à l'API RDF de Jena

Ceci est un tutoriel introductif à la fois au framework de description de ressources (RDF) du W3C et à Jena, une API Java pour RDF. Il est écrit pour le développeur peu familier avec RDF et qui apprend mieux par prototypage ou, pour d'autres raisons, désire aller rapidement à l'implémentation. Une familiarité avec XML et Java est requise.

Implémenter trop vite, sans d'abord comprendre le modèle de données RDF, mène à la frustration et à la déception. Cependant étudier le modèle de données seul est assez difficile et mène souvent à des énigmes métaphysiques tortueuses. Il vaut mieux approcher la compréhension du modèle de données et son utilisation en parallèle. Apprendre un peu du modèle de données et l'essayer. Ensuite apprendre un peu plus et réessayer. Ainsi la théorie complète la pratique et la pratique complète la théorie. Le modèle de données est assez simple, donc cette approche ne devrait pas prendre trop de temps.

RDF possède une syntaxe XML et tous ceux qui sont familiers avec XML le prendront pour tel. C'est une erreur. RDF devrait être compris en termes de son modèle de données. Les données RDF peuvent être représentées en XML, mais comprendre la syntaxe est secondaire par rapport à celle de la compréhension du modèle de données.

Une implémentation de l'API Jena, incluant le code source de travail pour tous les exemples dans ce tutoriel, peut être téléchargée depuis le site officiel de Jena.

Les deux et traducteurs

Traducteur : Julien Plu

Traducteur : Thibaut Cuvelier

L'article

Publié le 25 avril 2011 - Mis à jour le 2 juillet 2012

DÉBUTANT

Durée : 2 heures

Version PDF Version hors-ligne

I. Introduction▲

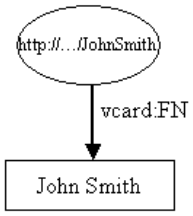
Le RDF est un standard (techniquement une recommandation du W3C) pour la

ePub, Azw et Mobi

Liens sociaux

description de ressources. Qu'est-ce qu'une ressource ? C'est une question plutôt profonde et la définition précise est toujours l'objet de débats. Pour nos fins, on peut imaginer que c'est tout ce que l'on peut identifier. Vous êtes une ressource, comme l'est votre page d'accueil, ce tutoriel ou la baleine blanche de Moby Dick.

Nos exemples dans ce tutoriel seront des personnes. Ils utilisent une représentation RDF des vCards. RDF se représente le mieux sous la forme d'un diagramme de nœuds et d'arcs. Une vCard simple pourrait ressembler à ceci en RDF :



La ressource, John Smith, est représentée comme une ellipse et est identifiée par un identifiant de ressource uniforme (URI (1)), dans ce cas http://.../JohnSmith. Si vous essayez d'accéder à cette ressource en utilisant votre navigateur, vous n'y arriverez probablement pas ; résistant à la tentation du poisson d'avril, vous seriez plutôt surpris qu'un navigateur puisse vous amener John Smith sur votre bureau. Si vous n'êtes pas familier avec les URI, pensez-y simplement comme des noms d'apparence plutôt bizarre.

Les ressources possèdent des propriétés. Dans ces exemples, nous sommes intéressés à ce genre de propriétés qui pourraient apparaître sur la carte de visite de John Smith. La première figure montre seulement une propriété, le nom complet de John Smith. Une propriété est représentée par un arc, nommé avec le nom de la propriété. Le nom d'une propriété est aussi une URI, mais comme les URI sont plutôt longues et encombrantes, le diagramme le montre dans la forme QName XML. La partie avant : s'appelle le préfixe d'espace de noms et représente un espace de noms. La partie après : s'appelle un nom local et représente un nom dans cet espace de nom. Les propriétés sont habituellement représentées dans cette forme *QName* quand elles sont écrites comme du RDF/XML et c'est un raccourci pratique pour les représenter dans des diagrammes et dans du texte. Strictement parlant, cependant, les propriétés sont identifiées par une URI. La forme nsprefix:localname est un raccourci pour l'URI de l'espace de noms concaténé avec le nom local. Il n'y a pas besoin que l'URI d'une propriété donne accès à quelque chose quand on y accède avec un navigateur.

Chaque propriété a une valeur. Dans ce cas la valeur est un littéral, que pour le moment nous considérerons comme une chaîne (2) de caractères. Ces littéraux sont montrés dans des rectangles.

Jena est une API Java qui peut être utilisée pour créer et manipuler des graphes RDF comme celui-ci. Jena possède des classes pour représenter des graphes, des ressources, des propriétés et des littéraux. Les interfaces représentant les ressources, les propriétés et les littéraux sont respectivement nommés Resource, Property et Literal. Dans Jena, un graphe est appelé un modèle et est représenté par l'interface Model.

Le code pour créer ce graphe, ou modèle, est simple :

Sélectionnez

```
// quelques définitions
static String personURI    = "http://somewhere/JohnSmith";
static String fullName     = "John Smith";

// créer un modèle vide
Model model = ModelFactory.createDefaultModel();

// créer la ressource
Resource johnSmith = model.createResource(personURI);

// ajouter la propriété
 johnSmith.addProperty(VCARD.FN, fullName);
```

Il commence avec quelques définitions de constantes et ensuite crée un Model vide utilisant la méthode createDefaultModel() de ModelFactory pour créer un modèle en mémoire. Jena contient d'autres implémentations de l'interface Model, par exemple celle qui utilise une base de données relationnelle : ces types de modèles sont aussi disponibles depuis ModelFactory.

La ressource John Smith est alors créée et une propriété y est ajoutée. La propriété est fournie par une classe « constante » de VCARD, qui détient les objets représentant toutes les définitions dans le schéma VCARD. Jena fournit des classes constantes pour d'autres schémas bien connus, comme RDF et RDF Schema eux-mêmes, Dublin Core et DAML.

Le code pour créer la ressource et ajouter la propriété peut être écrit d'une manière plus compacte avec un style en cascade :

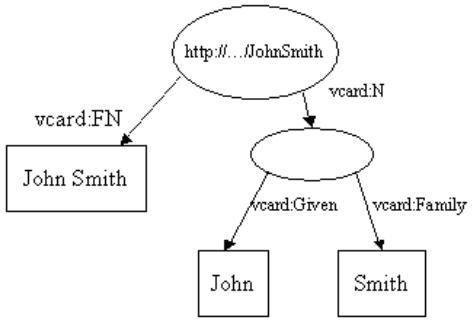
Sélectionnez

```
Resource johnSmith =
        model.createResource(personURI)
             .addProperty(VCARD.FN, fullName);
```

Le code de travail pour cet exemple peut être trouvé dans le répertoire /src-examples de la distribution Jena, dans le fichier tutorial 1. Comme exercice, prenez ce code et modifiez-le pour créer une vCard simple pour vous-même.

Maintenant, ajoutons un peu de détails à cette VCARD en explorant d'autres fonctionnalités de Jena et de RDF.

Dans le premier exemple, la valeur de la propriété était un littéral. Les propriétés RDF peuvent aussi prendre d'autres ressources comme valeur. En utilisant une technique RDF courante, cet exemple montre comment représenter les différentes parties du nom de John Smith :



Ici, nous avons ajouté une nouvelle propriété, vcard:N, pour représenter la structure du nom de John Smith. Il y a plusieurs choses dignes d'intérêt dans ce modèle. Notez que la propriété vcard:N prend une ressource comme valeur. Notez aussi que l'ellipse représentant le nom composé n'a pas d'URI. On appelle cela un nœud anonyme.

Le code Jena pour construire cet exemple est encore une fois très simple. D'abord quelques déclarations et puis la création du modèle vide.

Sélectionnez

```
// quelques définitions
String personURI    = "http://somewhere/JohnSmith";
String givenName    = "John";
String familyName   = "Smith";
String fullName     = givenName + " " + familyName;

// créer un modèle vide
Model model = ModelFactory.createDefaultModel();

// créer la ressource
//   et ajouter des propriétés en cascade
Resource johnSmith
  = model.createResource(personURI)
         .addProperty(VCARD.FN, fullName)
         .addProperty(VCARD.N,
                      model.createResource()
                           .addProperty(VCARD.Given, givenName)
                           .addProperty(VCARD.Family, familyName));
```

Le code de travail de cet exemple peut être trouvé comme tutoriel 2 dans le répertoire /src-examples de la distribution Jena.

II. Déclarations▲

Chaque arc dans un modèle RDF est appelé une déclaration. Chaque déclaration

affirme un fait à propos d'une ressource. Une déclaration se compose de trois parties :

- le sujet est la ressource que quitte l'arc ;
- le prédicat est la propriété qui donne un nom à l'arc ;
- l'objet est la ressource ou le littéral pointé par l'arc.

Une telle déclaration est quelquefois appelée un triplet, à cause de ses trois parties.

Un modèle RDF est représenté comme un ensemble de déclarations. Chaque appel à addProperty dans le tutoriel 2 ajoute une autre déclaration au modèle. Puisqu'un modèle est un ensemble de déclarations, ajouter une déclaration en double n'a pas d'effet. Les interfaces modèle de Jena définissent une méthode listStatements(), qui retourne un StmtIterator, un sous-type de la classe Java Iterator sur toutes les déclarations dans un modèle. StmtIterator possède une méthode nextStatement(), qui retourne la déclaration suivante de l'itérateur (le même que next() retournerait, déjà casté en Statement). L'interface Statement fournit des méthodes d'accès au sujet, au prédicat et à l'objet de la déclaration.

Maintenant, on va utiliser cette interface pour étendre le tutoriel 2, pour lister toutes les déclarations créées et les afficher. Le code complet pour ça se trouve dans tutoriel 3.

Sélectionnez

```
// liste des déclarations dans le modèle
StmtIterator iter = model.listStatements();

// affiche l'objet, le prédicat et le sujet de chaque déclaration
while (iter.hasNext()) {
    Statement stmt      = iter.nextStatement();  // obtenir la prochaine déclaration
    Resource  subject   = stmt.getSubject();     // obtenir le sujet
    Property  predicate = stmt.getPredicate();   // obtenir le prédicat
    RDFNode   object    = stmt.getObject();      // obtenir l'objet

    System.out.print(subject.toString());
    System.out.print(" " + predicate.toString() + " ");
    if (object instanceof Resource) {
       System.out.print(object.toString());
    } else {
        // l'objet est un littéral
        System.out.print(" \"" + object.toString() + "\"");
    }

    System.out.println(" .");
}
```

Puisque l'objet d'une déclaration peut être soit une ressource soit un littéral, la méthode getObject() retourne un objet de type RDFNode, qui est une superclasse commune à Resource et à Literal. L'objet sous-jacent est du type approprié, donc le code utilise instanceof pour le déterminer et le traiter en fonction.

Quand il est lancé, ce programme devrait produire une sortie comme ceci :

Sélectionnez

```
http://somewhere/JohnSmith http://www.w3.org/2001/vcard-rdf/3.0#N anon:14df86:ecc3de
anon:14df86:ecc3dee17b:-7fff http://www.w3.org/2001/vcard-rdf/3.0#Family  "Smith" .
anon:14df86:ecc3dee17b:-7fff http://www.w3.org/2001/vcard-rdf/3.0#Given  "John" .
http://somewhere/JohnSmith http://www.w3.org/2001/vcard-rdf/3.0#FN  "John Smith" .
```

Maintenant, vous savez pourquoi il est plus simple de dessiner des modèles. Si vous regardez plus minutieusement, vous verrez que chaque ligne consiste en trois champs, représentant le sujet, le prédicat et l'objet de chaque déclaration. Il y a quatre arcs dans le modèle, donc il y a quatre déclarations.
anon:14df86:ecc3dee17b:-7fff est un identifiant interne généré par Jena. Ce n'est pas une URI et ne doit pas être confondu avec une URI. Il s'agit simplement d'un label interne utilisé par l'implémentation de Jena.

Le groupe de travail RDFCore du W3C a défini une notation simple et similaire appelée N-Triples. Le nom signifie « triple notation ». Nous verrons dans la prochaine section que Jena a un module d'écriture de N-triples incorporé.

III. Écrire du RDF▲

Jena dispose de méthodes pour lire et écrire du RDF/XML. Elles peuvent être utilisées pour sauvegarder un modèle RDF dans un fichier et le relire plus tard.

Le tutoriel 3 a créé un modèle et l'a écrit sous forme de triplets. Le tutoriel 4 modifie le tutoriel 3 pour écrire le modèle en RDF/XML sur le flux de sortie standard. Le code est à nouveau très simple : model.write peut prendre un argument OutputStream.

Sélectionnez

```
// maintenant écrit le modèle sous forme XML dans un fichier
model.write(System.out);
```

La sortie devrait ressembler à quelque chose comme ça :

Sélectionnez

```
<rdf:RDF
  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
  xmlns:vcard='http://www.w3.org/2001/vcard-rdf/3.0#'
 >
  <rdf:Description rdf:about='http://somewhere/JohnSmith'>
    <vcard:FN>John Smith</vcard:FN>
    <vcard:N rdf:nodeID="A0"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="A0">
    <vcard:Given>John</vcard:Given>
    <vcard:Family>Smith</vcard:Family>
  </rdf:Description>
</rdf:RDF>
```

Les spécifications RDF spécifient comment représenter le RDF comme du XML. La syntaxe RDF/XML est assez complexe. Le lecteur est renvoyé au Primer, en cours de développement par le groupe de travail RDFCore pour une introduction plus détaillée. Cependant, regardons rapidement comment interpréter ce code.

RDF est habituellement inclus dans un élément <rdf:RDF>. L'élément est optionnel s'il y a d'autres manières de savoir que du XML est du RDF, mais il est généralement présent. L'élément RDF définit les deux espaces de noms utilisés dans le document. Il y a ensuite un élément <rdf:description> qui décrit la ressource dont l'URI est http://somewhere/JohnSmith. Si l'attribut rdf:about était absent, cet élément représenterait un nœud anonyme.

L'élément <vcard:FN> décrit une propriété de la ressource. Le nom de la propriété est FN dans l'espace de noms vcard. RDF convertit ceci en une référence à une URI en concaténant la référence d'URI pour le préfixe de l'espace de noms et FN, la partie du nom local du nom. Ceci donne une référence d'URI de http://www.w3.org /2001/vcard-rdf/3.0#FN. La valeur de la propriété est le littéral John Smith.

L'élément <vcard:N> est une ressource. Dans ce cas, la ressource est représentée par une référence d'URI relative. RDF convertit cela en une référence d'URI absolue en la concaténant avec l'URI de base du document courant.

Il y a une erreur dans ce RDF/XML ; il ne représente pas exactement le même modèle que celui que nous avons créé. Le nœud anonyme dans le modèle a été donné par une référence d'URI. Il n'est donc plus anonyme. La syntaxe RDF/XML n'est pas capable de représenter tous les modèles RDF ; par exemple il ne peut pas représenter un nœud anonyme qui est l'objet de deux déclarations. Le module d'écriture « idiot » que nous utilisons pour écrire ce RDF/XML n'essaie pas d'écrire correctement le sous-ensemble des modèles qui peuvent être écrits correctement. Il donne une URI à chaque nœud anonyme, ne le rendant plus anonyme.

Jena a une interface extensible qui permet à de nouveaux modules d'écriture pour d'autres langages de sérialisation pour RDF d'être facilement ajoutés dessus. L'appel ci-dessus a invoqué le module d'écriture « idiot » standard. Jena inclut aussi un module d'écriture RDF/XML plus sophistiqué qui peut être invoqué en spécifiant un autre argument lors de l'appel à la méthode write() :

Sélectionnez

```
// écrit maintenant le modèle en XML dans un fichier
model.write(System.out, "RDF/XML-ABBREV");
```

Ce module d'écriture, appelé « PrettyWritter », prend avantage des fonctionnalités de la syntaxe abrégée de RDF/XML pour écrire un modèle plus compact. Il est aussi en mesure de préserver les nœuds anonymes où cela est possible. Il n'est cependant pas approprié pour de grands modèles, la probabilité pour que ses performances soient acceptables étant très faible. Pour écrire de grands fichiers et préserver les nœuds anonymes, les écrire au format N-Triples :

Sélectionnez

```
// écrit maintenant le modèle en N-Triples dans un fichier
model.write(System.out, "N-TRIPLE");
```

Ceci produira une sortie similaire à celle du tutoriel 3 qui se conforme à la spécification N-Triples.

IV. Lire du RDF▲

Le tutoriel 5 démontre comment lire les déclarations enregistrées au format RDF/XML dans un modèle. Avec ce tutoriel, nous avons fourni une petite base de données de VCARD au format RDF/XML. Le code suivant le lira et l'écrira dans le flux standard de sortie. *Notez que pour que cette application se lance, le fichier d'entrée doit être dans le répertoire courant.*

Sélectionnez

```
// créer un modèle vide
 Model model = ModelFactory.createDefaultModel();

 // utiliser le FileManager pour trouver le fichier d'entrée
 InputStream in = FileManager.get().open( inputFileName );
if (in == null) {
    throw new IllegalArgumentException(
                                  "Fichier: " + inputFileName + " non trouvé");
}

// lire le fichier RDF/XML
model.read(in, null);

// l'écrire sur la sortie standard
model.write(System.out);
```

Le second argument de la méthode read() appelée est l'URI qui sera utilisée pour résoudre les URI relatives. Comme il n'y a pas de références aux URI relatives dans le fichier de test, il est permis qu'il soit vide. Au lancement, ce tutoriel 5 produira une sortie XML qui ressemble à ceci :

Sélectionnez

```
<rdf:RDF
  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
  xmlns:vcard='http://www.w3.org/2001/vcard-rdf/3.0#'
 >
  <rdf:Description rdf:nodeID="A0">
    <vcard:Family>Smith</vcard:Family>
    <vcard:Given>John</vcard:Given>
  </rdf:Description>
  <rdf:Description rdf:about='http://somewhere/JohnSmith/'>
    <vcard:FN>John Smith</vcard:FN>
    <vcard:N rdf:nodeID="A0"/>
  </rdf:Description>
  <rdf:Description rdf:about='http://somewhere/SarahJones/'>
    <vcard:FN>Sarah Jones</vcard:FN>
    <vcard:N rdf:nodeID="A1"/>
  </rdf:Description>
  <rdf:Description rdf:about='http://somewhere/MattJones/'>
    <vcard:FN>Matt Jones</vcard:FN>
    <vcard:N rdf:nodeID="A2"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="A3">
    <vcard:Family>Smith</vcard:Family>
    <vcard:Given>Rebecca</vcard:Given>
  </rdf:Description>
  <rdf:Description rdf:nodeID="A1">
    <vcard:Family>Jones</vcard:Family>
    <vcard:Given>Sarah</vcard:Given>
  </rdf:Description>
  <rdf:Description rdf:nodeID="A2">
    <vcard:Family>Jones</vcard:Family>
    <vcard:Given>Matthew</vcard:Given>
  </rdf:Description>
  <rdf:Description rdf:about='http://somewhere/RebeccaSmith/'>
    <vcard:FN>Becky Smith</vcard:FN>
    <vcard:N rdf:nodeID="A3"/>
  </rdf:Description>
</rdf:RDF>
```

V. Préfixes de contrôle▲

V-A. Définitions de préfixes explicites▲

Dans la section précédente, nous avons vu que la sortie XML déclarait un préfixe d'espace de noms vcard et utilisait ce préfixe pour abréger les URI. Alors que RDF utilise seulement des URI complètes, et pas cette forme raccourcie, Jena fournit des façons de contrôler les espaces de noms utilisés sur la sortie avec ses *associations de préfixes*. Voici un exemple simple :

Sélectionnez

```
Model m = ModelFactory.createDefaultModel();
 String nsA = "http://somewhere/else#";
 String nsB = "http://nowhere/else#";
 Resource root = m.createResource( nsA + "root" );
 Property P = m.createProperty( nsA + "P" );
 Property Q = m.createProperty( nsB + "Q" );
 Resource x = m.createResource( nsA + "x" );
 Resource y = m.createResource( nsA + "y" );
 Resource z = m.createResource( nsA + "z" );
 m.add( root, P, x ).add( root, P, y ).add( y, Q, z );
 System.out.println( "# -- pas de préfixes spéciaux définis" );
 m.write( System.out );
 System.out.println( "# -- nsA défini" );
 m.setNsPrefix( "nsA", nsA );
 m.write( System.out );
 System.out.println( "# -- nsA et cat définis" );
 m.setNsPrefix( "cat", nsB );
 m.write( System.out );
```

La sortie de ce fragment est constituée de trois morceaux de RDF/XML, avec trois associations de préfixes différents. D'abord celui par défaut, sans préfixe autre que les standards :

Sélectionnez

```
# -- no special prefixes defined

<rdf:RDF
    xmlns:j.0="http://nowhere/else#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:j.1="http://somewhere/else#" >
  <rdf:Description rdf:about="http://somewhere/else#root">
    <j.1:P rdf:resource="http://somewhere/else#x"/>
    <j.1:P rdf:resource="http://somewhere/else#y"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://somewhere/else#y">
    <j.0:Q rdf:resource="http://somewhere/else#z"/>
  </rdf:Description>
</rdf:RDF>
```

Nous voyons que l'espace de nom rdf est déclaré automatiquement, puisqu'il est requis pour des balises comme <RDF:rdf> et <rdf:resource>. Les déclarations d'un espace de noms sont aussi requises pour l'utilisation des deux propriétés P et Q, mais puisque leurs espaces de noms n'ont pas été introduits dans le modèle, ils obtiennent des espaces de noms inventés : j.0 et j.1.

La méthode setNsPrefix(String prefix, String URI) déclare que l'espace de nom URI peut être abrégé par prefix. Jena requiert que prefix soit un nom d'espace de nom légal en XML et qu'URI se termine par un caractère qui n'est pas un nom. Le module d'écriture RDF/XML transformera ces déclarations de préfixes en déclarations d'espaces de noms XML et les utilise dans sa sortie :

Sélectionnez

```
# -- nsA defined

<rdf:RDF
    xmlns:j.0="http://nowhere/else#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:nsA="http://somewhere/else#" >
  <rdf:Description rdf:about="http://somewhere/else#root">
    <nsA:P rdf:resource="http://somewhere/else#x"/>
    <nsA:P rdf:resource="http://somewhere/else#y"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://somewhere/else#y">
    <j.0:Q rdf:resource="http://somewhere/else#z"/>
  </rdf:Description>
</rdf:RDF>
```

L'autre espace de nom reçoit toujours un nom construit, mais le nom nsA est maintenant utilisé dans les étiquettes de propriétés. Il n'y a pas besoin pour le nom du préfixe d'avoir quoi que ce soit à faire avec les variables dans le code Jena :

Sélectionnez

```
# -- nsA and cat defined

<rdf:RDF
    xmlns:cat="http://nowhere/else#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:nsA="http://somewhere/else#" >
  <rdf:Description rdf:about="http://somewhere/else#root">
    <nsA:P rdf:resource="http://somewhere/else#x"/>
    <nsA:P rdf:resource="http://somewhere/else#y"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://somewhere/else#y">
    <cat:Q rdf:resource="http://somewhere/else#z"/>
  </rdf:Description>
</rdf:RDF>
```

Les deux préfixes sont utilisés pour la sortie, et aucun préfixe généré n'est requis.

V-B. Définitions de préfixes implicites▲

Aussi bien que pour les déclarations de préfixes fournies par les appels à setNsPrefix, Jena se souviendra des préfixes qui ont été utilisés dans l'entrée de model.read().

Prenez la sortie produite par le fragment précédent et collez-la dans un fichier quelconque, avec l'URL file:/tmp/fragment.rdf par exemple. Ensuite, lancez ce code :

Sélectionnez

```
Model m2 = ModelFactory.createDefaultModel();
m2.read( "file:/tmp/fragment.rdf" );
m2.write( System.out );
```

Vous verrez que les préfixes de l'entrée sont préservés dans la sortie. Tous les préfixes sont écrits, même s'ils ne sont utilisés nulle part. On peut supprimer un préfixe avec removeNsPrefix(String prefix) si vous ne voulez pas le voir dans la sortie.

Puisque les N-Triples n'ont pas de courte façon d'écrire les URI, il ne tient aucun compte des préfixes sur la sortie et n'en fournit aucun à l'entrée. La notation **N3**, aussi supportée par Jena, possède de tels noms de préfixes raccourcis, et les enregistre sur l'entrée et les utilise sur la sortie.

Jena a davantage d'opérations sur les associations de préfixes détenues par un modèle, comme l'extraction d'une Map Java des associations sortantes, ou l'ajout d'un groupe entier d'associations à la fois ; voir la documentation de PrefixMapping pour les détails.

VI. Paquets RDF de Jena▲

Jena est une API Java pour les applications Web sémantique. Le paquet-clé RDF pour le développeur d'applications est com.hp.hpl.jena.rdf.model. L'API a été définie en termes d'interfaces donc le code de l'application peut fonctionner avec différentes implémentations sans changement. Ce paquet contient des interfaces pour représenter des modèles, des ressources, des propriétés, des littéraux, des déclarations et tous les autres concepts-clés de RDF, et une ModelFactory pour créer des modèles. Ainsi ce code d'application reste indépendant de l'implémentation, il est préférable s'il utilise des interfaces dans la mesure du possible et pas les implémentations de classes spécifiques.

Le paquet com.hp.hpl.jena.tutorial contient le code source du travail pour tous les exemples utilisés dans ce tutoriel.

Les paquets com.hp.hpl...jenaimpl contiennent l'implémentation des classes qui peuvent être communes à plusieurs implémentations. Par exemple, elles définissent les classes ResourceImpl, PropertyImpl et LiteralImpl, qui peuvent être utilisées directement ou dérivées par différentes implémentations. Les applications doivent rarement, voire jamais, utiliser ces classes directement. Par exemple, plutôt que de créer une nouvelle instance de ResourceImpl, il vaut mieux utiliser la méthode createResource quel que soit le modèle utilisé. Ainsi, si l'implémentation du modèle a utilisé une implémentation optimisée de Resource, alors aucune

conversion entre les deux types ne sera nécessaire.

VII. Naviguer dans un modèle▲

Jusqu'ici, ce tutoriel a traité principalement de la création, de la lecture et de l'écriture de modèles RDF. Il est maintenant temps de traiter de l'accès à l'information tenue dans un modèle.

Compte tenu de l'URI d'une ressource, l'objet de ressource peut être récupéré depuis un modèle en utilisant la méthode Model.getResource(String uri). Cette méthode est définie pour retourner un objet Resource s'il en existe un dans le modèle, ou autrement d'en créer un nouveau. Par exemple, pour récupérer la ressource John Smith du modèle lu du fichier dans le tutoriel 5 :

Sélectionnez

```
// récupère la ressource vcard de John Smith du modèle
Resource vcard = model.getResource(johnSmithURI);
```

L'interface Resource définit un nombre de méthodes pour accéder aux propriétés d'une ressource. La méthode Resource.getProperty(Property p) accède à la propriété de la ressource. Cette méthode ne suit pas l'habituelle convention des accesseurs de Java en ce que le type de l'objet retourné est Statement, pas la Property que vous attendiez. Retourner l'ensemble de la déclaration permet à l'application d'accéder à la valeur de la propriété utilisant une de ses méthodes accesseurs qui retourne l'objet de la déclaration. Par exemple pour récupérer la ressource qui est la valeur de la propriété vcard:N :

Sélectionnez

```
// récupérer la valeur de la propriété N
Resource name = (Resource) vcard.getProperty(VCARD.N)
                                .getObject();
```

En général, l'objet de la déclaration pourrait être une ressource ou un littéral, ainsi le code de l'application, sachant que la valeur doit être une ressource, caste l'objet retourné. L'une des choses que Jena essaie de faire est de fournir des méthodes de types spécifiques ainsi l'application n'a pas à caster et la vérification du type peut être faite au moment de la compilation. Le fragment de code ci-dessus peut être plus commodément écrit :

Sélectionnez

```
// récupère la valeur de la propriété FN
Resource name = vcard.getProperty(VCARD.N)
                     .getResource();
```

De même, la valeur littérale d'une propriété peut être récupérée :

Sélectionnez

```
// récupère le nom de la propriété donnée
String fullName = vcard.getProperty(VCARD.FN)
                       .getString();
```

Dans cet exemple, la ressource VCARD a seulement une propriété vcard:FN et une propriété vcard:N. RDF autorise une ressource à répéter une propriété ; par exemple John peut avoir plus d'un surnom. Donnons-lui en deux :

Sélectionnez

```
// ajoute deux propriétés surnoms à la vcard
vcard.addProperty(VCARD.NICKNAME, "Smithy")
     .addProperty(VCARD.NICKNAME, "Adman");
```

Comme précédemment noté, Jena représente un modèle RDF *comme un ensemble* de déclarations, ainsi ajouter une déclaration avec le sujet, le prédicat et l'objet comme celui déjà dans le modèle n'aura aucun effet. Jena ne définit pas lequel des deux surnoms présents dans le modèle sera retourné. Le résultat de l'appel à vcard.getProperty(VCARD.NICKNAME) est indéterminé. Jena retournera une des valeurs, mais il n'y a aucune garantie que deux appels consécutifs retournent la même valeur.

S'il est possible qu'une propriété apparaisse plus d'une fois, alors la méthode Resource.listProperties(Property p) peut être utilisée pour retourner un itérateur qui les listera tous. Cette méthode renvoie un itérateur qui retourne des objets de type

Statement. On peut lister les surnoms comme ceci :

Sélectionnez

```
// mettre en place la sortie
System.out.println("Les surnoms de \""
                    + fullName + "\" sont :");
// liste les surnoms
StmtIterator iter = vcard.listProperties(VCARD.NICKNAME);
while (iter.hasNext()) {
    System.out.println("    " + iter.nextStatement()
                                    .getObject()
                                    .toString());
}
```

Ce code peut être trouvé dans le tutoriel 6. L'itérateur de déclarations iter produit toutes les déclarations avec un sujet vcard et un prédicat VCARD.NICKNAME, ainsi boucler sur lui nous permet d'aller chercher chaque déclaration en utilisant nextStatement(), récupérant le champ objet, et en le convertissant en chaîne de caractères. Le code produit la sortie suivante lorsqu'il est exécuté :

Sélectionnez

```
Les surnoms de "John Smith" sont :
    Smithy
    Adman
```

Toutes les propriétés d'une ressource peuvent être listées en utilisant la méthode listProperties() sans un argument.

VIII. Requêter un modèle▲

La section précédente traitait du cas de la navigation d'un modèle à partir d'une ressource avec une URI connue. Cette section traite de la recherche d'un modèle. Le noyau de l'API Jena supporte seulement une requête limitée primitive. Les équipements les plus puissants de la requête RDQL sont décrits ailleurs.

La méthode Model.listStatements(), qui liste toutes les déclarations dans un modèle, est peut-être la plus cruelle façon de requêter un modèle. Son utilisation n'est pas recommandée sur les très grands modèles. Model.listSubjects() est similaire, mais renvoie un itérateur sur toutes les ressources qui ont des propriétés, c'est-à-dire qui sont l'objet de quelques déclarations.

Model.listSubjectsWithProperty(Property p, RDFNode o) retournera un itérateur sur toutes les ressources qui ont la propriété p avec la valeur o. Si nous supposons que seules les ressources de VCARD ont la propriété vcard:FN, et que dans nos données, toutes ces ressources ont une telle propriété, alors nous pouvons trouver toutes les vcard comme ceci :

Sélectionnez

```
// liste les vcard
ResIterator iter = model.listSubjectsWithProperty(VCARD.FN);
while (iter.hasNext()) {
    Resource r = iter.nextResource();
    ...
}
```

Toutes ces méthodes de requête sont simplement du sucre syntaxique sur une méthode de requête primitive model.listStatements(Selector s). Cette méthode renvoie un itérateur sur toutes les déclarations dans le modèle « sélectionné » par s. L'interface de sélection est conçue pour être extensible, mais pour l'instant, il y a seulement une implémentation de celle-ci, la classe SimpleSelector du paquet com.hp.hpl.jena.rdf.model. L'utilisation de SimpleSelector est une des rares occasions dans Jena où il est nécessaire d'utiliser une classe spécifique plutôt qu'une interface. Le constructeur de SimpleSelector prend trois arguments :

Sélectionnez

```
Selector selector = new SimpleSelector(subject, predicate, object)
```

Ce sélecteur sélectionnera toutes les déclarations ayant un sujet qui correspond à subject, un prédicat qui correspond à predicate et un objet qui correspond à object. Si la valeur null est fournie dans toutes les positions, il correspond à tout ; autrement il correspond à des ressources égales correspondantes ou bien à des littéraux. Deux ressources sont égales si elles ont les mêmes URI ou le même nœud anonyme ; deux littéraux sont les mêmes si tous leurs composants sont

égaux. Ainsi :

Sélectionnez

```
Selector selector = new SimpleSelector(null, null, null);
```

sélectionnera toutes les déclarations dans un modèle.

Sélectionnez

```
Selector selector = new SimpleSelector(null, VCARD.FN, null);
```

sélectionnera toutes les déclarations ayant VCARD.FN comme prédicat, peu importe le sujet et l'objet. Comme un raccourci spécial,

Sélectionnez

```
listStatements( S, P, O )
```

est équivalent à

Sélectionnez

```
listStatements( new SimpleSelector( S, P, O ) )
```

Le code suivant, qui peut être trouvé dans son intégralité dans le tutoriel 7 liste les noms complets sur toutes les vcard dans la base de données.

Sélectionnez

```
// sélectionne toutes les ressources avec une propriété VCARD.FN
ResIterator iter = model.listSubjectsWithProperty(VCARD.FN);
if (iter.hasNext()) {
    System.out.println("La base de données contient les vcard de :");
    while (iter.hasNext()) {
        System.out.println("  " + iter.nextStatement()
                                      .getProperty(VCARD.FN)
                                      .getString());
    }
} else {
    System.out.println("Aucune vcard n'a été trouvée dans la base de données");
}
```

Cela doit produire une sortie similaire à ce qui suit :

Sélectionnez

```
La base de données contient les vcard de :
  Sarah Jones
  John Smith
  Matt Jones
  Becky Smith
```

Votre prochain exercice est de modifier ce code pour utiliser SimpleSelector à la place de listSubjectsWithProperty.

Voyons comment implémenter un certain contrôle plus fin sur les déclarations sélectionnées. SimpleSelector peut être dérivé et sa méthode de sélection modifiée pour effectuer un filtrage supplémentaire :

Sélectionnez

```
// sélectionne toutes les ressources avec une propriété VCARD.FN
// dont la valeur se termine avec "Smith"
StmtIterator iter = model.listStatements(
    new SimpleSelector(null, VCARD.FN, (RDFNode) null) {
        public boolean selects(Statement s)
            {return s.getString().endsWith("Smith");}
    });
```

Cet échantillon de code utilise une technique propre à Java qui consiste à surcharger la définition d'une méthode inline lors de la création d'une instance de la classe. Ici, la méthode selects(...) vérifie que le nom complet se termine par « Smith ». Il est important de noter que le filtrage basé sur les arguments sujet, prédicat et objet prend place avant que la méthode selects(...) soit appelée, de sorte que le test supplémentaire ne sera appliqué qu'aux déclarations correspondantes.

Le code complet peut être trouvé dans le tutoriel 8 et produit une sortie comme ceci :

Sélectionnez

```
La base de données contient des vcard pour :
  John Smith
  Becky Smith
```

Vous pourriez penser que :

Sélectionnez

```
/ fait tout le filtrage dans la méthode selects
StmtIterator iter = model.listStatements(
  new
      SimpleSelector(null, null, (RDFNode) null) {
          public boolean selects(Statement s) {
              return (subject == null   || s.getSubject().equals(subject))
                  && (predicate == null || s.getPredicate().equals(predicate))
                  && (object == null    || s.getObject().equals(object))
          }
      }
 });
```

est équivalent à :

Sélectionnez

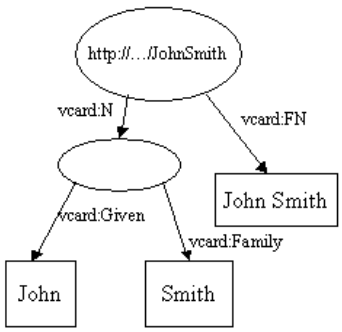
```
StmtIterator iter =
  model.listStatements(new SimpleSelector(subject, predicate, object)
```

Alors que fonctionnellement elles peuvent être équivalentes, la première forme listera tous les états dans le modèle et testera chacun d'eux individuellement, tandis que la seconde permettra de maintenir les index par l'implémentation pour améliorer les performances. Essayez-le sur un grand modèle et voyez par vous-même, mais faites une tasse de café d'abord.

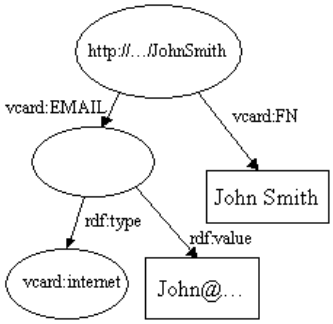
IX. Opérations sur les modèles▲

Jena fournit trois opérations pour manipuler des modèles dans leur ensemble. Ce sont des opérations ensemblistes communes d'union, d'intersection et de différence.

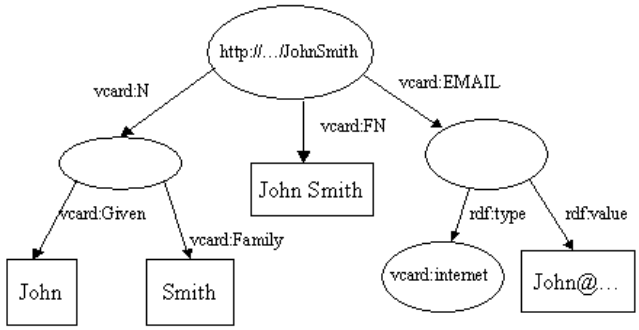
L'union de deux modèles est l'union des ensembles de déclarations qui représentent chaque modèle. C'est l'une des opérations clés que la conception de RDF supporte. Il permet à des données provenant de sources de données différentes d'être fusionnées. Considérons les deux modèles suivants :



et :



Quand ils sont fusionnés, les deux nœuds http://.../JohnSmith sont fusionnés en un seul et l'arc vcard:FN dupliqué est supprimé pour produire :



Regardons le code pour faire ça (le code complet est dans le tutoriel 9) et voyons ce qu'il se passe.

Sélectionnez

```
// lit les fichiers RDF/XML
model1.read(new InputStreamReader(in1), "");
model2.read(new InputStreamReader(in2), "");

// fusionne les modèles
Model model = model1.union(model2);

// affiche le modèle comme RDF/XML
model.write(system.out, "RDF/XML-ABBREV");
```

La sortie produite par le module d'écriture élégant ressemble à ceci :

Sélectionnez

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:vcard="http://www.w3.org/2001/vcard-rdf/3.0#">
  <rdf:Description rdf:about="http://somewhere/JohnSmith/">
    <vcard:EMAIL>
      <vcard:internet>
        <rdf:value>John@somewhere.com</rdf:value>
      </vcard:internet>
    </vcard:EMAIL>
    <vcard:N rdf:parseType="Resource">
      <vcard:Given>John</vcard:Given>
      <vcard:Family>Smith</vcard:Family>
    </vcard:N>
    <vcard:FN>John Smith</vcard:FN>
  </rdf:Description>
</rdf:RDF>
```

Même si vous n'êtes pas familier avec les détails de la syntaxe RDF/XML, il devrait être raisonnablement clair que les modèles ont fusionné comme prévu.
L'intersection et la différence des modèles peuvent être calculées de manière similaire, utilisant les méthodes .intersection(Model) et .difference(Model) ; voir la Javadoc de la différence et de l'intersection pour plus de détails.

X. Les conteneurs▲

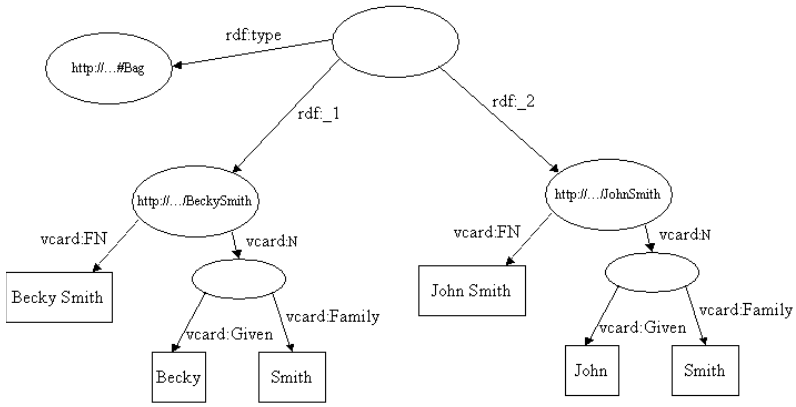
RDF définit un type particulier de ressources pour représenter des collections de

choses. Ces ressources sont appelées des conteneurs. Les membres d'un conteneur peuvent être soit des littéraux, soit des ressources. Il y a trois types de conteneurs :

- un BAG est une collection non ordonnée ;
- une ALT est une collection non ordonnée destinée à représenter des alternatives ;
- une SEQ est une collection ordonnée.

Un conteneur est représenté par une ressource. Cette ressource aura une propriété rdf:type dont la valeur devrait être l'un des rdf:Bag, rdf:Alt ou rdf:Seq, ou une sous-classe de l'un d'eux, selon le type du conteneur. Le premier membre du conteneur est la valeur de la propriété rdf:_1 du conteneur ; le second est la valeur de la propriété rdf:_2 du conteneur et ainsi de suite. Les propriétés rdf:_nnn sont connues comme des *propriétés ordinales*.

Par exemple, le modèle d'un simple sac contenant les vcard des Smith pourrait ressembler à ceci :



Alors que les membres du sac sont représentés par les propriétés rdf:_1, rdf:_2, etc., l'ordre des propriétés n'est pas significatif. On pourrait échanger les valeurs de l'attribut des propriétés rdf:_1 et rdf:_2 et le modèle résultant représenterait la même information.

Les Alt sont destinées à représenter des alternatives. Par exemple, prenons une ressource qui représente un produit logiciel. Il pourrait avoir une propriété pour indiquer d'où il pourrait être obtenu. La valeur de cette propriété pourrait être une collection Alt contenant différents sites à partir desquels elle peut être téléchargée. Les Alt ne sont pas ordonnées, cependant la propriété rdf:_1 a une signification particulière : elle représente le choix par défaut.

Alors que les conteneurs peuvent être manipulés en utilisant les mécanismes de base des ressources et des propriétés, Jena possède d'explicites interfaces et implémentations de classes pour les manipuler. Ce n'est pas une bonne idée d'avoir un objet manipulant un conteneur, et à la même période de modifier l'état de ce conteneur en utilisant les méthodes de plus bas niveau.

Essayons de modifier le tutoriel 8 pour créer ce sac :

Sélectionnez

```
// créer un sac
Bag smiths = model.createBag();

// sélectionner toutes les ressources avec une propriété VCARD.FN
// dont la valeur se termine avec "Smith"
StmtIterator iter = model.listStatements(
    new SimpleSelector(null, VCARD.FN, (RDFNode) null) {
        public boolean selects(Statement s) {
                return s.getString().endsWith("Smith");
        }
    });
// ajouter les Smith au sac
while (iter.hasNext()) {
    smiths.add(iter.nextStatement().getSubject());
}
```

Si nous écrivons ce modèle, il contient quelque chose comme ce qui suit :

Sélectionnez

```
<rdf:RDF
  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
  xmlns:vcard='http://www.w3.org/2001/vcard-rdf/3.0#'
 >
...
  <rdf:Description rdf:nodeID="A3">
    <rdf:type rdf:resource='http://www.w3.org/1999/02/22-rdf-syntax-ns#Bag'/>
    <rdf:_1 rdf:resource='http://somewhere/JohnSmith/'/>
    <rdf:_2 rdf:resource='http://somewhere/RebeccaSmith/'/>
  </rdf:Description>
</rdf:RDF>
```

qui représente la ressource sac.

L'interface du conteneur fournit un itérateur pour lister le contenu d'un conteneur :

Sélectionnez

```
// affiche les membres du sac
NodeIterator iter2 = smiths.iterator();
if (iter2.hasNext()) {
    System.out.println("Le sac contient :");
    while (iter2.hasNext()) {
        System.out.println("  " +
            ((Resource) iter2.next())
                            .getProperty(VCARD.FN)
                            .getString());
    }
} else {
    System.out.println("Le sac est vide");
}
```

qui produit la sortie suivante :

Sélectionnez

```
Le sac contient :
  John Smith
  Becky Smith
```

Un exemple de code exécutable peut être trouvé dans le tutoriel 10, qui recolle les morceaux ci-dessus dans un exemple complet.

Les classes de Jena offrent des méthodes pour manipuler les conteneurs incluant l'ajout de nouveaux membres, l'insertion de nouveaux membres au milieu d'un conteneur et supprimer des membres existants. Les classes conteneurs de Jena garantissent actuellement que la liste des propriétés ordinales utilisées commence à rdf:_01 et soit contiguë. Le groupe de travail RDFCore a assoupli cette contrainte, ce qui permet une représentation partielle des conteneurs. Il s'agit donc d'une zone de Jena qui peut être changée dans le futur.

XI. Plus à propos des littéraux et des types de données▲

Les littéraux RDF ne sont pas de simples chaînes de caractères. Les littéraux peuvent avoir une étiquette de langue pour indiquer la langue du littéral. Le littéral « chat » avec une étiquette de langue anglaise est considéré comme différent du littéral « chat » avec une étiquette de langue française. Ce comportement plutôt étrange est un artefact de la syntaxe RDF/XML originale.

En outre, il y a vraiment deux sortes de littéraux. Dans l'une, la composante chaîne de caractères est juste ça, une chaîne de caractères ordinaire. Dans l'autre composante chaîne de caractères, cela devrait être un fragment bien équilibré de XML. Quand un modèle RDF est écrit comme RDF/XML une construction spéciale utilisant un attribut parseType='Literal' est utilisé pour le représenter.

Dans Jena, ces attributs d'un littéral peuvent être définis quand le littéral est construit, par exemple dans le tutoriel 11 :

Sélectionnez

```
// crée la ressource
Resource r = model.createResource();

// ajoute la propriété
r.addProperty(RDFS.label, model.createLiteral("chat", "en"))
 .addProperty(RDFS.label, model.createLiteral("chat", "fr"))
```

```
 .addProperty(RDFS.label, model.createLiteral("<em>chat</em>", true));

// écrit le model
model.write(system.out);
```

produit :

Sélectionnez

```
<rdf:RDF
  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
  xmlns:rdfs='http://www.w3.org/2000/01/rdf-schema#'
 >
  <rdf:Description rdf:nodeID="A0">
    <rdfs:label xml:lang='en'>chat</rdfs:label>
    <rdfs:label xml:lang='fr'>chat</rdfs:label>
    <rdfs:label rdf:parseType='Literal'><em>chat</em></rdfs:label>
  </rdf:Description>
</rdf:RDF>
```

Pour que deux littéraux soient considérés comme égaux, ils doivent être tous les deux soit des littéraux XML soit de simples littéraux. En outre, soit les deux ne doivent avoir aucune étiquette de langue, soit les étiquettes de langues sont présentes mais elles doivent être égales. Pour de simples littéraux, les chaînes de caractères doivent être égales. Les littéraux XML ont deux notions d'égalité. La notion simple est que les conditions précédemment mentionnées soient vraies et les chaînes de caractères soient aussi égales. L'autre notion est qu'ils peuvent être égaux si la canonisation de leur chaîne de caractères est égale.

Les interfaces de Jena supportent aussi les littéraux typés. L'ancienne façon (voir ci-dessous) traite les littéraux typés comme un raccourci pour les chaînes de caractères : les valeurs typées sont converties selon la méthode habituelle de Java en chaînes de caractères et ces chaînes de caractères sont stockées dans le modèle. Par exemple, essayez (en notant que pour de simples littéraux, nous pouvons omettre l'appel à model.createLiteral()) :

Sélectionnez

```
// créer la ressource
Resource r = model.createResource();

// ajoute la propriété
r.addProperty(RDFS.label, "11")
 .addProperty(RDFS.label, 11);

// affiche le model
model.write(system.out, "N-TRIPLE");
```

La sortie produite est :

Sélectionnez

```
_:A... <http://www.w3.org/2000/01/rdf-schema#label> "11" .
```

Étant donné que les deux littéraux sont réellement juste la chaîne de caractères « 11 », alors une seule déclaration est ajoutée.

Le groupe de travail RDFCore a défini des mécanismes pour soutenir les types de données en RDF. Jena supporte ces utilisations des mécanismes de *littéral typé* ; ils ne sont pas abordés dans ce tutoriel.

XII. Glossaire▲

Nœud anonyme : représente une ressource, mais n'indique pas une URI pour la ressource. Les nœuds anonymes agissent comme des variables existentiellement qualifiées dans la logique du premier ordre.

Dublin Core : un standard pour les métadonnées sur les ressources Web. De plus amples informations peuvent être trouvées sur le site Web de Dublin Core.

Littéral : une chaîne de caractères qui peut être la valeur d'une propriété.

Objet : la partie d'un triplet qui est la valeur de la déclaration.

Prédicat : la partie de la propriété dans un triplet.

Propriété : une propriété est un attribut d'une ressource. Par exemple, DC.Title est une propriété, comme l'est RDF.type.

Ressource : certaines entités. Elle pourrait être une ressource Web comme une page Web, ou une chose physique concrète comme un arbre ou une voiture. Elle pourrait être une idée abstraite comme les échecs ou le football. Les ressources sont nommées par une URI.

Déclaration : un arc dans un modèle RDF, normalement interprété comme un fait.

Sujet : la ressource qui est la source d'un arc dans un modèle RDF.

Triplet : une structure contenant un sujet, un prédicat et un objet. Un autre terme pour une déclaration.

XIII. L'article original▲

Cet article est la traduction de An Introduction to RDF and the Jena RDF API.

XIV. Remerciements▲

Merci à Claude Leloup, jacques_jean, Mathieu Robin et _Max_ pour leur relecture orthographique.

---

(1)
Uniform Ressource Identifier. L'identifiant d'une ressource RDF peut inclure un identifiant de fragment, par exemple http://hostname/rdf/tutorial/#ch-Introduction, donc, à proprement parler, un ressource RDF est identifiée par une URI de référence.
(2)
En plus d'être une chaîne de caractères, les littéraux ont aussi un encodage de langue optionnel pour représenter la langue de la chaîne de caractère. Par exemple le littéral « two » doit avoir comme encodage de langue « en » pour l'anglais et le littéral « deux » doit avoir comme encodage de langue « fr » pour le français.
