (*notifying*) di base
  - Questi eventi dicono quali statement sono stati effettivamente aggiunti/rimossi durante una transazione

# ForwardChainingRDFSInferencer

```
File dataDir = ...;
Repository rep = new SailRepository(
    new ForwardChainingRDFSInferencer(
        new MemoryStore(dataDir)
    )
);
```

Il *ForwardChainingRDFSInferencer* è uno stackable sail che implementa il reasoning RDFS attraverso il *forward chaining*:

- Aggiunge triple assiomatiche al repository (es. *rdfs:subClassOf rdfs:range rdfs:Class*)
- Quando si fa il commit di una transazione, calcola le triple implicate logicamente applicando iterativamente le regole che specificano la semantica RDFS.

Siccome la semantica di RDFS (e OWL) è monotona, l'**aggiunta di una tripla** non può invalidare inferenze precedenti: il reasoner tenta semplicemente di applicare nuovamente le regole che specificano la semantica RDFS.

Diversamente, la **cancellazione di una tripla** può invalidare inferenze precedenti: il reasoner cancella tutte le triple inferite, ed avvia il processo di reasoning da capo

# Note sulle Configurazioni dei Repository

Eclipse RDF4J **non distingue** la *creazione di un repository* dall'*apertura di un repository creato in precedenza*.

Internamente, un **repository persistente** guarda la propria directory dati, e *carica i dati* che eventualmente sono stati salvati in precedenza.

L'oggetto **Repository** dovrebbe essere *sempre configurato nello stesso modo*, a meno che non si sappia che un certo cambiamento della configurazione è sicuro (es. il native store crea eventuali indici non esistenti all'avvio)

# RepositoryConnection - Acquisizione

```
Repository rep = ... ;
RepositoryConnection conn = rep.getConnection();
try {
    // use the connection
}
finally {
    conn.close();
}
```

try-finally

```
Repository rep = ... ;
try(RepositoryConnection conn = rep.getConnection()) {
    // use the connection
}
```

try-with-resources



# RepositoryConnection – add (1/2)

```
ValueFactory vf = conn.getValueFactory();
String ns = "http://example.org#";
IRI Socrates = vf.createIRI(ns, "Socrates");
IRI Chaerephon = vf.createIRI(ns, "Chaerephon");
IRI g1 = vf.createIRI("http://example.org/g1");
IRI g2 = vf.createIRI("http://example.org/g2");

conn.add(Socrates, RDF.TYPE, FOAF.PERSON, g1);
conn.add(Socrates, RDFS.LABEL, vf.createLiteral("Socrates", "en"), g1);
conn.add(Chaerephon, RDF.TYPE, FOAF.PERSON, g2);
conn.add(Chaerephon, RDFS.LABEL, vf.createLiteral("Chaerephon", "en"), g2);
conn.add(Socrates, FOAF.KNOWS, Chaerephon);
conn.add(Chaerephon, FOAF.KNOWS, Socrates);
```

# RepositoryConnection – add (2/2)

Ci sono diversi overaload per il metodo *RepositoryConnection.add(..)* che accettano input differenti:

- Iterable<? Extends Statement>
- Iteration<? Extends Statement>
- File
- InputStream
- Reader
- Statement
- Statement esploso nelle sue componenti (soggetto, predicato, oggetto)

Il parametro context può essere usato per indicare i contesti in cui aggiungere gli statement (sovrascrivendo ogni eventuali contesto indicato nell'input).

# RepositoryConnection – export (1/3)

```
RDFHandler nquadOutputter = Rio.createWriter(RDFFormar.TRIG, System.out);
conn.export (nquadOutputter, gl);
```

```
<http://example.org/gl> {
  <http://example.org#Socrates> a <http://xmlns.com/foaf/0.1/Person>;
  <http://www.w3.org/2000/01/rdf-schema#label> "Socrates"@en .
}
```

Il metodo `RepositoryConnection.export(..)` esporta unicamente gli statement espliciti

# RepositoryConnection – export (2/3)

```
conn.setNamespace("ex", ns);
conn.setNamespace(RDFS.PREFIX, RDFS.NAMESPACE);
conn.setNamespace(FOAF.PREFIX, FOAF.NAMESPACE);
RDFHandler nquadOutputter = Rio.createWriter(RDFFormar.TRIG, System.out);
conn.export (nquadOutputter, gI);
```

```
@prefix ex: <http://example.org#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
<http://example.org/gI> {
  ex:Socrates a foaf:Person;
  rdfs:label "Socrates"@en .
}
```

# RepositoryConnection (3/3)

```
RDFHandler nquadOutputter = Rio.createWriter(RDFFormat.TRIG, System.out);
conn.export (nquadOutputter);
```

@prefix

@prefix

@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<http://example.org/g1> {

ex:Socrates a foaf:Person; rdfs:label "Socrates"@en .

}

<http://example.org/g2> {

ex:Chaerephon a foaf:Person; rdfs:label "Chaerephon"@en .

}

{

ex:Socrates foaf:knows ex:Chaerephon .

ex:Chaerephon foaf:knows ex:Socrates .

}

# RepositoryConnection - getStatements (1/2)

```
try(RepositoryResult<Statement> result = conn.getStatements(Socrates, null, null, gI)) {  
    while (result.hasNext()) {  
        System.out.println(result.next());  
    }  
}
```

```
(http://example.org#Socrates, http://www.w3.org/1999/02/22-rdf-syntax-ns#type,  
http://xmlns.com/foaf/0.1/Person) [http://example.org/gI]  
(http://example.org#Socrates, http://www.w3.org/2000/01/rdf-schema#label,  
"Socrates"@en) [http://example.org/gI]
```

Il metodo `RepositoryConnection.getStatements(..)` include gli statement inferiti (per impostazione predefinita, a meno che non si usi l'overload con il parametro booleano `includeInferred` a `false`)

# RepositoryConnection - getStatements (2/2)

```
try(RepositoryResult<Statement> result = conn.getStatements(Socrates, null, null)) {  
    QueryResults.stream(result).forEach(System.out::println);  
}
```

```
(http://example.org#Socrates, http://www.w3.org/1999/02/22-rdf-syntax-ns#type,  
http://xmlns.com/foaf/0.1/Person) [http://example.org/gI]  
(http://example.org#Socrates, http://www.w3.org/2000/01/rdf-schema#label,  
"Socrates"@en) [http://example.org/gI]  
(http://example.org#Socrates, http://xmlns.com/foaf/0.1/knows,  
http://example.org#Chaerephon) [null]
```

# RepositoryConnection - remove

Ci sono diversi overload dell'operazione *RepositoryConnection.remove(..)* che accettano diversi input:

- `Iterable<? Extends Statement>`
- `Iteration<? Extends Statement>`
- `Statement`
- `Statement` esploso nelle sue componenti (soggetto, predicato, oggetto)

Il parametro *context* può essere usato per specificare da quali named graph rimuovere le triple input (sovrascrivendo ogni contesto indicato nell'input). Se nessun contesto è specificato come argomento del metodo né da parte degli `statement` di input, il metodo opera sull'intero repository.



# RepositoryConnection – tuple query

```

TupleQuery query = conn.prepareTupleQuery(
    "PREFIX <foaf: http://xmlns.com/foaf/0.1/>\n" +
    "SELECT ?acquaintance WHERE {\n" +
    "  ?subject foaf:knows ?acquaintance . \n" +
    "}\n"
);
query.setBinding("subject", Socrates);
// query.setIncludeInferred(true/false); // default is true
try(TupleQueryResult queryResult = query.evaluate()) {
    while(result.hasNext()) {
        BindingSet bindingSet = result.next();
        Value acquaintance = bindingSet.getValue("acquaintance");
        System.out.println(acquaintance);
    }
}

```

# RepositoryConnection – graph query

```

TupleQuery query = conn.prepareGraphQuery(
    "PREFIX <foaf: http://xmlns.com/foaf/0.1/>\n" +
    "DESCRIBE ?acquaintance WHERE {\n" +
    "  ?subject foaf:knows ?acquaintance . \n" +
    "}\n"
);
query.setBinding("subject", Socrates);
// query.setIncludeInferred(true/false); // default is true
try(GraphQueryResult queryResult = query.evaluate()) {
    while(result.hasNext()) {
        Statement statement = result.next();
        System.out.println(statement);
    }
}

```

Model statements =  
 QueryResults.asModel(query.evaluate())

# RepositoryConnection – update

```
Update update = conn.prepareUpdate(
    "PREFIX <foaf: http://xmlns.com/foaf/0.1/>\n" +
    "DELETE { ?person foaf:givenName 'Bill' }\n"+
    "INSERT { ?person foaf:givenName 'William' }\n" +
    "WHERE {\n" +
    "  ?person foaf:givenName 'Bill'\n" +
    "}\n");
// update.setIncludeInferred(true/false); // default is true
update.execute();
```

# RepositoryConnection – transazioni (1/6)

---

Per impostazione predefinita, ogni operazione su una repository connection viene eseguita autonomamente, e ne viene fatto il commit nel repository (*autocommit*).

Tuttavia, è possibile *gestire transazioni* che permettono di raggruppare diverse operazioni, in modo da poterne fare il commit o il rollback come un'unità.

# RepositoryConnection – transazioni (2/6)

```
try(RepositoryConnection conn = rep.getConnection()) {
    conn.begin()
    try {
        // do useful stuff within the transaction
        conn.commit();
    } catch(RepositoryException e) {
        conn.rollback();
    }
}
```

```
try(RepositoryConnection conn = rep.getConnection()) {
    conn.begin()
    // do useful stuff within the transaction
    conn.commit();
}
```

Se viene sollevata un'eccezione prima del *commit()*, il metodo *close()* chiamato dal *try-with-resources* effettua il rollback ed aggiunge un messaggio nel log

# RepositoryConnection – transazioni (3/6)

Esiste un overload del metodo `begin()` che permette di specificare il livello di isolamento.

Da [http://docs.rdf4j.org/programming/#\\_transaction\\_isolation\\_levels](http://docs.rdf4j.org/programming/#_transaction_isolation_levels)

- **NONE** The lowest isolation level; transactions can see their own changes, but may not be able to roll them back, and no isolation among transactions is guaranteed. This isolation level is typically used for things like bulk data upload operations.
- **READ\_UNCOMMITTED** Transactions can be rolled back, but are not necessarily isolated: concurrent transactions may be able to see other's uncommitted data (so-called 'dirty reads').

# RepositoryConnection – transazioni (4/6)

Esiste un overload del metodo `begin()` che permette di specificare il livello di isolamento.

Da [http://docs.rdf4j.org/programming/#\\_transaction\\_isolation\\_levels](http://docs.rdf4j.org/programming/#_transaction_isolation_levels)

- **READ\_COMMITTED** In this transaction isolation level, only data from concurrent transactions that has been committed can be seen by the current transaction. However, consecutive reads within the same transaction may see different results. This isolation level is typically used for long-lived operations.
- **SNAPSHOT\_READ** In addition to being **READ\_COMMITTED**, query results in this isolation level will observe a consistent snapshot. Changes occurring to the data while a query is evaluated will not affect that query's result. This isolation level is typically used in scenarios where there multiple concurrent transactions that do not conflict with each other.

# RepositoryConnection – transazioni (5/6)

Esiste un overload del metodo `begin()` che permette di specificare il livello di isolamento.

Da [http://docs.rdf4j.org/programming/#\\_transaction\\_isolation\\_levels](http://docs.rdf4j.org/programming/#_transaction_isolation_levels)

- **SNAPSHOT** In addition to being `SNAPSHOT_READ`, successful transactions in this isolation level will operate against a particular dataset snapshot. Transactions in this isolation level will either see the complete effects of other transactions (consistently throughout) or not at all. This isolation level is typically used in scenarios where a write operation depends on the result of a previous read operation.
- **SERIALIZABLE** In addition to `SNAPSHOT`, this isolation level requires that all other transactions must appear to occur either completely before or completely after a successful serializable transaction. This isolation is typically used when multiple concurrent transactions are likely to conflict.



# RepositoryConnection – transazioni (6/6)

Il *MemoryStore* ed il *NativeStore* supportano i seguenti livelli di isolamento:

- NONE
- READ\_COMMITTED
- SNAPSHOT\_READ (*livello di isolamento predefinito*)
- SNAPSHOT
- SERIALIZABLE

Entrambi i *triplestore* usano un "*locking ottimistico*": assumendo che non si verifichino conflitti, permettono in generale scritture concorrenti da parte delle transazioni, facendo poi fallire le transazioni durante il commit in caso di problemi.

Se un *triplestore* non supporta un certo livello di isolamento, ne userà (se disponibile) uno che offre maggiori garanzie.

# RDF4J SERVER & WORKBENCH

L'**RDF4J Server** è un'applicazione web che espone un numero di Repository attraverso un'**API REST** dedicata:

<http://docs.rdf4j.org/rest-api/>

Questa API **estende** e **combina**:

- SPARQL I.I Protocol (inviare query e update SPARQL ai processori)
- SPARQL I.I Graph Store HTTP Protocol (gestione di named graph, es. aggiunta, rimozione, scaricamento, ...)

La principale estensione riguarda la **gestione delle transazioni**

Viene fornito come un WAR da usare con un servlet container tipo Apache Tomcat.

L'***RDF4J Workbench*** è un'applicazione web per accedere a repository esposti da un *RDF4J Server*.

Viene fornito come un WAR da usare con un servlet container tipo Apache Tomcat.

Nelle lezioni precedenti abbiamo usato il **Workbench di GraphDB**, che:

- Può essere avviato autonomamente senza bisogno di un servlet container esterno
- Integra la user interface (il workbench) ed il server (che espone zero o più repository tramite un'API REST consumata dal workbench)

# GraphDB Workbench – URL del Repository (per accedervi tramite API) (1/2)

The screenshot shows the GraphDB Workbench interface in a web browser. The browser address bar shows 'localhost:7200'. The interface has a dark blue header with the GraphDB logo and a 'FREE' badge. A left sidebar contains navigation links: Import, Explore, SPARQL, Monitor, Setup, and Help. The main content area is titled 'View resource' and features a search bar for 'Search RDF resources...' with 'Text' and 'Visual' view options. Below this, there are two panels: 'Active repository' and 'Saved SPARQL queries'. The 'Active repository' panel shows a local repository named 'Test' with a database icon. It displays statistics: 'total statements 951', '657 explicit', '294 inferred', and '1.45 expansion ratio'. A hand icon points to the 'Test' label. Below the statistics are links for 'Import RDF data', 'Import tabular data with OntoRefine', and 'Export RDF data'. The 'Saved SPARQL queries' panel contains sections for 'Add statements' (with a PREFIX), 'Clear graph' (with a CLEAR GRAPH query), 'Remove statements' (with a PREFIX), and 'SPARQL Select template' (with a SELECT query template).

# GraphDB Workbench

The screenshot shows the GraphDB Workbench web interface in a browser window. The address bar displays `localhost:7200`. A modal dialog is open in the center, prompting the user to press **Ctrl+C** / **Cmd+C** to copy the URL `http://192.168.1.27:7200/repositories/Test` to the clipboard and press Enter to close. The background interface includes a sidebar with navigation options: Import, Explore, SPARQL, Monitor, Setup, and Help. The main content area is divided into two panels: 'Active repository' and 'Saved SPARQL queries'. The 'Active repository' panel shows a local repository named 'Test' with 951 total statements (657 explicit, 294 inferred) and a 1.45 expansion ratio. The 'Saved SPARQL queries' panel contains sections for 'Add statements', 'Clear graph', 'Remove statements', and a 'SPARQL Select template'.

1. Eclipse RDF4J – A Java Framework for RDF.

<http://rdf4j.org/>

2. Broekstra, J., Kampman, A., & Van Harmelen, F.

(2002, June). [Sesame: A generic architecture for](#)

[storing and querying rdf and rdf schema](#). In

*International semantic web conference* (pp. 54-68).

Springer, Berlin, Heidelberg.