Université de Sherbrooke
Département d'informatique

Exploitation de bases de données relationnelles et orientées objet IFT287

Notes complémentaires et synthétiques

Marc Frappier, Ph.D.
professeur

UNIVERSITÉ DE
SHERBROOKE

Avertissement

Ce document n'est pas un substitut au livre de référence du cours ni aux manuels de référence des différents langages utilisés dans le cadre du cours.

Contents

Chapter 8

Le langage SQL

- le langage SQL est une norme de l'ANSI (SQL-92) et de l'ISO (SQL2) pour les SGBD relationnels
- il comporte à la fois des énoncés de définitions des données (LDD) et de manipulation des données (LMD).
- il existe 3 niveaux de conformité dans la norme SQL-92: Entry, Intermediate, and Full;
- les fabricants de SGBD essaient de se conformer à cette norme, mais ils le font chacun à des niveaux différents de conformité;
- la version Oracle 7.2 est conforme au niveau Entry et contient quelques éléments des niveaux Intermediate et Full
- dans le cours, nous utilisons Oracle; il est possible que des requêtes SQL exécutant sur d'autres SGBD (comme Microsoft Access) ne puisse être exécutées sous Oracle, et vice-versa.

8.1 Langage de définition des données

8.1.1 Table

- une table est un ensemble de tuples;
- on utilise aussi *relation* comme synonyme de table, et *ligne* ou *enregistrement* comme synonymes de tuple;
- tous les tuples d'une table ont le même format; ce format est défini par un ensemble d'attributs;
- on peut représenter graphiquement une table par une matrice ou les colonnes sont les attributs; la première ligne comporte les noms des attributs, et les autres lignes représentent les tuples; l'ordre d'énumération des tuples ou des attributs n'a pas d'importance;

- la définition en SQL d'une table comporte les éléments suivants:
  - son nom
  - ses attributs
  - ses contraintes d'intégrité; il y a trois types de contraintes d'intégrité:
    * clé primaire;
    * clé unique;
    * clé étrangère.

8.1.2 Types en SQL

La norme ANSI SQL définit des types de données. Il appartient aux fabricant de bases de données (comme Oracle, IBM, Microsoft) de suivre cette norme. Le tableau ci-dessous donne les types ANSI SQL et les types spécifiques à Oracle. Par souci de portabilité, il est préférable d'utiliser les type ANSI supportés par Oracle; ils sont en caractères gras dans la colonne ANSI SQL. Postgres utilise les types ANSI SQL.

Type de données ANSI SQL	**Type correspondant spécifique à Oracle**
CHARACTER (n), **CHAR(n)**	CHAR(n)
NUMERIC (p, s), **DECIMAL (p, s)**, **DEC (p, s)**	NUMBER (p, s)
INTEGER, **INT**, **SMALLINT**	NUMBER (38)
FLOAT (p), **REAL**, **DOUBLE PRECISION**	NUMBER
CHARACTER VARYING(n), **VARCHAR(n)**	VARCHAR2(n)
DATE	DATE
TIME	DATE
TIMESTAMP	DATE

- CHAR(n)
  - représente une chaîne de caractères de longueur fixe n;
  - une chaîne de caractères est mise entre des apostrophes simples (');
  - pour inclure un ' dans une chaîne de caractères, on utilise deux '.

  Exemple 8.1 *La chaîne* `'abc12'` *est une valeur du type* `CHAR(5)`. *La chaîne* `'ab''12'` *contient un* ' *au milieu.*

- NUMERIC(p,s)
  - p indique le nombre total de chiffres stockés pour un nombre; la valeur de p doit être entre 1 et 38;
  - $s > 0$ indique le nombre total de chiffres après la virgule;
  - $s < 0$ indique un arrondissement du nombre de s chiffre avant la virgule;

Exemple 8.2

- `NUMERIC(5,2)` *peut contenir une valeur comportant 5 chiffres, dont 3 avant la virgule (soit 5-2) et 2 chiffres après la virgule; exemple de valeur : 123,45*
- `NUMERIC(2,5)` *peut contenir une valeur comportant de 2 chiffres; comme* $p < s$ *dans ce cas, on a seulement des chiffres* après *la virgule; les 3 premiers (soit 5-2) ont comme valeur 0; exemple de valeur : 0,00012*
- `NUMERIC(5,-2)` *peut contenir une valeur comportant de 7 chiffres avant la virgule (soit* $5 - -2$*), mais seulement les 5 premiers chiffres sont stockés, les 2 derniers sont toujours 0; il n'y aucun chiffre après la virgule; exemple de valeur : 1234500;lorsqu'on stocke une valeur dans la base de données, elle est toujours arrondie à deux chiffres* avant *la virgule; exemple 1234550 est stockée comme 1234600 et 1234549 est stocké comme 1234500.*
- `NUMERIC(2,-5)` *peut contenir une valeur comportant de 7 chiffres avant la virgule (soit* $2 - -5$*), mais seulement les 2 premiers chiffres sont stockés, les 5 derniers sont toujours 0; il n'y aucun chiffre après la virgule; exemple de valeur : 1200000;*

- REAL
  - permet de stocker un nombre en virgule flottant (c-a-d une valeur représentée par une mantisse et un exposant)
- VARCHAR(n)
  - permet de stocker une chaîne de caractères de longueur maximale n;
  - par rapport à CHAR(n), permet de mieux gérer l'espace disque si les chaînes de caractères ne sont pas toujours de longueur n;
- date et heure
  - le type DATE de la norme SQL2 comprend seulement une date en format YYYY-MM-DD;
  - le type TIME de la norme SQL2 comprend seulement une heure en format HH:MM:SS;
  - le type TIMESTAMP de la norme SQL2 comprend la date et l'heure en format (YYYY-MM-DD HH:MM:SS.ffffff), où ffffff représente une fraction de seconde;
  - le type DATE d'Oracle est presque équivalent au type TIMESTAMP de SQL2; il comprend la date et l'heure (YYYY-MM-DD HH:MM:SS), mais sans fraction de seconde; la valeur d'une date varie entre le 1^{er} janvier 4712 avant J-C et 31 décembre 4712 après J-C.
  - la date sous Oracle est affichée selon le format donné par le paramètre global NLS_DATE_FORMAT;
  - il existe plusieurs fonctions pour manipuler des dates en Oracle;

Exemple 8.3 `to_date('29-02-2000','DD-MM-YYYY')` *retourne la valeur de la date 29 février 2000 selon le type DATE d'Oracle.*

8.1.3 Définition des tables

Notation utilisée pour décrire la syntaxe du langage SQL

- `MOT_CLÉ` : mot réservé du langage SQL;
- «symbole terminal» : peut être remplacé par un identificateur ou une constante (nombre, chaîne de caractère, etc);
- «symbole non terminal» : doit être remplacé par sa définition;
- ::= : définition d'un symbole non terminal;
- "{" et "}" : équivalent des parenthèses en mathématiques;
- $^+$: une ou plusieurs occurrences;
- * : zéro ou plusieurs occurrences;
- [élément optionnel]
- | : choix entre plusieurs options

8.1.3.1 Syntaxe générale

«creation-table» ::=
  `CREATE TABLE` «nom-table» (
    «liste-attributs»
    [, «liste-contraintes»]
)

8.1.3.2 Définition des attributs

«liste-attributs» ::=
  «attributs» {, «attributs» }*

«attribut» ::=
  «nom-attribut» «type» [`DEFAULT` «expression»]
    [`NOT NULL`] [`CHECK` («condition»)]

- la valeur par défaut est utilisée si la valeur de l'attribut n'est pas spécifiée lors de la création d'un tuple;
- la condition doit être vérifiée lors de la création ou de la mise à jour d'un tuple;
- `NOT NULL` : la valeur de l'attribut ne peut contenir la valeur spéciale `NULL`;
- exemples de condition

- «nom-attribut» { = | > | >= | ... } «expression»
- «nom-attribut» `IN` («liste-valeurs»)
- «condition» { `AND` | `OR` } «condition»
- `NOT` «condition»
- plusieurs autres (voir manuel Oracle).

8.1.3.3 Définition des contraintes

«liste-contraintes» ::=
«contrainte» {, «contrainte» }*

«contrainte» ::=
«cle-primaire» | «cle-unique» | «cle-etrangere»

8.1.3.3.1 Clé primaire

«cle-primaire» ::=
`CONSTRAINT` «nom-contrainte» `PRIMARY KEY` («liste-noms-attribut»)

- il ne peut y avoir deux tuples avec les mêmes valeurs pour les attributs de la clé primaire;
- on peut définit une seule clé primaire pour une table;
- la valeur d'un attribut d'une clé primaire ne peut être `NULL` dans un tuple.

8.1.3.3.2 Clé unique

«cle-unique» ::=
`CONSTRAINT` «nom-contrainte» `UNIQUE` («liste-noms-attribut»)

- il ne peut y avoir deux tuples dans la table avec les mêmes valeurs pour les attributs de la clé unique;
- on peut définir *plusieurs* clés uniques pour une table;
- un attribut d'une clé unique peut être `NULL`, toutefois, la combinaison de tous les attributs non `NULL` doit être unique.

8.1.3.3.3 Clé étrangère

On dénote deux cas possibles:

1. faire référence à la **clé primaire** d'une autre table

   «cle-etrangere» ::=
   `CONSTRAINT` «nom-contrainte»
   `FOREIGN KEY` («liste-attributs»)
   `REFERENCES` «nom-table-referencee»
   [`ON DELETE CASCADE`]

   - les types de «liste-attributs» doivent être les mêmes que les types des attributs de la clé primaire de «nom-table-référencée»;
   - pour chaque tuple de la table dont les attributs de clé étrangère sont tous différents de `NULL`, il doit exister un tuple dans «nom-table-référencée» avec la même valeur pour «liste-attributs»;
   - `ON DELETE CASCADE` : si un tuple dans «nom-table-référencée» est supprimé, tous les tuples de la table qui le référence sont aussi supprimés.

2. faire référence à une **clé unique** d'une autre table

   «cle-unique» ::=
   `CONSTRAINT` «nom-contrainte»
   `FOREIGN KEY` («liste-attributs»)
   `REFERENCES` «nom-table-referencee»
   [(«liste-attributs-cle-unique»)]
   [`ON DELETE CASCADE`]

   - les types de «liste-attributs» doivent être les mêmes que les types «liste-attributs-clé-unique»;
   - pour chaque tuple de la table dont les attributs de clé étrangère sont tous différents de `NULL`, il doit exister un tuple dans «nom-table-référencée» avec la même valeur pour «liste-attributs»;
   - `ON DELETE CASCADE` : si un tuple dans «nom-table-référencée» est supprimé, tous les tuples de la table qui le référence sont aussi supprimés.

8.1.4 Modification des tables

`ALTER TABLE` «relation» {
«ajout-attribut» |
«modification-attribut» |
«suppression-attribut» |
«ajout-contrainte» |
«suppression-contrainte» }

8.1.4.1 Ajout d'attributs

«ajout-attribut» ::= `ADD` («liste-attributs»)

- ajoute les attributs de la liste à la table;

8.1.4.2 Modification d'attributs

«ajout-attribut» ::= `MODIFY` («liste-attributs»)

- modifie le type, la valeur par défaut ou l'option NULL or NOT NULL des attributs de la liste;
- on spécifie seulement les parties à modifier;
- pour modifier le type, la valeur de chaque attribut doit être `NULL` pour tous les tuples de la table;
- pour spécifier NOT NULL, il faut que l'attribut satisfasse déjà cette condition.

8.1.4.3 Suppression d'attributs

«ajout-attribut» ::= `DROP` («liste-noms-attribut»)

- supprime les attributs;
- non disponible en Oracle.

8.1.4.4 Ajout de contraintes

«ajout-attribut» ::= `ADD` («liste-contraintes»)

- ajoute les contraintes;
- les tuples de la table doivent satisfaire la contrainte.

8.1.4.5 Suppression de contraintes

«ajout-attribut» ::= `DROP` «nom-contrainte» [`CASCADE`]

- supprime la contrainte;
- CASCADE : supprime aussi toutes les contraintes qui dépendent de la contrainte supprimée.

8.1.5 Suppression des tables

`DROP TABLE` «nom-table» [CASCADE CONSTRAINTS]

8.1.5.1 L'exemple de la bibliothèque

Voici le schéma relationnel.

1. `editeur`

<u>idediteur</u>	nom	pays

2. `auteur`

<u>idauteur</u>	nom

3. `livre`

<u>idlivre</u>	titre	idauteur	idediteur	dateAcquisition	prix

4. `membre`

<u>idmembre</u>	nom	telephone	limitePret

5. `pret`

<u>idmembre</u>	<u>idlivre</u>	<u>datePret</u>

6. `reservation`

<u>idreservation</u>	idmembre	idlivre	dateReservation

Voici les énoncés de création des tables.

```
-----------------------------------------
-- une ligne de commentaire commence par deux '--'
-- Exemple de la bibliotheque
-- Marc Frappier, Universite de Sherbrooke
-- 2001-01-08
-----------------------------------------

DROP TABLE editeur CASCADE CONSTRAINTS;
CREATE TABLE editeur (
idEditeur       numeric(3) ,
nom             varchar(10) NOT NULL,
pays            varchar(10) NOT NULL,
CONSTRAINT cleEditeur PRIMARY KEY (idEditeur)
);
-- note: le point-virgule est requis par SQL/PLUS
-- il ne fait pas partie de la syntaxe de SQL

DROP TABLE auteur CASCADE CONSTRAINTS;
CREATE TABLE auteur (
idAuteur        numeric(3) ,
nom             varchar(10) NOT NULL,
CONSTRAINT cleAuteur PRIMARY KEY (idAuteur)
);

DROP TABLE livre CASCADE CONSTRAINTS;
CREATE TABLE livre (
idLivre         numeric(3) ,
titre           varchar(10) NOT NULL
                check(upper(substr(titre,1,1)) = substr(titre,1,1)),
idAuteur        numeric(3) NOT NULL ,
idEditeur       numeric(3) NOT NULL ,
dateAcquisition date ,
prix            numeric(7,2) ,
CONSTRAINT cleLivre PRIMARY KEY (idLivre) ,
CONSTRAINT cleCandidateTitreAuteur UNIQUE (titre,idAuteur) ,
CONSTRAINT cleCandidateTitreEditeur UNIQUE (titre,idEditeur) ,
CONSTRAINT refLivreAuteur FOREIGN KEY (idAuteur) REFERENCES auteur ,
CONSTRAINT refLivreEditeur FOREIGN KEY (idEditeur) REFERENCES editeur
);
```

```
DROP TABLE membre CASCADE CONSTRAINTS;
CREATE TABLE membre (
idMembre        numeric(3) ,
nom             varchar(10) NOT NULL ,
telephone       numeric(10) check(telephone >= 8190000000 and
                                  telephone <=8199999999) ,
limitePret      numeric(2) check(limitePret > 0 and limitePret <= 10) ,
CONSTRAINT cleMembre PRIMARY KEY (idMembre)
);

DROP TABLE pret CASCADE CONSTRAINTS;
CREATE TABLE pret (
idMembre        numeric(3) ,
idLivre         numeric(3) ,
datePret        date NOT NULL ,
CONSTRAINT clePret PRIMARY KEY (idMembre,idLivre,datePret) ,
CONSTRAINT refPretMembre FOREIGN KEY (idMembre) REFERENCES membre ,
CONSTRAINT refPretLivre FOREIGN KEY (idLivre) REFERENCES livre
);

DROP TABLE reservation CASCADE CONSTRAINTS;
CREATE TABLE reservation (
idReservation   numeric(3) ,
idMembre        numeric(3) ,
idLivre         numeric(3) ,
dateReservation date NOT NULL ,
CONSTRAINT cleReservation PRIMARY KEY (idReservation) ,
CONSTRAINT cleCandidateReservation UNIQUE (idMembre,idLivre) ,
CONSTRAINT refReservationMembre FOREIGN KEY (idMembre) REFERENCES membre ,
CONSTRAINT refReservationLivre FOREIGN KEY (idLivre) REFERENCES livre
);
```

8.2 Langage de manipulation des données

8.2.1 Insert

`INSERT INTO` «nom-table»
  [(«liste-noms-attribut»)]
  { `VALUES` («liste-expressions») | «select» }

8.2.2 Update

`UPDATE` «nom-table»
  `SET` { «liste-affectation» | «affectation-select» }
  [`WHERE` «condition»]

«liste-affectation» ::=
  «affectation» [, «affectation»*]

«affectation» ::=
  «nom-attribut» = «expression»

«affectation-select» ::=
  («liste-noms-attribut») = «select»

- «expression» peut être un énoncé select.

8.2.3 Delete

`DELETE FROM` «nom-table»
  [`WHERE` «condition»]

- si `WHERE` n'est pas spécifié, l'énoncé `DELETE` supprime tous les tuples.

8.2.4 Select

8.2.4.1 Syntaxe générale

«enonce-select-base» ::=
`SELECT` [`DISTINCT`] «liste-expressions-colonne»
`FROM` «liste-expressions-table»
[`WHERE` «condition-tuple»]
[`GROUP BY` «liste-expressions-colonne»]
[`HAVING` «condition-groupe»]
[`ORDER BY` «liste-expressions-colonne»]

«enonce-select-compose» ::=
«enonce-select-base»
{ `UNION` [`ALL`] | `INTERSECT` | `MINUS` }
«enonce-select-compose»

8.2.4.2 Sémantique

Le résultat d'un énoncé `SELECT` est égal au résultat des opérations suivantes. *Note: chaque SGBD utilise un algorithme propre pour exécuter un énoncé* `SELECT`*. Toutefois, le résultat est le même que celui donné par la procédure ci-dessous.*

1. évalue le produit cartésien des relations du FROM;
2. sélectionne les tuples satisfaisant la clause WHERE;
3. tuples regroupés selon la clause GROUP BY;
4. sélectionne les groupes selon la condition HAVING;
5. évalue les expressions du SELECT;
6. élimine les doublons si clause DISTINCT;
7. évalue l'union, l'intersection, ou la différence des selects (si nécessaire);
8. trie les tuples selon la clause ORDER BY.

8.2.4.3 Expressions

- expression élémentaire : nom d'attribut, fonction, constante;
- expression composée : opérateur appliqué sur expressions élémentaires ou composées;
- alias : renommer une expression ou une relation;
- *, R.* : retourne tous les attributs de la table
- expression appliquée à un subselect (clause WHERE)

- ⟨expr⟩ in (⟨select⟩) : vrai si ⟨expr⟩ est un élément de l'ensemble des tuples retournés par le select;
- ⟨expr⟩ not in (⟨select⟩) : vrai si ⟨expr⟩ n'appartient pas aux tuples retournés par le select;
- ⟨expr⟩ >any (⟨select⟩) : vrai s'il existe un tuple t ∈ ⟨select⟩ tel que ⟨expr⟩ > t;
- ⟨expr⟩ >all (⟨select⟩) : vrai si pour tous les tuple t ∈ ⟨select⟩, ⟨expr⟩ > t;
- exists (⟨select⟩) : vrai si l'ensemble des tuples du select est non vide;
- not exists (⟨select⟩) : vrai si l'ensemble des tuples du select est vide.

8.2.4.4 Fonctions d'agrégation

- appliquée à l'ensemble des tuples d'un select;
- `count(`«expr»`)`, `sum(`«expr»`)`, `avg(`«expr»`)`, `min(`«expr»`)`, max(«expr»), etc;
- `count(*)` : compte aussi les valeurs `NULL`;
- `count(`attribut`)` : compte seulement les valeurs non `NULL`;
- `count(distinct` attribut`)` : une valeur est comptée une seule fois, même si plusieurs tuples ont cette valeur;
- `GROUP BY` : fonction appliquée aux groupes, plutôt qu'à l'ensemble du select.

8.2.4.5 Opérations ensemblistes

1. `UNION` : union de tous les tuples des subselects avec élimination des doublons;
2. `UNION ALL` : union de tous les tuples des subselects sans élimination des doublons;
3. `INTERSECT` : intersection avec élimination doublon;
4. `MINUS` : différence, avec élimination doublon.

8.2.4.6 Quelques exemples

1. **Sélection de colonnes d'une table** : Afficher la liste des livres avec leur titre.

```
select  idlivre, titre
from    livre
```

2. **Sélection de lignes d'une table avec une condition élémentaire** : Afficher la liste des livres avec leur titre pour l'auteur idauteur = 3.

```
select  idlivre, titre
from    livre
where   idauteur = 3
```

3. **Sélection de lignes d'une table avec une condition composée** : Afficher la liste des livres avec leur titre pour les auteurs d'idauteur = 3 ou 5.

```
select  idlivre, titre
from    livre
where   idauteur = 3 or idauteur = 5
```

ou bien

```
select  idlivre, titre
from    livre
where   idauteur in (3,5)
```

4. **Ordonnancement du résultat** : Afficher la liste des livres avec leur titre pour les livres des auteurs d'idauteur = 3 ou 5, triée en ordre croissant de idauteur et titre.

```
select  idlivre, titre
from    livre
where   idauteur in (3,5)
order by idauteur, titre
```

5. **Spécification de colonnes calculées** : Afficher la liste des livres avec leur titre et le prix incluant la TPS et la TVQ, pour les livres des auteurs d'idauteur = 3 ou 5, triée en ordre croissant de idauteur et titre.

```
select  idlivre, titre, prix*1.075*1.07 PrixTTC
from    livre
where   idauteur in (3,5)
order by idauteur, titre
```

6. **Sélection de lignes à partir d'expressions** : Afficher la liste des livres avec leur titre et le prix incluant la TPS et la TVQ, pour les livres des auteurs d'idauteur = 3 ou 5 et dont le prix TTC est $\leq$ 100 $, triée en ordre croissant de idauteur et titre.

```
select  idlivre, titre, prix*1.075*1.07 PrixTTC
from    livre
where   idauteur in (3,5) and
        prix*1.075*1.07 <= 100
order by idauteur, titre
```

7. **Fonction d'agrégation** : Afficher le nombre d'éditeurs dans la base de données.

```
select  count(idediteur) "nb editeurs"
from    editeur
```

8. **Fonction d'agrégation avec élimination des doublons** : Afficher le nombre d'éditeurs et le nombre d'auteurs dans la base de données.

```
select  count(distinct t1.idediteur) "nb editeurs",
        count(distinct t2.idauteur) "nb auteurs"
from    editeur t1, auteur t2
```

9. **Jointure de plusieurs tables** : Afficher la liste des livres avec leur titre, nom de l'auteur et nom de l'éditeur, pour les idauteur = 3 ou 5, triée en ordre croissant de idauteur et titre

```
select  t1.idlivre, t1.titre, t2.nom, t3.nom
from    livre t1, auteur t2, editeur t3
where   t1.idauteur = t2.idauteur and
        t1.idediteur = t3.idediteur and
        t1.idauteur in (3,5)
order by t1.idauteur, t1.titre
```

10. **Calcul d'expressions de groupe** : Afficher la liste des éditeurs avec le nombre de livres édités.

```
select  t1.idediteur, t1.nom, count(*) "nb livres"
from    editeur t1, livre t2
where   t1.idediteur = t2.idediteur
group by t1.idediteur, t1.nom
order by t1.idediteur
```

    On note que si un éditeur n'a pas de livre édité, il n'apparaît pas dans le résultat du select.

11. **Sélection de groupes** : Afficher la liste des éditeurs avec le nombre de livres édités, en sélectionnant les éditeurs qui ont édité 5 livres ou plus.

```
select  t1.idediteur, t1.nom, count(t2.idlivre) "nb livres"
from    editeur t1, livre t2
where   t1.idediteur = t2.idediteur
group by t1.idediteur, t1.nom
having  count(t2.idlivre) >= 5
order by t1.idediteur
```

12. **Jointure externe (*outer join*)** : Afficher la liste des éditeurs avec le nombre de livres édités. Si un éditeur n'a aucun livre, afficher 0.

```
select  t1.idediteur, t1.nom, count(t2.idlivre) "nb livres"
from    editeur t1, livre t2
where   t1.idediteur = t2.idediteur (+)
group by t1.idediteur, t1.nom
order by t1.idediteur
```

    Le (+) permet de faire un (right) outer join entre la table `editeur` et la table `livre`.

13. **Opérateur any** : Afficher les auteurs qui ont au moins un éditeur en commun avec l'auteur d'idauteur 1.

```
select  distinct t1.nom, t1.idauteur
from    auteur t1, livre t2
where   t1.idauteur = t2.idauteur and
        t2.idediteur =any(
    select  t3.idediteur
    from    editeur t3, livre t4
    where   t3.idediteur = t4.idediteur and
            t4.idauteur = 1)
```

14. **Opérateur all** : Afficher le (ou les) livre(s) dont le prix est le plus élevé (les éléments maximaux) pour chaque éditeur.

```
select  t1.idediteur, t1.nom, t2.idlivre, t2.titre, t2.prix
from    editeur t1, livre t2
where   t1.idediteur = t2.idediteur and
        t2.prix >=all (
    select  t3.prix
    from    livre t3
    where   t3.idediteur = t2.idediteur)
order by t1.idediteur, t2.idlivre
```

    Si plusieurs livres ont le prix le plus élevé pour un éditeur donné, chacun est affiché.

15. **Opérateur all** Afficher le livre le plus cher (le supremum) de chaque éditeur, s'il existe.

```
select  t1.idediteur, t1.nom, t2.idlivre, t2.titre, t2.prix
from    editeur t1, livre t2
where   t1.idediteur = t2.idediteur and
        t2.prix >all (
    select  t3.prix
    from    livre t3
    where   t3.idediteur = t2.idediteur and
            t3.idlivre != t2.idlivre)
order by t1.idediteur, t2.idlivre
```

16. **Opérateur exists** : Afficher les auteurs qui ont publié au moins un livre avec l'éditeur 1.

```
select  t1.idauteur, t1.nom
from    auteur t1
where   exists (
    select  *
    from    livre t2
    where   t2.idauteur = t1.idauteur and
            t2.idediteur = 1)
```

L'énoncé ci-dessous est équivalent.

```
select  distinct t1.idauteur, t1.nom
from    auteur t1, livre t2
where   t1.idauteur = t2.idauteur and
        t2.idediteur = 1
```

17. **Opérateur not exists** : Afficher le (ou les) livre(s) dont le prix est le plus élevé (les éléments maximaux) pour chaque éditeur.

```
select  t1.idediteur, t1.nom, t2.idlivre, t2.titre, t2.prix
from    editeur t1, livre t2
where   t1.idediteur = t2.idediteur and
        not exists (
    select  *
    from    livre t3
    where   t3.idediteur = t2.idediteur and
            t2.prix < t3.prix)
order by t1.idediteur, t2.idlivre
```

Si plusieurs livres ont le prix le plus élevé pour un éditeur donné, chacun est affiché.

18. **Opérateur not exists** Afficher le livre le plus cher (le supremum) de chaque éditeur, s'il existe.

```
select  t1.idediteur, t1.nom, t2.idlivre, t2.titre, t2.prix
from    editeur t1, livre t2
where   t1.idediteur = t2.idediteur and
        not exists (
    select  *
    from    livre t3
    where   t3.idediteur = t2.idediteur and
            not (t3.idlivre = t2.idlivre) and
            t2.prix <= t3.prix)
order by t1.idediteur, t2.idlivre
```

19. **Requête de type 'pour tous' (Quantification universelle)** Afficher les auteurs qui ont publié un livre avec chaque éditeur.

```
select  t1.idauteur, t1.nom
from    auteur t1
where   not exists (
    select *
    from    editeur t2
    where   not exists (
        select *
        from    livre t3
        where   t3.idauteur = t1.idauteur and
                t3.idediteur = t2.idediteur))
```

20. **select imbriqué (clause from)** : Afficher le (ou les) livre(s) ayant le plus grand nombre de prêts pour chaque éditeur (éléments maximaux).

```
select  t3.idediteur, t3.nom, t3.idlivre, t3.titre, t3.nbpret
from    (
    select t1.idediteur, t1.nom, t2.idlivre, t2.titre,
           count(t5.idlivre) nbpret
    from editeur t1, livre t2, pret t5
    where t1.idediteur = t2.idediteur (+) and
          t2.idlivre = t5.idlivre (+)
    group by t1.idediteur, t1.nom, t2.idlivre, t2.titre) t3
where   not exists (
    select *
    from livre t4, pret t6
    where t4.idediteur = t3.idediteur and
          t4.idlivre = t6.idlivre (+)
    group by t4.idlivre
    having count(t6.idlivre) > t3.nbpret)
order by t3.idediteur
```

21. **Operation ensemblistes (union)** Afficher la liste de tous les auteurs et de tous les editeurs en indiquant leur type ('auteur' ou 'editeur').

```
select  t1.nom nom_a_e, 'auteur' type
from    auteur t1
union
(select  t2.nom nom_a_e, 'editeur' type
from    editeur t2)
order by nom_a_e
```

8.3 Divers

8.3.1 Table virtuelle : vue

- une vue est construite à partir d'un select;
- CREATE VIEW ⟨nom-vue⟩ AS
  ⟨select⟩ ;
- une vue peut-être utilisée dans un select; la vue est évaluée au moment où le FROM du select est évalué;
- elle peut-être utilisé dans un UPDATE si la vue réfère à une seule table, et qu'un tuple de la vue correspond à exactement un tuple de la table originale (clé (primaire ou unique) incluse);
- elle permet de restreindre l'accès aux tables (Chap 20).

8.3.2 Contraintes d'intégrité

- CREATE ASSERTION ⟨nom-contrainte⟩
  CHECK (⟨condition⟩)
- ⟨condition⟩ comme dans un WHERE d'un select
- pas disponible en Oracle
- Oracle

```
CREATE [OR REPLACE] TRIGGER <nom-trigger>
    {BEFORE|AFTER} {INSERT|DELETE|UPDATE} ON <nom-table>
    [FOR EACH ROW [WHEN (<condition>)]]
    <enonce PL/SQL>
```

8.3.3 Index

```
CREATE [ UNIQUE ] INDEX <nom-index>
ON TABLE <nom-table> (<liste-nom-attributs>)
```

8.3.4 Schéma

schéma : ensemble de tables, vues, domaines, séquences, procédures, synonymes, index, contraintes d'intégrité, et autres. CREATE SCHEMA ⟨nom-schéma⟩

catalogue (dictionnaire de données en Oracle): ensemble de schémas.

domaine : CREATE DOMAIN ⟨nom-du-domaine⟩ AS ⟨type⟩ [DEFAULT ⟨valeur⟩]; (pas disponible en Oracle)

8.3.5 La valeur spéciale NULL

- on utilise IS NULL pour tester si une expression est NULL;
- on utilise IS NOT NULL pour tester si une expression n'est pas NULL;
- une fonction évaluée avec une valeur NULL retourne une valeur NULL (sauf la fonction nvl(«expression»,«valeur») qui retourne «valeur» si expression est NULL, sinon elle retourne la valeur de «expression»);
- dans toutes les fonctions de groupe sauf count(*), la valeur spéciale NULL est ignorée;
- count(*) compte le nombre de tuple (doublons inclus), incluant les valeurs NULL;
- une expression booléenne atomique utilisant une valeur NULL retourne la valeur inconnu;
- une expression booléenne composée est définie selon les tables suivantes:

not	vrai	faux	inconnu
	faux	vrai	inconnu

and	vrai	faux	inconnu
vrai	vrai	faux	inconnu
faux	faux	faux	faux
inconnu	inconnu	faux	inconnu

or	vrai	faux	inconnu
vrai	vrai	vrai	vrai
faux	vrai	faux	inconnu
inconnu	vrai	inconnu	inconnu

- une clé primaire ne peut contenir de valeurs NULL pour les attributs d'un tuple;
- une clé unique peut contenir des valeurs NULL pour un tuple. Toutefois, l'unicité doit être préservée pour les valeurs non NULL.

8.3.6 Oracle et la norme SQL2

Quelques différences entre la syntaxe SQL d'Oracle et la syntaxe SQL de la norme SQL2.

Oracle	SQL2
from R1 u, R2 v	R1 as u, R2 as v
inexistant	CREATE DOMAIN
select * from R, S where R.B = S.C (+)	select * from R left outer join S on R.B = S.C
select * from R, S where R.B (+) = S.C	select * from R right outer join S on R.B = S.C
select * from R, S where R.B(+) = S.C(+)	select * from R full outer join S on R.B = S.C

Chapter 24

Le modèle client-serveur avec JDBC et JAVA

24.1 Brève introduction à Java

24.1.1 Exécution

- compilation : `javac pgm.java`;

  crée un fichier `pgm.class`

- exécution : `java pgm`

  - exécute la méthode `main` du fichier pgm.class
  - java est un langage interprété; le fichier pgm.class ne contient pas des instructions en langage d'assemblage; il contient plutôt des instructions de haut niveau (appelées Java byte code) qui sont interprétées par la machine virtuelle de Java (i.e., le programme `java`); cela permet d'exécuter un fichier `.class` sur n'importe quelle plateforme (Sun, Mac, PC) où une machine virtuelle de Java est disponible.

- un package est en ensemble de classes que l'on peut réutiliser à l'aide de l'énoncé `import`.

- JDK (Java Development Kit)

  - contient le compilateur, l'interpréteur et un ensemble de packages utilitaires (voir la documentation de Java sur le site web pour obtenir une description de ces packages).

24.1.2 Aperçu de la syntaxe

La syntaxe est similaire à C++, toutefois, on note les différences suivantes.

- il n'y a que des classes; toute fonction doit appartenir à une classe (i.e., elle doit être une méthode d'une classe);

- il n'y a pas de pointeur, mais il y a le concept de référence, qui est similaire;
- il y a 2 sortes de types en Java: types primitifs et types références;
  - types primitifs :

Type primitif	Valeurs	Taille
boolean	true ou false	1 bit
char	unicode	16 bits
byte	entier	8 bits
short	entier	16 bits
int	entier	32 bits
long	entier	64 bits
float	point flottant	32 bits
double	point flottant	64 bits

  - types références : ce sont les vecteurs et les classes; une variable de type référence est comme un pointeur avec indirection implicite.

24.1.3 Les classes

- déclaration d'une classe

```
class <NomClasse> {
<<declarations-variable-ou-methode>>
}
```

- déclaration d'une variable d'un type `<NomClasse>` (instance d'une classe)

```
NomClasse o;
```

Après l'exécution de cette déclaration, la valeur de la variable `o` est la *référence* `null`. Pour que `o` réfère à un objet, il faut l'allouer explicitement avec l'opérateur `new`. La déclaration

```
NomClasse o = new NomClasse(x,y,...);
```

déclare une variable `o` et l'initialise avec une référence vers un objet de la classe `NomClasse`. L'objet est initialisé par l'exécution de la méthode `NomClasse(x,y,...)`. Cette méthode est un constructeur.

- constructeur
  - l'opérateur `new` est appelé sur une méthode qui est un constructeur de classe.
  - le nom d'un constructeur est le même que le nom de la classe.

```
<Nom-Classe>(<type1> <param1>, ..., <typen> <paramn>)
{ <<corps-de-la-methode>> }
```

Dans l'exemple ci-dessous, on déclare un constructeur de la classe `Livre`.

```
Livre(Connection conn) { ... }
```

- déclaration d'une méthode

```
<type-de-retour> <nom-methode>(<type1> <param1>, ..., <typen> <paramn>)
{ <<corps-de-la-methode>> }
```

Dans l'exemple ci-dessous, on déclare une méthode de la classe `Livre`.

```
boolean existe(int idLivre)  { ... }
```

Si la méthode ne retourne pas de valeur, on utilise `void` comme type de retour.

- appel d'une méthode sur un objet

```
o.methode(x,y, ...)
```

- variable/méthode d'instance et variable/méthode de classe
  - le modificateur `static` indique qu'une variable (ou une méthode), est instanciée une seule fois, peu importe le nombre d'objets instanciés pour la classe. On l'appelle alors une variable de classe (ou une méthode de classe). Lorsque le modificateur `static` n'est pas spécifié, on l'appelle alors une variable d'instance (ou une méthode d'instance).

```
class C {
...
  static int v; // variable de classe
  int w; // variable d'instance
  static int m(Type1 v1, ..., Typen vn) { ... } // methode de classe
  int n(Type1 v1, ..., Typen vn) { ... } // methode d'instance
...
}
```

A l'intérieur de la classe `C`, on utilise directement une variable d'instance ou une variable de classe de la même manière, soit directement avec son nom (`v` ou `w`). À l'extérieur de la classe `C`, on utilise une variable de classe avec le nom de la classe en préfixe (`C.v`) et une variable d'instance avec le nom de l'objet en préfixe (`o.w`). Une méthode de classe ne peut être appelée sur un objet. À l'intérieur de la classe `C`, on l'appelle comme suit: `m(p1,...,pn)`. À l'extérieur de la classe `C`, on l'appelle comme suit: `C.m(p1,...,pn)`. Une méthode d'instance est appelée sur un objet. Un appel `o.n(p1,...,pn)` s'applique à l'objet `o`; dans le corps de la méthode `n`, l'objet `o` est accessible et dénoté par le mot-clé `this`. A l'intérieur de la classe `C`, l'appel `n(p1,...,pn)` s'applique sur l'objet `this`, c'est-à-dire qu'il est équivalent à `this.n(p1,...,pn)`.

- modificateur de déclaration de variable ou de méthode

  Il en existe 4 : `public`, `private`, `protected`, `package`.

- `private` : indique que la variable ou la méthode est accessible seulement à l'intérieur de la classe;

```
class Livre {
...
private boolean existe(int idLivre)  { ... }
...
}
```

  La méthode `existe` ne peut être appelée qu'à l'intérieur de la classe `Livre`.

- `public` : indique que la variable ou la méthode est accessible de n'importe quelle autre classe;

```
class Livre {
...
public boolean existe(int idLivre)  { ... }
...
}
```

  La méthode `existe` peut être appelée de n'importe quelle classe.

- passage de paramètres : par valeur seulement;

  Exemple. Considérons la méthode `swap` ci-dessous.

```
class toto {

    public static void swap(Object a, Object b) {

    Object temp = a;
    a = b;
    b = temp
    }
}


...
C x = new C();
C y = new C();
toto.swap(x,y);
```

  Désignons par o_1 l'objet référencé par la variable `x`, et o_2 l'objet référencé par la variable `y`. Après l'appel de `swap(x,y)`, la variable `x` réfère encore à o_1 et `y` à o_2. On peut expliquer ce résultat comme suit. Puisque le passage de paramètres est fait par valeur en Java, la variable `a` est *distincte* de la variable `x` : cela signifie que les modifications faites à la variable `a` n'influencent pas la variable `x`. Au début de l'exécution de `swap`, la

variable `a` contient une *copie* du contenu de la variable `x`. Donc, la variable `a` contient une *référence* à l'objet `o1`. L'exécution de `swap` modifie les variables `a` et `b`, mais elle ne modifie pas les variables `x` et `y` (pcq le passage de paramètre est fait par valeur). De plus, les objets référés par `x` et `y` ne sont pas modifés. Une affectation de la forme

```
a = b
```

change la valeur de la variable `a` (qui est une *référence*), et non pas l'objet *référé* par `a`. Donc, l'objet o_1 auquel `a` référait avant l'affectation n'a pas été modifié. Après l'exécution de cette affectation, `a` réfère au même objet que `b`, soit o_2.

- On utilise la méthode `clone()` pour créer une copie d'un objet. Cette méthode est héritée de la classe `Object` (toute classe en Java est une spécialisation de cette classe). On doit redéfinir cette méthode pour chaque classe que l'on crée (si nécessaire). La plupart des classes du Java Development Kit redéfinissent cette méthode.

  ```
  NomClasse o1 = new NomClasse();
  NomClasse o2 = o1.clone();
  ```

- Égalité d'objets: on peut vérifier l'égalité de deux objets référencés respectivement par les variables `o1` et `o2` avec la méthode `equals`. Comme la méthode `clone()`, la méthode `equals` est héritée de la classe `Object`. On doit redéfinir cette méthode pour chaque classe que l'on crée (si nécessaire). La plupart des classes du Java Development Kit redéfinisse cette méthode. L'expression `o1.equals(o2)` retourne vrai ssi l'objet référencé par `o1` a la même valeur que l'objet référencé par `o2`.

- Égalité de référence : on peut vérifier si deux variables réfèrent au même objet en utilisant `o1 == o2`. Notons que si `o1 == o2` retourne vrai, alors `o1.equals(o2)` retourne aussi vrai.

  Considérons l'exemple suivant utilisant la classe `Integer` (une classe qui permet de traiter un entier comme un objet).

  ```
  import java.io.*;
  public class egalite
  {
    public static void main(String argv[])
      {
      Integer a = new Integer(1);
      Integer b = new Integer(2);
      Integer c = new Integer(1);
      Integer d = a;
      System.out.println("a == b vaut " + (a == b));
      System.out.println("a.equals(b) vaut " + a.equals(b));
      System.out.println("a == c vaut " + (a == c));
      System.out.println("a.equals(c) vaut " + a.equals(c));
  ```

```
    System.out.println("a == d vaut " + (a == d));
    System.out.println("a.equals(d) vaut " + a.equals(d));
    }
}
```

Il imprime le résultat suivant:

```
a == b vaut false
a.equals(b) vaut false
a == c vaut false
a.equals(c) vaut true
a == d vaut true
a.equals(d) vaut true
```

Les variables `a` et `c` réfèrent à deux objets distincts (donc `a == c` retourne faux), mais qui sont égaux au niveau de leur contenu (donc `a.equals(c)` retourne vrai).

- gestion de la mémoire : il n'est pas nécessaire de désallouer les objets; l'interpréteur s'en charge (garbage collection).

24.1.4 Les interfaces

Une interface est un type, un peu comme une classe, sauf qu'elle ne peut être instanciée. C'est une liste de méthodes sans implémentation. Une classe peut implémenter une interface. Elle hérite doit alors implémenter toutes les méthodes déclarées dans l'interface. On peut déclarer une variable en donnant comme type un nom d'interface et l'instancier avec un objet d'une classe implémentant cette interface.

24.1.5 Les vecteurs

- C'est une classe particulière.
- Déclaration d'un vecteur à une dimension

  ```
  <type>[] a;
  ```

  La valeur de la variable `a` est la référence null.

- On peut déclarer des vecteurs à plusieurs dimensions. On utilise une paire de `[]` pour chaque dimension. Par exemple, voici un vecteur d'entiers à trois dimensions.

  ```
  int[][][] a;
  ```

- Allocation du vecteur : comme pour les objets, on utilise l'opérateur `new`, en spécifiant les dimensions. Dans l'exemple ci-dessous, on alloue un vecteur d'entiers à trois dimensions de $l \times m \times n$.

```
a = new int[l][m][n];
```

- Les composantes sont numérotées de 0 à $l-1$, $m-1$, $n-1$, etc. Dans l'exemple ci-dessous, on affecte la valeur 3 à une composante.

```
a[0][m-1][n-1] = 3;
```

- On peut obtenir la longueur d'un vecteur avec la variable `length` qui est définie pour chaque dimension d'un vecteur.

```
int[][][] a;
int l = 2;
int m = 3;
int n = 4;
a = new int[l][m][n];
System.out.println("dimension 1 = " + a.length);
System.out.println("dimension 2 = " + a[0].length);
System.out.println("dimension 3 = " + a[0][0].length);
```

L'exemple ci-dessus imprime le résultat suivant:

```
dimension 1 = 2
dimension 2 = 3
dimension 3 = 4
```

24.1.6 La classe String

- Elle permet de stocker une chaîne de caractères.
- Une string est un objet immutable; on ne peut modifier sa valeur; on doit utiliser une nouvelle string pour obtenir une modification; il faut utiliser la classe `StringBuffer` si on veut modifier une chaîne de caractères.
- déclaration et allocation : on peut allouer une string en utilisant une chaîne de caractères entourée de "

```
String a = "toto";
```

- on peut concaténer deux string avec l'opérateur `+`.

```
String a = "ift";
String b = "286";
String c = a + b;
```

La variable `c` réfère à la chaîne `"ift286"`.

Note: Par souci de simplicité et par abus de langage, nous ne faisons plus de distinction, dans la suite de ce chapitre, entre une variable de type référence et l'objet référencé par la variable . Nous utiliserons l'expression "l'objet `o`" au lieu de l'expression plus longue "l'objet référencé par `o`".

Soit un membre m déclaré dans une class A					
	accès au membre m à partir de				
	classe A	classe B			
		B et A même package		B et A package différent	
visibilité		B sous-classe de A	B n'est pas sous-classe de A	B sous-classe de A	B n'est pas sous-classe de A
private	O	N	N	N	N
protected	O	O	O	O	N
public	O	O	O	O	O
package	O	O	O	N	N

Tableau 24.1: Description des modificateurs d'accès

24.1.7 Visibilité des déclarations

L'accès aux membres (i.e., les méthodes et les attributs) d'une classe peut être restreint, afin de masquer les détails de l'implémentation d'une classe et de limiter les dépendances entre les classes. Trois modificateurs d'accès peuvent être utilisés: `public`, `protected` et `private`. Si aucune modificateur est spécifié, l'accès par défaut est appelé *package*.

public un membre m déclaré comment étant `public` est accessible de n'importe quelle autre classe.

protected un membre m déclaré comment étant `protected` dans une classe C d'un package P est accessible seulement par les autres classes de P et par une sous-classe C' de C, peu importe le package de C'.

private un membre m déclaré comment étant `private` dans une classe C est accessible seulement de C.

valeur par défaut (*package*) Si aucun modificateur d'accès n'est spécifié, le membre est accessible par toutes les classes du package.

Bien entendu, un membre m est toujours accessible dans la classe où il est déclaré, peu importe le modificateur d'accès utilisé. Le tableau 24.1 résume ces contraintes d'accès.

24.1.8 Exception

Le langage Java utilise les `Exception` pour gérer les erreurs d'exécution. Si une méthode m ou une opération ne peut réaliser le traitement prévu, elle peut le signaler à la méthode appelante m' (i.e., m' a appelé m) en lui envoyant une exception (i.e., m envoie une exception à m'). Une exception est un objet d'une classe. Il existe toute une hiérarchie d'exceptions en Java. La figure 24.1 représente cette hiérarchie des classes d'exception. La classe `java.lang.Throwable` est la plus générale; elle a deux sous-classes: i) `java.lang.Error`, qui est utilisée par la machine virtuelle java pour les erreurs sévères de bas niveau dans la machine virtuelle; ii)

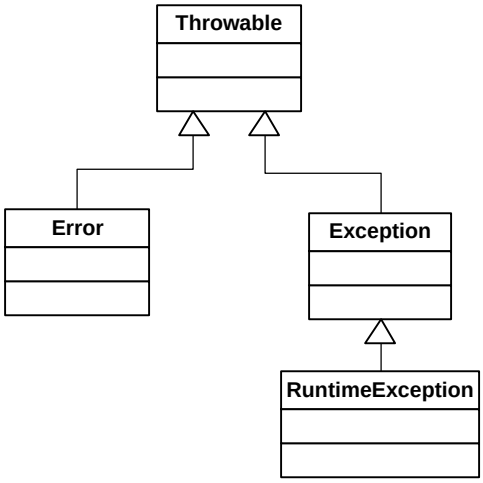

Figure 24.1: Hiérarchie des classes d'exception

`java.lang.Exception`, qui est utilisée par les applications pour les erreurs plus communes. Si une méthode m lève une exception de type `java.lang.Exception`, cette exception doit être déclarée dans l'entête de la méthode, sauf si cette exception est de type `java.lang.-RuntimeException`.

Une méthode lève (envoie) une exception e à l'aide de l'instruction `throw`. Une méthode peut traiter une exception envoyée par une méthode appelée en utilisant l'instruction `try-catch`.

```
try { ... }
catch (TypeException e1) { ... }
...
catch (TypeException en) { ... }
```

La clause `try` est d'abord exécutée. Si une exception est levée durant l'exécution du corps de `try`, cette exception est comparée aux types d'exception des clauses `catch`. La première clause dont le type d'exception satisfait celui de l'exception levée est exécutée. Si aucune clause `catch` n'est satisfaite, l'exécution du `try-catch` termine anormalement et reçue est envoyée à l'appelant. On peut ajouter une clause `finally` après le dernier `catch`.

```
try { ... }
catch (TypeException e1) { ... }
...
catch (TypeException en) { ... }
finally  {...}
```

La clause finally est toujours exécutée, quelque soit le résultat du `try` ou du `catch`. Si l'exécution d'un `catch` ou d'un `finally` lève une exception, alors cette exception est retournée à l'appelant. Si les deux lèvent une exception, celle du `finally` est retournée à l'appelant.

La figure 24.2 donne un exemple d'usage des exceptions. La méthode `division` lève une exception si elle est appelée avec une valeur nulle ou négative pour `x`. Dans ce cas, l'exécution

```
 1:package test1;
 2:
 3:public class ExempleException {
 4:
 5:  static int division(int x, int y) throws Exception
 6:  {
 7:    if (x <= 0)
 8:      throw new Exception("La valeur de x doit etre superieure a zero.");
 9:    else return y / x;
10:  }
11:
12:  static int m1(int x, int y)
13:  {
14:    try {
15:      return division(x,y);
16:    }
17:    catch (Exception e) {
18:      return 0;
19:    }
20:    finally {
21:      System.out.println("La methode m1 termine");
22:    }
23:  }
24:
25:  static int m2(int x, int y) throws Exception
26:  {
27:    int z = division(x,y);
28:    System.out.println("La methode m2 termine");
29:    return z;
30:  }
31:
32:  public static void main(String[] args) throws Exception
33:  {
34:    System.out.println("Resultat de m1 avec 5, 10 = " + m1(5,10));
35:    System.out.println("Resultat de m1 avec 0, 10 = " + m1(0,10));
36:    System.out.println("Resultat de m2 avec 5, 10 = " + m2(5,10));
37:    System.out.println("Resultat de m2 avec 0, 10 = " + m2(0,10));
38:  }
39:}
```

Figure 24.2: Un exemple de traitement des exceptions.

```
La methode m1 termine
Resultat de m1 avec 5, 10 = 2
La methode m1 termine
Resultat de m1 avec 0, 10 = 0
La methode m2 termine
Resultat de m2 avec 5, 10 = 2
java.lang.Exception: La valeur de x doit etre superieure a zero.
  at test1.ExempleException.division(ExempleException.java:8)
  at test1.ExempleException.m2(ExempleException.java:27)
  at test1.ExempleException.main(ExempleException.java:37)
Exception in thread "main"
```

Figure 24.3: Résultat de l'exécution de la méthode `main` de la figure 24.2

de `division` se termine immédiatement après l'exécution de l'instruction `throw`. On remarque que la clause `throws` est utilisée dans l'entête de la méthode `division`, pour indiquer au compilateur que `division` peut lever une exception de type `Exception`. Le compilateur Java vérifie que toute méthode qui lève une exception de type autre que `RuntimeException` déclare cette exception dans son entête. La méthode `m1` appelle `division`; si la valeur de `x` n'est pas strictement positive, alors l'appel de `division` retourne une exception. Comme l'appel de `division` est effectué au sein d'un `try ...catch ...`, alors la clause `catch` s'exécute; s'il n'y a pas d'exception, la clause `try` se termine normalement; la clause `finally` s'exécute toujours, après le `try` ou le `catch`, même si la clause `try` ou la clause `catch` contient un `return`. Dans cet exemple, un appel à `m1` entraîne toujours l'impression du message "La méthode m1 termine". Notez que la clause `finally` peut être omise; on obtient alors énoncé `try ... catch ...`.

On remarque que la méthode `m2` appelle directement `division` sans utiliser un `try ... catch ... finally ...` . Lorsque `division` lève une exception, la méthode `m2` termine anormalement immédiatement après l'appel de `division` et retourne l'exception reçue à sa méthode appelante. Par exemple, si on exécute la méthode `main` de la figure 24.2, on obtient le résultat illustré à la figure 24.3. On note que le dernier appel à `division` lève une exception qui n'est pas traitée par `m2`. Cette exception est envoyée par `m2` à `main`, qui la renvoie à son tour à la machine virtuelle Java, qui termine alors anormalement et imprime l'exception avec toute l'information qu'elle contient; entre autres, elle imprime la pile d'appels des méthodes d'où provient l'exception, avec les numéros de ligne des appels dans les méthodes appelantes. On note aussi que l'entête de `m2` contient la clause `throws`, car l'appel à `division` peut lever une exception. Comme cette exception n'est pas captée par `m2` avec un `try-catch-finally`, la méthode `m2` peut donc aussi lever une exception. On remarque aussi que la clause `throws` n'apparaît pas dans l'entête de `m1`, car l'exception est captée par un `try-catch-finally`.

24.2 Brève introduction au diagramme de classe d'UML

La notation UML comporte plusieurs diagrammes. Lors de la remise de travaux, nous n'utiliserons que le diagramme de classe pour documenter la conception d'une application.

Ces notes de cours utilisent aussi les diagrammes de séquences.

La figure 24.4 (page 36) illustre les symboles du diagramme de classe qui seront utilisés. La figure 24.5 (page 37) donne le diagramme de classe correspondant au code Java ci-dessous. Les classes et les interfaces sont représentées par des rectangles. Les attributs et les méthodes y sont décrits, avec leur visibilité (voir tableau 24.1 page 29), nom, paramètres et type de retour. Les déclarations des variables de classes et les méthodes de classes sont soulignées. Les constantes sont soulignées et annotées avec l'expression `{frozen}`. Les liens entre les classes sont représentés par des lignes de différents types. Le lien de *dépendance* entre la classe `A1` et la classe `StringBuffer` indique que la classe `A1` utilise la classe `StringBuffer` dans une de ses méthodes (la méthode `m`). Notez que cela *n'indique pas* que `A1` a un attribut de type `StringBuffer`; nous utiliserons un autre type de lien pour cela, les associations. Le lien de spécialisation entre la classe `A2` et la classe `A1` dénote la déclaration Java `class A2 extends A1`. Un lien d'association entre deux classes indique qu'une classe a un attribut (typiquement un attribut d'instance) dont le type est donné par l'autre classe; le nom de cet attribut est indiqué par le rôle. Si l'association est bidirectionnelle, les classes se réfèrent mutuellement via deux attributs dont les valeurs sont dépendantes. Par exemple, l'association `R1` entre `A1` et `B` est représentée par l'attribut `lesA1` de la classe `b` et l'attribut `unB` de la classe `A1`. Les multiplicité (x..y) indiquent à combien d'objet un objet d'une classe est associé. Les valeurs des attributs doivent être cohérentes : si un objet a_1 de `A1` réfère à un objet b_1 de `B` via l'attribut `unB`, alors la liste `lesA1` de b_1 doit contenir une référence à a_1, et vice-versa. Si l'association est unidirectionnelle, seule une classe réfère à l'autre (voir l'association `R2`). Notons qu'un role dénote un attribut; il n'est donc pas nécessaire d'inclure cet attribut dans la liste des attributs donnée dans le rectangle de la classe, car cela est redondant et entraîne de la confusion.

```
public class A1 {

  /* variables de classe */

  private static int unEntier = 0;
  public final static double PI = 3.1415;

  /* variables d'instances */

  public char[] unVecteur;
  boolean unBooleen;
  protected String uneString;
  private B unB;

  /* Constructeur */
  public A1(boolean pUnBooleen) {
    unB = null;
    uneString = "toto";
    unBooleen = pUnBooleen;
    unVecteur = new char[2];
```

```
    unVecteur[0] = 'a';
    unVecteur[1] = 'B';
    unEntier = unEntier + 1;
  }

  public StringBuffer m() {
    StringBuffer sb = new StringBuffer();
    sb.append("Bonjour ");
    sb.append("Bienvenu à IFT287 ");
    sb.append("Bonne session");
    return sb;
  }

  public void setUnB(B pUnB)
  {
  unB = pUnB;
  pUnB.addUnA1(this);
  }
}

public class A2 extends A1 {

  private B[] lesB;

  public A2()
  {
    super(true);
    lesB = new B[2];
  }

  public void setLesB(B b1, B b2)
  {
    lesB[0] = b1;
    lesB[1] = b2;
  }
}


import java.util.*;
public class B {

  private List<A1> lesA1;

  public B()
  {
```

```
    lesA1 = new LinkedList<A1>();
  }

  public void addUnA1(A1 pUnA1)
  {
    lesA1.add(pUnA1);
  }
}
```

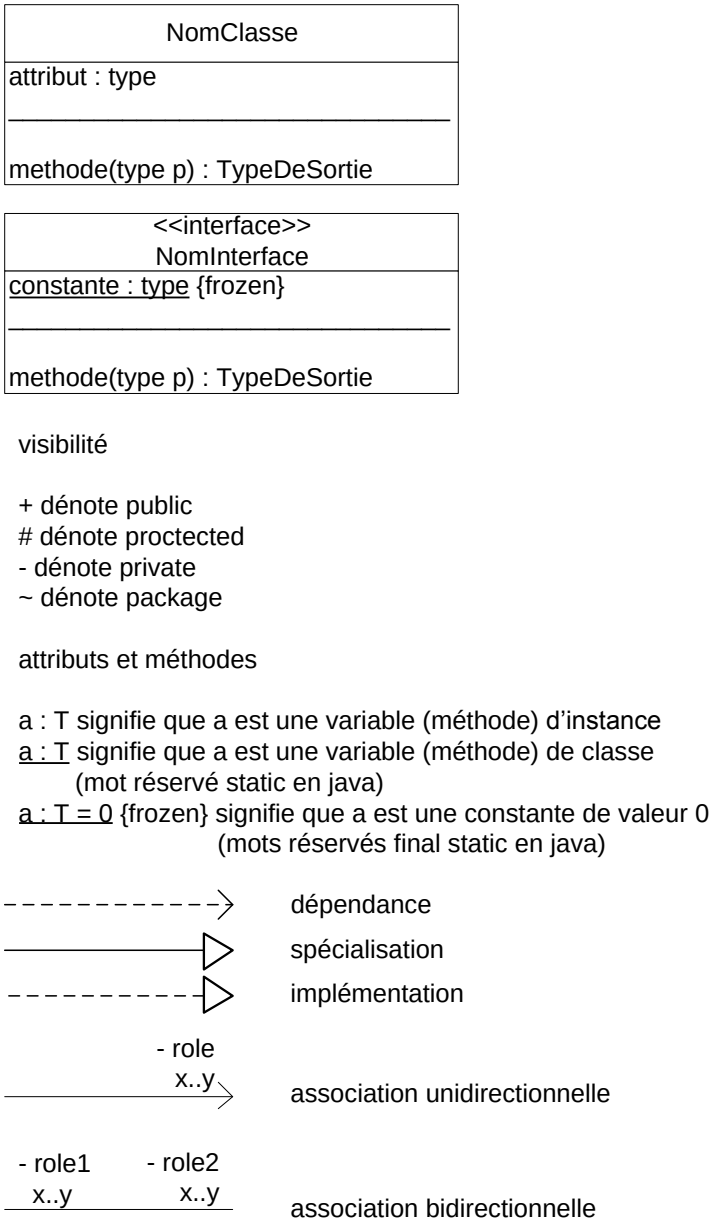

Figure 24.4: Symboles utilisés pour les diagrammes de classe

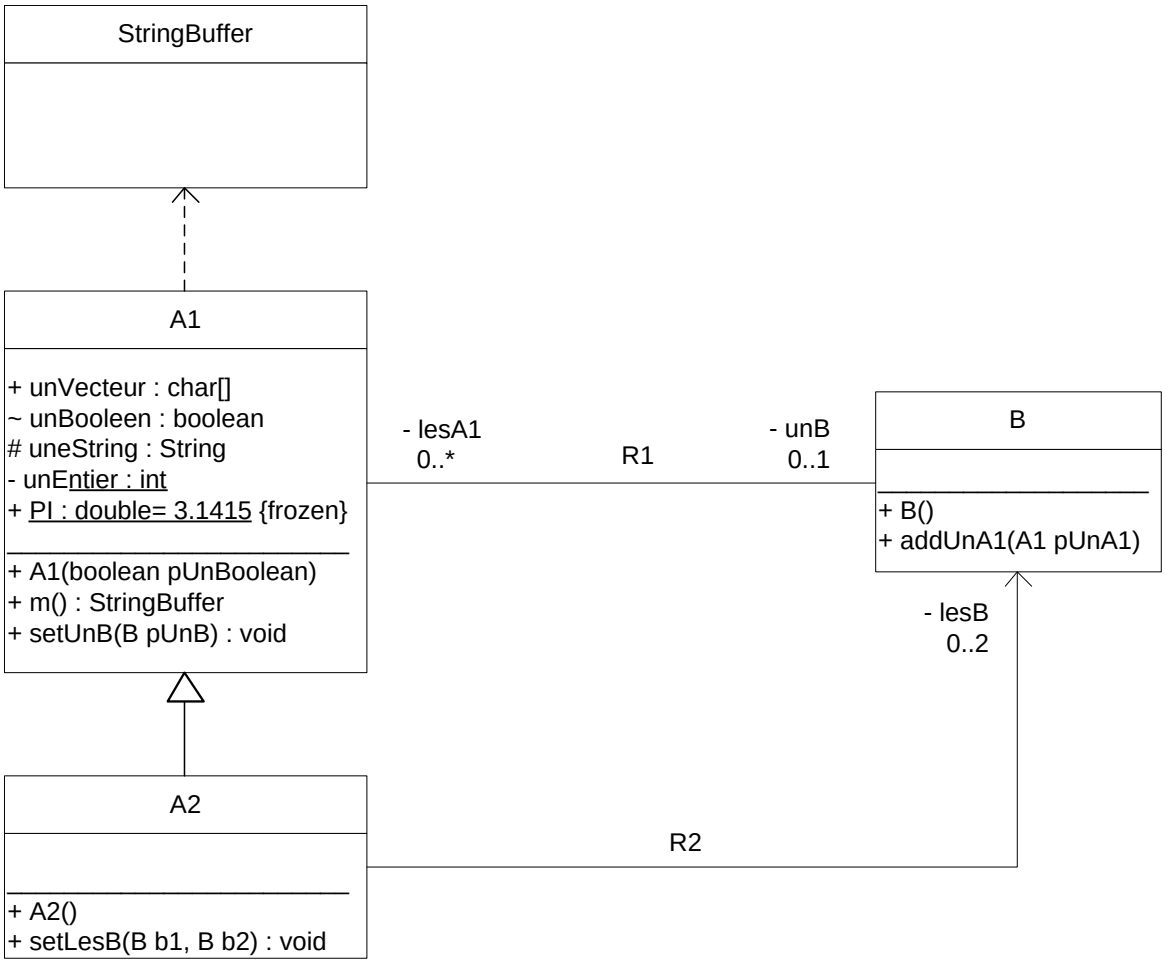

Figure 24.5: Diagramme de classe correspondant au code Java de la section 24.2

24.3 JDBC

- JDBC : Java Database Connectivity
- JDBC est un package (ensemble de classes) écrit en Java pour soumettre des énoncés SQL à un serveur de BD.
- Portabilité
  - Chaque fabricant doit implanter le package JDBC pour sa base de données.
  - JDBC est disponible pour plusieurs BD (Oracle, DB2, Postgres, mySQL, Sybase, etc.).
  - Les applications utilisant JDBC sont indépendantes du serveur SQL. Il suffit de modifier quelques lignes pour changer de serveur SQL (par exemple, passer d'un serveur Oracle à un serveur DB2). Toutefois, la portabilité est limitée par les différences syntaxiques entre les différents dialectes du langage SQL.

24.3.1 Étapes pour soumettre des commandes SQL en Java

1. Importer les classes de JDBC (java.sql)

   ```
   import java.sql.*;
   ```

2. Charger le pilote de JDBC pour le serveur SQL

   - pour Oracle

     ```
     DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
     ```

     ou bien, de manière plus générale,

     ```
     Driver d = (Driver)
     Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
     DriverManager.registerDriver(d);
     ```

     Cet énoncé charge l'implémentation de JDBC par Oracle et l'enregistre. L'implémentation de JDBC pour Oracle est donnée par la classe `oracle.jdbc.driver.OracleDriver()`.

   - pour Postgres

     ```
     Driver d = (Driver)
     Class.forName("org.Postgresql.Driver").newInstance();
     DriverManager.registerDriver(d);
     ```

3. Établir une connexion avec le serveur SQL

- Pour Oracle

```
Connection conn = DriverManager.getConnection(
    "jdbc:oracle:thin:@oracleacad.usherbrooke.ca:1521:orcl",
    user, pass);
```

  La chaîne de connexion `jdbc:oracle:thin:@oracleacad.usherbrooke.ca:1521:orcl` comprend les parties suivantes.

  - `jdbc:oracle:thin` dénote le pilote JDBC léger (il en existe un autre plus spécifique à Oracle).
  - `oracleacad.usherbrooke.ca` dénote l'adresse internet du serveur SQL Oracle au Département d'informatique. Cette adresse n'est pas accessible de l'extérieur du campus, pour des raisons de sécurité. Pour se connecter de l'extérieur du campus, il faut utiliser une connexion RPV (réseau privé virtuel).
  - `1521` dénote le port.
  - `orcl` dénote le nom de la base de données.

  Les autres informations se décrivent comme suit.

  - `Connection` est une interface qui permet de stocker une connexion avec le serveur SQL.
  - `conn` est un objet que l'on utilise par la suite pour créer des énoncés SQL à soumettre au serveur.
  - `user` est une string qui dénote le nom d'utilisateur sur le serveur SQL;
  - `pass` est une string qui dénote le mot de passe du user id sur le serveur SQL.

- Pour Postgres

```
Connection conn = DriverManager.getConnection(
    "jdbc:Postgresql:Postgres", user, pass);
```

4. Soumettre les commandes SQL (à voir dans les sections suivantes)

5. Fermer la connexion

```
conn.close();
```

24.3.2 Soumettre les commandes SQL

- un objet `Connection` permet de créer des objets qui peuvent exécuter des commandes SQL

- 3 classes de commandes

  - `Statement` : commande sans paramètres.

- `PreparedStatement` : commande compilée avec paramètres; plus efficace si on doit exécuter plusieurs fois la même commande.
- `CallableStatement` : commande pour exécuter une procédure en PL/SQL, avec paramètres.

24.3.3 Méthodes de `Statement`

- pour créer un objet :

```
Statement stmt = conn.createStatement ();}
```

  - déclaration d'un objet avec initialisation;
  - `Statement` : nom de la classe;
  - `stmt` : nom de l'objet (instance);
  - `conn.createStatement ()` : objet `conn` de la classe `Connection` auquel on applique la méthode `createStatement ()`.

- pour exécuter un SELECT :

```
ResultSet rset = stmt.executeQuery
    ("select idlivre, titre from livre");
```

  - `ResultSet` : Classe qui permet de contenir le résultat d'un SELECT;
  - `rset` : objet de la classe `ResultSet`; contient le résultat du SELECT;
  - `stmt.executeQuery(<enonce-select>)` : objet `stmt` auquel on applique la méthode `executeQuery` pour soumettre la requête `<enonce-select>` au serveur SQL.

- pour extraire les tuples du résultat

  - `rset.next()` : permet d'avancer au tuple suivant (lire) dans `rset`; il faut faire un `next` pour avancer vers le premier tuple du résultat. La méthode `next` retourne vrai si elle a lu un tuple, faux sinon.
  - `rset.getXXXX(<nomDeColonne>)` : retourne la colonne de nom `<nomDeColonne>` du tuple courant; la valeur retournée est de type XXXX; les méthodes suivantes sont disponibles

Type SQL	Description	Méthode de JDBC	Type Java
varchar	chaîne longueur variable	getString	String
char	chaîne longueur fixe	getString	String
numeric(p,s)	point flottant avec échelle	getDouble ou getFloat	double ou float
integer	entier	getInt ou getLong	int ou long
date	date (-4712 à + 4712)	getDate	java.sql.Date
time	heure HH:MM:SS.pppppp	getTime	java.sql.Time
timestamp	date et heure	getTimestamp	java.sql.Timestamp

– `rset.getXXXX`(i) : retourne la $i^{ième}$ colonne du tuple courant; les méthodes disponibles sont similaires à `rset.getXXXX(<nomDeColonne>)`; cette approche par no de colonne est plus délicate à utiliser, car si on change la requête SQL, les colonnes peuvent être décalées par rapport à la version précédente; il faut alors modifier les appels de méthode pour ajuster le no de colonne.

- Pour savoir si la valeur lue est NULL, on utilise la méthode `wasNull()` de `ResultSet`.
- Une valeur NULL d'un attribut est convertie par les méthodes `getXXXX` comme suit.

Type SQL	valeur Java retournée
nombre	0
date	null
chaîne de caractères	null

- un `ResultSet` est automatiquement fermé dans les cas suivants:
  - la fermeture de l'objet `Statement` qui l'a créé;
  - la ré-exécution de l'objet `Statement` qui l'a créé.

  Donc, si on veut manipuler deux objets de type `ResultSet` en même temps, il faut aussi utiliser un objet de type `Statement` pour chacun.
- pour exécuter une mise à jour (insert, update, delete, create table, alter table, etc)

  ```
  int nb = stmt.executeUpdate("create table ...");
  ```

  La méthode retourne le nombre de tuples insérés, mis à jour, ou modifiés, dans le cas d'un insert, update ou delete, respectivement.
- fermer la commande : `stmt.close()`

Exemple: Voir le programme `Exemple.java` sur la page web du cours.

24.3.4 Méthodes de `PreparedStatement`

Un Prepared statement contient des `?` qui peuvent être ensuite initialisés avec des méthodes `setXXXX`(i, *valeur*) où XXXX est le type de la variable qui contient *valeur*, i est le $i^{ième}$ paramètre de la commande et *valeur* est la valeur à affecter au paramètre.

- création de l'énoncé

  ```
  PreparedStatement stmt = conn.prepareStatement(
  "insert into editeur (idediteur, nom, ville, pays) values " +
  "(?,?,?,?)")
  ```

- affectation de valeurs aux paramètres

```
stmt.setInt(1,idediteur);
stmt.setString(2,nom);
stmt.setString(3,ville);
stmt.setString(4,pays);
```

Pour mettre une valeur NULL, on utilise `setNull`(i,$sqlType$), où $sqlType$ est un type SQL standard (CHAR, VARCHAR, INTEGER, etc)

- exécution de l'énoncé

```
int nb = stmt.executeUpdate();
```

Pour un select, on exécute

```
Resultset rset = stmt.executeQuery();
```

Exemple: Voir le programme `Livre.java` sur la page web du cours.

24.4 Conception d'une application en architecture trois niveaux

Dans cette section, nous étudions une architecture typique pour l'implémentation d'une application de type système d'information interagissant avec un utilisateur et une base de données. Cette architecture est du type "trois niveaux", souvent appelée « *three-tier architecture* » en anglais. La figure 24.6 illustre cette architecture. Le niveau présentation s'occupe de formatter l'information pour la présenter à l'utilisateur. Cette couche s'adapte au type d'interface: caractères, client web, téléphone intelligent, etc. Le niveau affaires (aussi appelé métier) s'occupe de traiter l'information en appliquant les règles d'affaires; il interroge une source de données, dont il ne connait pas la représentation (BD OO, BD relationnelle, fichier XML, séquentiel, etc) et produit l'information sous une forme abstraite (liste, tableau, ensemble, map, etc) pour la couche présentation. Le niveau données interroge une source de données et s'adapte à la forme de cette source. Les trois niveaux sont indépendants, de sorte qu'on peut modifier l'un sans avoir à changer l'autre. Par exemple, on peut créer deux types d'interfaces pour une même application, page web classique et téléphone portable, sans avoir à modifier les deux autres niveaux.

24.4.1 Architecture

On peut diviser un système d'information en 4 types de composantes:

1. **gestionnaire des interactions avec l'utilisateur**:
   c'est le module principal de contrôle; il s'occupe d'initialiser l'environnement, de lire les transactions et de les répartir aux classes appropriées.

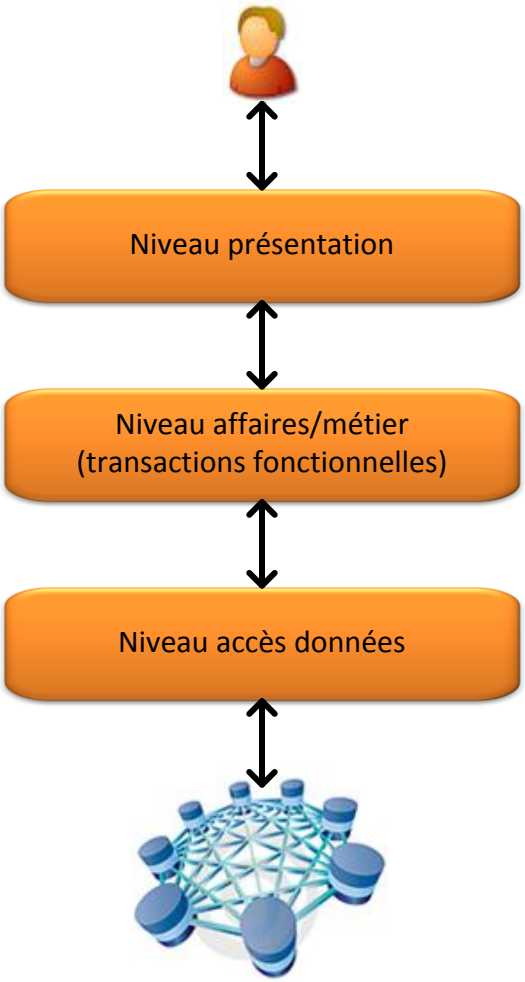

Figure 24.6: Architecture trois niveaux (*three-tier architecture*)

2. **gestionnaire de connexions**:
   il s'occupe d'établir une connexion avec le serveur SQL et d'initialiser certains paramètres.

3. **gestionnaire de transactions**:
   il exécute un sous-ensemble de transactions qui sont fonctionnellement reliées; on regroupe typiquement les transactions d'un système en sous-ensembles de transactions qui font référence au même processus dans le domaine d'application et on définit un gestionnaire pour chaque sous-ensemble. Par exemple, pour le système de gestion de la bibliothèque, on retrouve 4 gestionnaires:

   - le gestionnaire des membres, qui s'occupe des transactions `inscrire` et `désinscrire`;
   - le gestionnaire des livres, qui s'occupe des transactions `acquerir` et `vendre`;
   - le gestionnaire des prêts, qui s'occupe des transactions `preter`, `renouveler` et `retourner`;
   - le gestionnaire des réservations, qui s'occupe des transactions `reserver`, `prendreRes` et `annulerRes`.

   Ce découpage est un peu arbitraire, au sens où d'autres découpages seraient aussi acceptables (par exemple, regrouper les gestionnaires des livres et des prêts en une seule classe). On estime toutefois qu'il est plus facile de maintenir des classes qui ont un haut degré de *cohésion fonctionnelle*.

4. **gestionnaire de table**:
   il s'occupe de faire toutes les requêtes SQL pour une table donnée; il y a donc typiquement un gestionnaire par table de la base de données.

La figure 24.7 illustre les classes du système de gestion de bibliothèque. Le gestionnaire de transactions y est omis, ainsi que certaines associations, pour éviter de surcharger inutilement le diagramme. La figure 24.8 illustre le modèle relationnel de la BD de ce système, alors que la figure 24.9 illustre le modèle conceptuel (entité-relation avec UML).

24.4.2 Le gestionnaire des interactions avec l'utilisateur

C'est la classe principale du système. Elle effectue les traitements suivants :

- appel du gestionnaire de connexion pour l'ouverture d'une connexion;
- allocation des gestionnaires de table;
- allocation des gestionnaires de transactions;
- lecture des transactions et répartitions aux gestionnaires de transactions;
- appel du gestionnaire de connexion pour la fermeture de la connexion.

Exemple: voir `Biblio.java` sur la page web du cours.

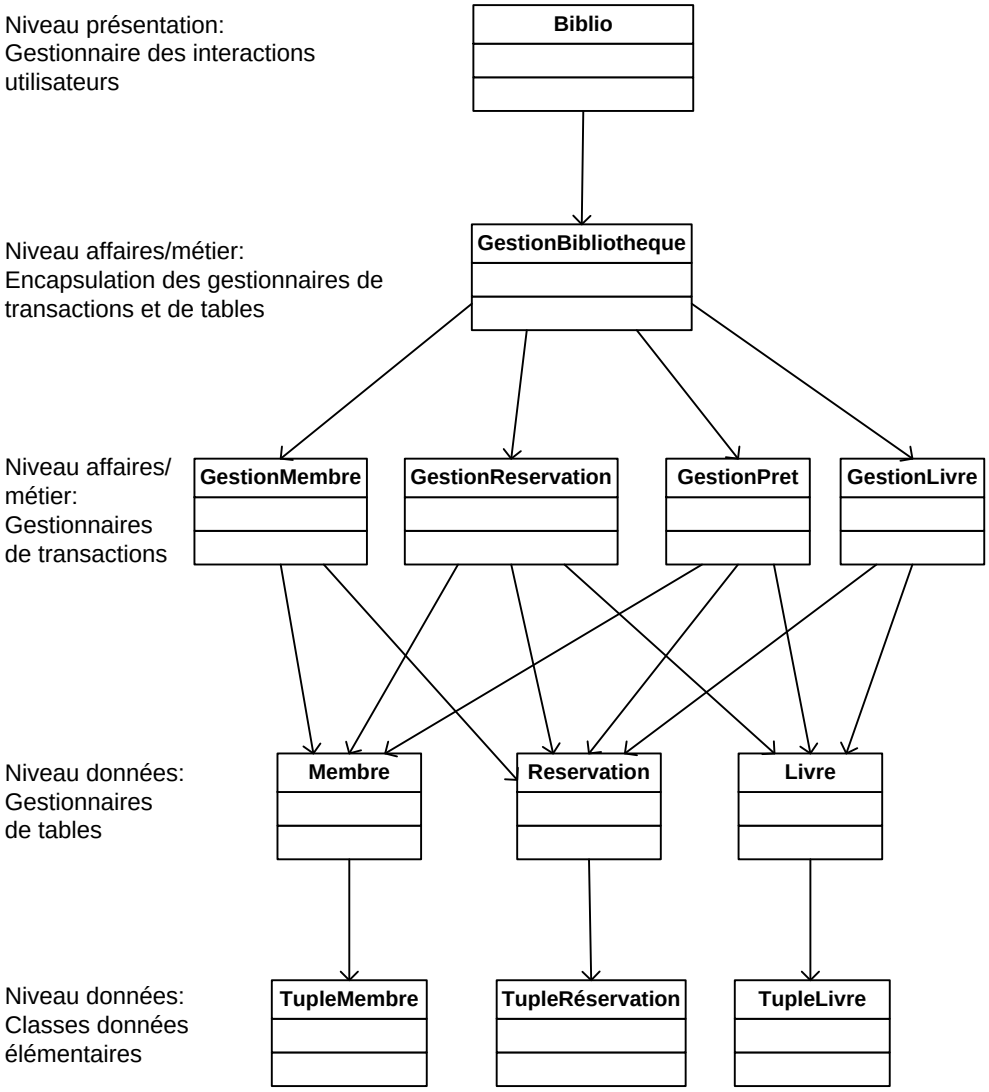

Figure 24.7: Diagramme de classe pour le système de gestion de bibliothèque

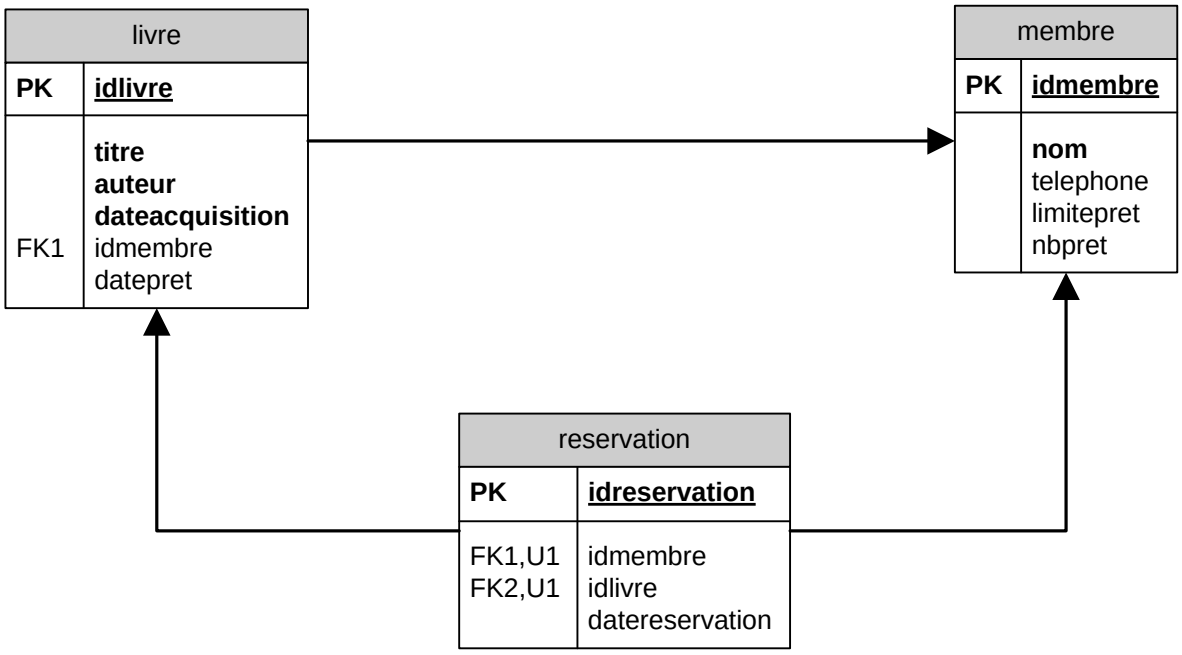

Figure 24.8: Modèle relationel de la BD du système de gestion de bibliothèque

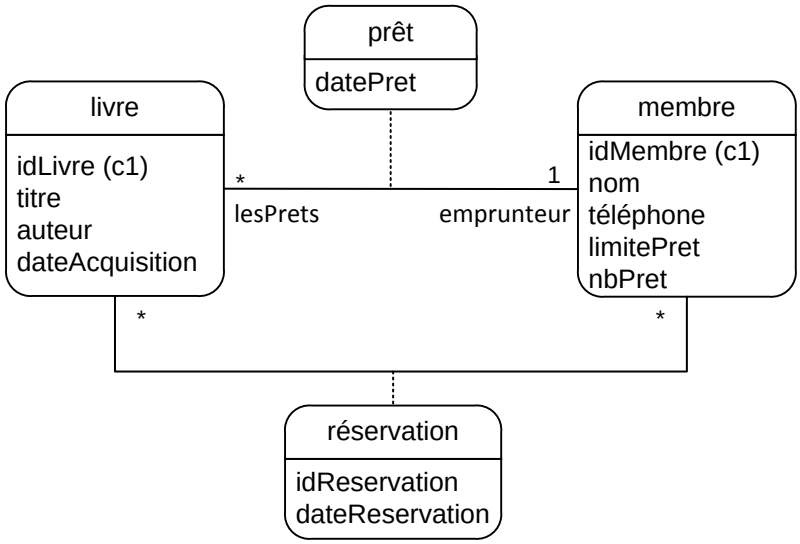

Figure 24.9: Modèle conceptuel de données du système de gestion de bibliothèque

24.4.3 Le gestionnaire des connexions

- charge l'implémentation de JDBC pour le serveur SQL choisi;
- ouvre une connexion avec le serveur SQL;
- initialise la connexion en mode `autocommit false`;
- initialise le niveau d'isolement des transactions au niveau approprié; le niveau d'isolement détermine les interférences (conflits) admissibles entre les transactions;
  - il est recommandé d'utiliser le niveau `Connection.TRANSACTION_SERIALIZABLE`, ce qui assure le plus haut degré d'intégrité disponible pour l'accès en concurrence à la base de données; toutefois, ce niveau est aussi le moins efficace en terme de performance;
  - avec Oracle, le niveau par défaut est `Connection.TRANSACTION_READ_COMMITTED`; à ce niveau, les interférences suivantes sont possibles: un tuple lu peut être modifié par une autre transaction après la lecture, avant que la transaction se termine, ce qui peut engendrer une corruption de la base de données. Par exemple, considérons les deux transactions suivantes qui enregistrent un prêt du même livre à deux membres différents dans la base de données. Considérons le cas où ces transactions sont exécutées en concurrence, sur deux postes de travail différents.
    * T1 = `preter 1 1 2000-09-29`
    * T2 = `preter 1 2 2000-09-29`

    On utilise les abréviations suivantes pour dénoter les opérations effectuées sur la base de données.
    * $r(l)$: lecture du livre l
    * $u(l)$: mise-à-jour du livre l
    * $r(m)$: lecture du membre m
    * $u(m)$: mise-à-jour du membre m

    Le tableau ci-dessous illustre un scénario possible de l'ordre d'exécution des opérations sur la base de données.

Ordre	T1	T2
1	$r(l_1)$	
2	$r(m_1)$	
3		$r(l_1)$
4		$r(m_2)$
5	$u(l_1)$	
6	$u(m_1)$	
7		$u(l_1)$
8		$u(m_2)$

    La transaction T1 lit le livre l_1 et le membre m_1 pour vérifier si l'emprunt est valide. Supposons que l'emprunt est valide (le livre est disponible et le membre

n'a pas atteint sa limite de prêts). Ensuite, la transaction T2 s'exécute; elle lit le livre l_1 et le membre m_2 pour vérifier si l'emprunt est valide. Supposons qu'il est valide. Ensuite, la transaction T1 continue son exécution: le livre et le membre sont mis à jour pour enregistrer l'emprunt. Ensuite, la transaction T2 continue son exécution: le livre et le membre sont mis à jour pour enregistrer l'emprunt. La mise à jour de T2 sur le livre l_1 écrase celle effectuée par T1. L'attribut idmembre du livre l_1 réfère au membre m_2. Le nombre de prêts du membre m_1 a été incrémenté de 1, même si le prêt n'est pas enregistré à son nom, ce qui correspond à une corruption de l'état de la base de données. Ce type d'interférence ne peut survenir en mode `Connection.TRANSACTION_SERIALIZABLE`, mais il peut survenir en mode `Connection.TRANSACTION_READ_COMMITTED`.

Un niveau d'isolement permet d'éviter certaines interférences.

- *Dirty reads* : Une transaction T1 lit un enregistrement R maj par une transaction T2 qui n'est pas encore terminée. Si T2 fait un rollback, sa mise-à–jour sur r est effacée, mais T1 ne sera pas alerté de cette annulation des modifications à r. T1 continuera son exécution en ayant lu une valeur erronée de r, donc il peut produire une incohérence en utilisant une valeur qui n'a jamais existé dans la BD.

- *Non-repeatable reads* :

  Un enregistrement r lu par une transaction T1 est ensuite modifié et validé par une transaction T2 avant la fin de T1. Si T1 re-lit r, il verra la maj faite par T2.

- *Phantom reads* : Une transaction T1 fait un SELECT avec une condition C. Durant l'exécution, l'exécution d'un SELECT avec la même condition C retourne un ensemble différent, qui peut comprendre de nouveau enregistrements.

Le tableau ci-dessous illustre les caractéristiques de chaque niveau d'isolement du standard SQL, où un "X" signifie que le problème peut survenir.

Niveau d'isolement	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	X	X	X
Read Committed		X	X
Repeatable Read			X
Serializable			

En mode read committed, une transaction ne voit que les mises à jour validées (committed) par les autres transactions. Elle voit aussi ses propres mises à jour, bien sûr.

Une transaction exécutant en mode sérialisable ne voit pas les mises à jour des autres transactions effectuées après le début de la transaction. La transaction s?éxécute comme si la BD n'était pas modifiée par les autres transactions. Si un enregistrement est modifié par une autre transaction, et que la transaction en mode sérialisable essaie ensuite de modifier cet enregistrement, elle reçoit une erreur.

Une transaction débute avec le premier énoncé SQL depuis le démarrage de la session. Elle termine avec un commit ou rollback. Une nouvelle transaction débute après. Les

transactions de sessions différentes se chevauche, d?où la possibilité d?interférence dans la mise à jour des données.

Voici quelques exemples de détection des interférences selon le mode d'isolation.

Read Committed : No dirty read

Session 1	Session 2	
Mode sérialisable et autocommit false	Mode readcommitted et autocommit false;	
	\set AUTOCOMMIT off	
\set AUTOCOMMIT off SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL SERIALIZABLE; drop table t cascade; commit; create table t (A numeric(3) primary key, B numeric(3)); commit; INSERT INTO T (A,B) VALUES (1,1); select * from t where a=1;		
a | b ---+--- 1 | 1 (1 row)		
	select * from t;	
	a | b ---+--- (0 rows)	L'insertion de la session 1 n'est pas visible par session 2, vu que pas encore de commit dans la session 1
	.	
Commit;		
	select * from t;	
	a | b ---+--- 1 | 1 (1 row)	L'insertion de la session 1 est maintenant visible par session 2, vu que qu'il y a eu commit dans la session 1. (c'est l'essentiel du mode readcommitted). Toutefois, on voit que le résultat du même SELECT varie durant la transaction. Ce n'est pas le cas en mode serializable.

Détection des interférences en mode sérialisable : update

<table>
<tr><td colspan="3">select * from t where a=1;</td><td></td></tr>
<tr><td colspan="3">a | b
---+---
1 | 1
(1 row)</td><td></td></tr>
<tr><td colspan="3"></td><td>update t set b=0 where a=1;
UPDATE 1</td></tr>
<tr><td colspan="3"></td><td>select * from t where a=1;</td></tr>
<tr><td colspan="3"></td><td>a | b
---+---
1 | 0
(1 row)</td></tr>
<tr><td colspan="3">select * from t where a=1;</td><td></td></tr>
<tr><td>a | b
---+---
1 | 1
(1 row)</td><td colspan="2">Comme en mode readcommitted, session 1 ne voit pas les maj de session 2.</td><td></td></tr>
<tr><td colspan="2">update t set b=b+1 where a=1;</td><td>La commande attend que session 2 fasse un commit ou un rollback avant de terminer, vu que session 2 a mis à jour ce tuple.</td><td></td></tr>
<tr><td colspan="3"></td><td>Commit;</td></tr>
<tr><td colspan="3">ERROR: could not serialize access due to concurrent update</td><td></td></tr>
<tr><td colspan="3">Rollback;</td><td></td></tr>
</table>

Absence de détection des interférences en mode readcommitted

	select * from t where a=1;	
	a | b ---+--- 1 | 0 (1 row)	
select * from t where a=1;		
a | b ---+--- 1 | 0 (1 row)		
update t set b=1; UPDATE 1		
	select * from t where a=1;	
	a | b ---+--- 1 | 0 (1 row)	
	update t set b=b+1;	exécution en attente d'un commit dans session 1
Commit;		
	UPDATE 1	Exécution du update s'effectue suite au commit de session 1.
	select * from t where a=1;	
	a | b ---+--- 1 | 2 (1 row)	Cette transaction voyait b=0, mais l'exécution de b=b+1 donne 2, car elle a attendu le commit de session 1 pour procéder à la maj.
	Commit;	

Détection des interférences en mode sérialisable : delete

select * from t where a=1;		
a | b ---+--- 1 | 2 (1 row)		
		Delete from t where a=1; DELETE 1
update t set b=1;	exécution en attente d'un commit dans session 2	
		Commit;
ERROR: could not serialize access due to concurrent update		
Rollback;		

Mode sérialisable : faiblesses!

		INSERT INTO T (A,B) VALUES (1,1); Commit;	
select * from t where a=1;			
a | b ---+--- 1 | 0 (1 row)			
		INSERT INTO T (A,B) VALUES (2,1); INSERT 1	
delete from t; DELETE 1			
select * from t; a | b ---+--- (0 rows)	Cette transaction croit avoir supprimé tous les tuples, mais ce n'est pas le cas. Elle ne voit pas le tuple a=2		
		select * from t; a | b ---+--- 1 | 0 2 | 1 (2 rows)	Cette transaction ne voit pas que tuple a=1 n'existe plus.
		Commit;	
		select * from t; a | b ---+--- 1 | 0 2 | 1 (2 rows)	Cette transaction ne voit toujours pas que tuple a=1 n'existe plus.
Commit;			
select * from t; a | b ---+--- 2 | 1 (1 row)	Le commit dans sesson 1 fait en sorte que session 1 peut voir le tuple a=2 inséré par session 2		
		Commit;	
		select * from t; a | b ---+--- 2 | 1 (1 row)	Suite au commit de session 1, session 2 voit que le tuple a=1 a été supprimé

24.4.4 Les gestionnaires de table

- Lors de la création d'une instance de gestionnaire de table, on crée les instances de `PreparedStatement` pour les requêtes à effectuer.
- On définit typiquement une méthode pour chaque transaction.
- Une méthode effectue les requêtes nécessaires et retourne, si désiré, le résultat.
- Une méthode d'interrogation peut retourner un booléen, un tuple ou imprimer directement le résultat.
- Une méthode de mise à jour peut retourner le nombre de tuples mis à jour par la requête.

Exemple: voir `Livre.java` et `Membre.java` sur la page web du cours.

24.4.5 Les gestionnaires de transactions

24.4.5.1 Création d'une instance

- le constructeur reçoit en paramètre une référence à un gestionnaire pour chaque table utilisée par les transactions;
- il doit sauvegarder chaque référence pour pouvoir faire les accès à la base de données;
- il doit s'assurer que les gestionnaires de tables utilisent tous la même connexion, afin que les transactions soient correctement validées (commit) ou annulées (rollback);
- il doit sauvegarder la référence à la connexion pour pouvoir faire les validations ou les annulations de transactions.

24.4.5.2 Traitement d'une transaction

En général, on définit une méthode pour chaque transaction. Une transaction comporte les étapes suivantes:

- validation des paramètres de la transaction; les services des gestionnaires de table sont utilisés pour accéder à la base de données;
- mise à jour des tables en utilisant les services des gestionnaires de table;
- pour détecter les erreurs, on exécute ces opérations à l'intérieur d'un `try`; si une opération lève une exception, la clause `catch` s'exécute.

```
try {
    validation;
    mise a jour;
    conn.commit;
```

```
    }
catch (Exception e)
    {
    conn.rollback();
    throw new BiblioException
        (msg);
    }
```

- La clause `try` se termine par un appel de la méthode `commit` sur la connexion utilisée. Cela rend les mises à jour permanentes et visibles pour les autres utilisateurs de la base de données.

- La clause `catch (Exception e)` est exécutée si l'exception est une instance de la classe `Exception`. Il peut y avoir plusieurs clauses `catch`, si on désire effectuer des traitements différents pour chaque classe d'exceptions. Typiquement, on annule les mise à jour de la transaction en appelant la méthode `rollback` sur la connexion utilisée. Ensuite, on lève une exception qui sera reçue par la méthode appelante.

- En mode sérialisable, il **faut toujours faire un** `commit` ou un `rollback`, même si la transaction n'effectue aucune mise à jour. Cela permet de libérer les verrous engendrés par les lectures (mécanisme de contrôle interne utilisé par le SGBD pour implémenter le mode sérialisable).

La figure 24.10 (page 56) illustre les principales interactions entre les objets du système pour exécuter une transaction `preter`. Une flèche représente soit une transaction (entre l'utilisateur et le système) ou soit un appel de méthode.

Exemple: voir `GestionLivre.java`, `GestionMembre.java` et `GestionPret.java` sur la page web du cours.

24.5 Gestion des dates et heures en Java et SQL

La plupart des application de bases de données doivent gérer du temps sous forme de date et heure. Cette gestion peut se faire en SQL ou en Java.

24.5.1 SQL

Les dates sont stockées à l'interne selon le fuseau horaire de Greenwich (Angleterre) et affichées selon le fuseau horaire de la session en cours en effectuant les conversions appropriées. Le fuseau horaire est initialisé par défaut au fuseau horaire du SGBD. Les opérateurs de manipulation de date tiennent donc compte des fuseaux horaires, des années bissextiles et des heures avancées (heure d'hiver, heure d'été). Tous ces paramètres sont modifiés sous Oracle avec la commande `ALTER SESSION SET NLS_`*paramètre* sous Oracle. La date et heure courante est appelée `sysdate` sous Oracle et `current_date`, `current_time`, et `current_timestamp` sous Postgres.

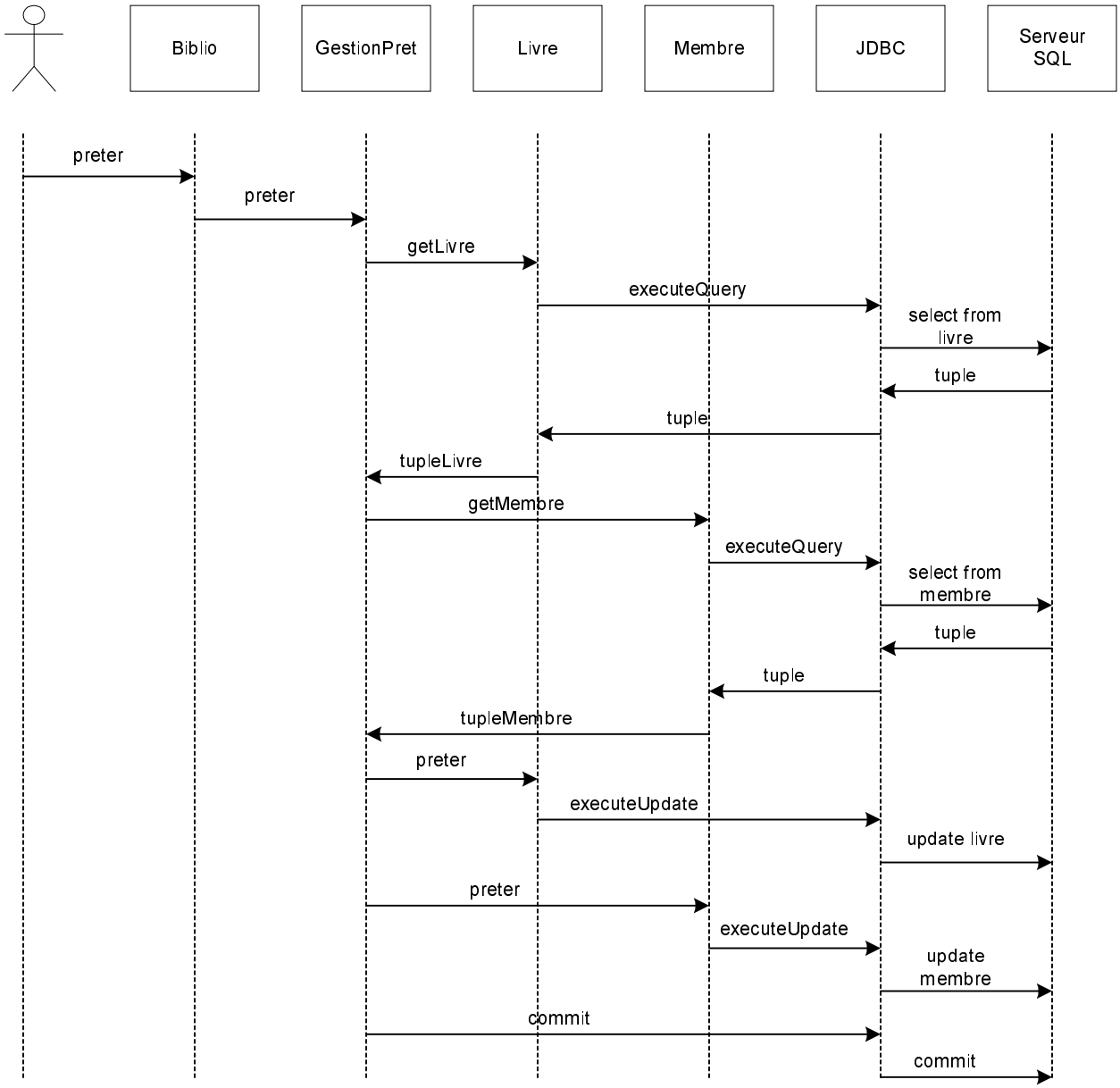

Figure 24.10: Diagramme de séquence pour une transaction `preter`

Avec Oracle et Postgres, les opérateurs + et - peuvent être appliquées à des dates. Soit $date_1$ et $date_2$ des expressions de type DATE; soit *jours* un nombre réel représentant un nombre de jours (donc, la partie fractionnaire représente une fraction de journée).

1. $date_1 - date_2$: retourne le nombre de jours (nombre réel dont la partie fractionnaire représente une fraction de jour) entre $date_1$ et $date_2$;

   Exemple 24.1 `to_date('01-01-2000 18:54:19','DD-MM-YYYY HH24:MI:SS') - to_date('31-12-1999','DD-MM-YYYY')` *retourne 1.78771991 jours.*

2. $date_1$ + *jours* : retourne la date obtenue en ajoutant *jours* à $date_1$;

   Exemple 24.2 `to_date('31-12-1999','DD-MM-YYYY') + 1.7` *retourne la date 2000-01-01 16:48:00.*

3. $date_1$ - *jours* : retourne la date obtenue en retranchant *jours* à $date_1$.

   Exemple 24.3 `to_date('31-12-1999','DD-MM-YYYY') - 1.7` *retourne la date 1999-12-29 07:12:00.*

Une des faiblesses de cette approche est la portabilité. Ces opérateurs ne sont pas disponibles dans tous les SGBD. Une autre approche consiste à utiliser des classes de Java.

24.5.2 Java

Le JDK propose plusieurs classes très puissantes pour manipuler des dates.

1. La classe `java.util.Date` permet de représenter un instant dans le temps, c'est-à-dire une date avec heures, minutes et secondes. Entre autres, elle permet d'instancier une date avec la date et l'heure courante, et de faire des comparaison (<,>,=) entre deux dates. Une date est représentée par le nombre de milisecondes écoulées depuis le 1er janvier 1970 à 0h00 GMT (heure de Greenwich en Angleterre).

2. La classe `java.util.GregorianCalendar` permet de faire des manipulations de dates selon le calendrier grégorien. Entre autres, elle permet d'ajouter des années, des mois, des jours, des heures, des minutes et des secondes à un objet calendar. Elle permet aussi de comparer deux calendar (<,>,=).

3. La classe `java.text.SimpleDateFormat` permet de convertir une Date en une String, et vice-versa, selon le format voulu. Une instance est associée à un fuseau horaire (par défaut, le fuseau horaire de l'horloge de l'ordinateur). Lors de la conversion d'une String en Date, c'est ce fuseau horaire qui est utilisé.

4. Le package `java.sql` contient plusieurs classes pour représenter les types SQL standard comme DATE (`java.sql.Date`), TIME (`java.sql.Time`) et TIMESTAMP (`java.sql.TimeStamp`). Entre autres, elles permettent de convertir une String en un objet `java.sql.Date`, `java.sql.Time` et `java.sql.TimeStamp`.

Pour plus d'informations sur ces classes, voir la classe
`http://www.dmi.usherb.ca/ frappier/ift287/TestDate.java`
et la documentation du JDK.

Chapter 11

Concepts de bases de données objet

11.1 Concepts de BD OO

- objectifs des BD OO
  - rendre les objets persistants, c'est-à-dire que les objets persistent (sont stockés dans un fichier) après la fin de l'exécution d'un programme
  - partager des objets entre plusieurs programmes qui s'exécutent en parallèle
  - lire des objets de manière transparente (persistance transitive)
  - écrire des objets de manière transparente à l'aide du concept de transaction, qui permet aussi de préserver la cohérence des objets

  persistance des objets d'un programme
- exemples : FastObjects, Object Store, etc.
- JDO (Java Data Objects): norme pour la définition de SGBD orienté objet; remplace l'ancienne norme de l'ODMG (Object Data Management Group)
- tout objet peut être rendu persistent; la structure des objets peut-être arbitrairement complexe
- classe : similaire à un schéma relationnel; la norme JDO associe à chaque classe un ensemble qui permet de stocker toutes les objets d'une classe (« *extent* »).
- supporte l'héritage; un « *extent* » peut contenir aussi les objets des sous-classes de la classe associée
- supporte les relations entre les objets (peristance transitive
- chaque objet est identifié de manière unique par un identificateur d'objet;
- l'identificateur d'objet est une valeur interne au SGBD OO; elle n'est pas accessible à l'utilisateur de la BD.

11.2 Structure

1. Puisque tout objet peut être rendu persistant, on peut définir n'importe quelle structure de données. En comparaison, les BD relationnelles n'utilisent que des tuples dont les composantes sont des types primitifs. Les SGDBs relationnels étendus offrent quelques structures additionnelles (par exemple, tables imbriquées, stockage d'un objet dans un tuple), elles n'atteignent pas le degré de flexibilité offert par les SGBD OO.
2. On utilisera beaucoup les structures suivantes, car elles sont très générales et très utiles:
   - ensemble (« *set* »)
   - multi-ensemble (« *bag* » ou « *collection* ») : similaire à un ensemble, mais permet plusieurs occurrences du même objet;
   - listes (« *list* ») : ensemble d'éléments qui sont ordonnés
   - fonction (« *map* ») : ensemble de couples clé-valeur

11.3 Persistance

Il y a deux sortes d'objets: transitoire et persistant.

- objet transitoire : c'est un objet qui n'existe qu'en mémoire vive; il disparaît à la fin de l'exécution du programme.
- objet persistant :
  - c'est un objet qui est stocké sur disque et qui peut être chargé en mémoire vive.
  - Il est créé durant l'exécution d'un programme et sauvegardé avant la fin de l'exécution du programme.
  - Pour être stocké, l'objet doit avoir un **nom persistant**, ou bien il doit être accessible à partir d'un objet ayant un nom persistant.

Le concept de **persistance transitive** permet de stocker les objets persistants sur disque. Lorsqu'un objet ayant un nom persistant est sauvegardé, les objets qu'il référence sont aussi sauvegardés si leur classe est décrite comme étant admissible à la persistance (« *persistence capable* » en anglais); aussi appelé persistance par accessibilité (« *persistence by reachability* »).

11.4 Hiérarchie de types et de classes

L'expression hiérarchie de classe n'a pas le même sens dans le contexte des bases de données que dans le contexte des langages de programmation. Dans les langages de programmation, une hiérarchie de classes dénote ce qu'on appelle une hiérarchie de types dans les bases de données. Dans les bases de données, une classe est un type **et** un *ensemble* d'instances de ce type. Dans les langages de programmation, une classe est juste un **type**. Pour éviter cette confusion, l'auteur du livre de référence du cours utilise le terme *extension* pour désigner l'ensemble des instances d'une classe.

11.4.1 Hiérarchie de types

En C++ ou en Java, une classe est un type. Une classe peut être une *spécialisation* d'une autre classe; cette dernière est appelée la classe *générale*. La classe spécialisée *hérite* de tous les attributs et de toutes les méthodes définis dans la classe générale. En Java, on utilise le mot clé `extends` pour indiquer qu'une classe est une spécialisation d'une autre classe. Le terme *sous-classe* est aussi souvent employé comme synonyme du terme spécialisation.

Exemple: La déclaration ci-dessous définit la classe `B` comme une spécialisation de la classe `A`.

```
class B extends A { ... }
```

La classe B hérite de tous les attributs et de toutes les méthodes définies dans la classe `A`. En Java, il existe une classe spéciale, `Object`, du package `java.lang`. Par définition du langage Java, toute classe est une spécialisation de la classe `java.lang.Object`. Cette classe contient, entre autres, les déclarations des méthodes `equals` et `clone`.

Exemple: Dans l'exemple ci-dessous, la classe `TupleLivreAudio` est définie comme une spécialisation de la classe `TupleLivre`. Un livre audio est un livre (donc il a tous les attributs d'un livre), mais il a en plus un attribut particulier, soit la durée de la cassette du livre (en minutes). Voici la déclaration de la classe `TupleLivre`.

```
public class TupleLivre {

  private int    idLivre; /* cle */
  private String titre;
  private String auteur;
  private String dateAcquisition;
  private String dateEmprunt;
  private TupleMembre emprunteur;

  /* constructeur */
  public TupleLivre(int idLivre, String titre, String auteur,
                    String dateAcquisition) {

    this.idLivre = idLivre;
    this.titre = titre;
    this.auteur= auteur;
    this.dateAcquisition = dateAcquisition;
  }

  /* emprunter */
  public void emprunter(TupleMembre mb, String dateEmprunt) {

    this.emprunteur = mb;
    this.dateEmprunt = dateEmprunt;
  }
```

```
    public void afficher() {

    System.out.print(idLivre + " " + titre + " " + auteur);
 }

}
```

Voici la déclaration de la classe `LivreAudio`.

```
public class TupleLivreAudio extends TupleLivre {

  private int   duree;

  /* methodes d'acces */
  public int    getDuree() {return duree;}

  /* Constructeur */
  public TupleLivreAudio(int idLivre, String titre, String auteur,
                         String dateAcquisition, int duree) {

    super(idLivre, titre, auteur, dateAcquisition);
    this.duree = duree;
  }

    public void afficher() {

    System.out.print(idLivre + " " + titre + " " + auteur + " " duree);
 }
}
```

Une instance de la classe `TupleLivreAudio` a tous les attributs de la classe `TupleLivre`. Le constructeur de `TupleLivreAudio` fait appel au constructeur de la classe Livre pour les initialiser; cet appel est fait en utilisant le mot clé `super`. On peut ensuite référer aux attributs et aux méthodes d'un livre audio avec la syntaxe habituelle.

```
TupleLivreAudio tupleLivreAudio =
    new TupleLivreAudio(1,"T1","Auteur1","2000-03-01",120);
tupleLivreAudio.preter(m1,"2000-03-02");
```

Le *polymorphisme* est une propriété souvent associée aux langages OO. Il permet de *surcharger* un nom de méthode, c'est-à-dire de définir plusieurs méthodes ayant le même nom. La méthode à invoquer lors d'un appel de méthode est déterminée soit à la compilation (*polymorphisme statique*), ou soit à l'exécution (*polymorphisme dynamique*). Le choix est fait en fonction des types des paramètres actuels de la méthode. Dans l'exemple précédent, il y a surcharge de la méthode `afficher`. Elle est définie à la fois dans la classe `Livre` et dans la classe `LivreAudio`. Lors de l'exécution du code suivant, le système déterminera quelle méthode appeler en fonction de la valeur de la variable `tupleLivre1`.

```
TupleLivre tupleLivre1;
tupleLivre1 = new TupleLivre(1,"T1","Auteur1","2000-03-01");
tupleLivre1.afficher();
tupleLivre1 = new TupleLivreAudio(1,"T1","Auteur1","2000-03-01",120);
tupleLivre1.afficher();
```

On note en premier lieu que la variable `tupleLivre1` est déclarée comme une référence à un objet de la classe `TupleLivre`. Cette variable est d'abord instanciée avec un objet de la classe `TupleLivre`. Le premier appel de la méthode `afficher` entraîne l'exécution de la méthode définie dans la classe `TupleLivre`, car la variable `tupleLivre1` réfère à un objet de cette classe. Ensuite, la variable `tupleLivre1` est instanciée avec un objet de la classe `TupleLivreAudio`. Cela est tout à fait admissible, même si la variable est de type `TupleLivre`. Dans les langages orientés objets, on peut affecter à une variable une instance de n'importe quelle spécialisation de la classe définissant le type de la variable. Le deuxième appel de la méthode `afficher` invoque la méthode définie dans la classe `TupleLivreAudio`, car la variable `tupleLivre1` réfère à un objet de cette classe.

11.4.2 Hiérarchie de classes

Dans le contexte des bases de données, une classe est un ensemble d'objets (du même type ou de types différents). On utilise le concept de classe pour modéliser les données d'une application. Par exemple, la classe `Livre` dans un système de gestion d'une bibliothèque dénote l'ensemble des tous les livres de la collection de la bibliothèque. Les transactions ajoutent, modifient et suppriment les objets de la classe. Le concept de spécialisation a un sens légèrement différent de celui que nous avons vu dans la section précédente. L'expression "la classe B est une sous-classe de la classe A" signifie que tous les objets de B appartiennent aussi à A. En C++ ou en Java, on représente l'ensemble des objets d'une classe (i.e.,, son extension) à l'aide d'un objet de type collection, soit un Set, un Bag, une List, ou bien un Map.

11.5 Le modèle objet de ODMG

Les normes (« *standard* ») permettent de :

- porter facilement une application d'un SGBD à un autre;
- d'évaluer les fonctions offertes par un SGBD et de les comparer à d'autres SGBD;
- d'utiliser plusieurs SGBD différents dans une même application.

La norme ODMG

- propose un modèle objet;
- définit un langage de définition d'objets (LDO, en anglais « *ODL - Object Definition Language*»;

- définit un langage de d'interrogation des objets (LIO, en anglais « *OQL - Object Query Language* »);
- s'applique à trois langages de programmation
  - C++
  - Smalltalk
  - Java.

11.5.1 Les objets

Un objet est décrit selon quatre caractéristiques.

identificateur : identifie de manière unique un objet (deux objets distincts ont des identificateurs différents); c'est une valeur interne au SGBD; il n'est typiquement pas accessible pour les programmes d'application.

nom (optionnel) : C'est un nom persistant. Un programme d'application peut lire un objet en utilisant ce nom. Typiquement, on associe un nom seulement aux collections d'objets.

durée de vie : il y a deux valeurs possibles pour cette caractéristique : transitoire ou persistant (voir chapitre 11).

structure : Un objet est soit atomique ou soit une collection d'objets. Un objet atomique est défini avec les clauses `interface` et `class`. Une `interface` ne peut être instanciée. Une `class` peut être instanciée. On peut lui associer une extension (une collection) dont le nom est spécifié avec le mot clé `extent`. Il existe plusieurs types de collections :

```
Set<t>, Bag<t>, List<t>, Array<t>, Dictionary<k,v>
```

où `t,k,v` sont des types d'objets ou des types de littéraux. Un `Array` est une liste de longueur fixe; on peut dynamiquement modifier la dimension d'un `Array`. Un `List` est de longueur variable. Un `Dictionary` est un synonyme de `Map`.

11.5.2 Les littéraux

Un littéral est une donnée qui n'a pas d'identificateur d'objet. Il ne peut être stocké directement de manière persistante. Pour être persistant, il doit faire partie d'un objet. On dénote trois types de littéraux.

atomic `long`, `short` (entiers), `float`, `double` (points flottants), `char` (caractère), `string` (chaîne de caractères), `boolean` (booléen), `enum` (énumération).

composé `date` (date avec année mois jour), `interval` (intervalle de temps en jours, heures, minutes, secondes, milisecondes), `time` (heure, minute, seconde, miliseconde). On peut aussi définir ses propres structures avec l'opérateur `struct`.

collection Il existe plusieurs littéraux de type collection. Ce sont les mêmes types que pour les objets, sauf que leur nom commence par une lettre minuscule.

```
set<t>, bag<t>, list<t>, array<t>, dictionary<k,v>
```

11.6 Le langage de définition des objets (ODL)

Il permet de définir un schéma de base de données objet. Il est composé de déclarations de classes et d'interfaces. Par souci de concision et de simplicité, nous nous intéressons seulement au concept de classe. Le langage supporte également l'héritage (entre classes, avec le mot clé `extends`, et entre une classe et une interface, ou entre deux interfaces, avec le symbole ":").

11.6.1 Les classes

Une classe est un type et, optionnellement, une collection d'objets. Sa définition comporte donc les éléments suivants.

nom de classe : Spécifié par le mot clé `class`.

nom de la collection (optionnel) : Spécifié par le mot clé `extent`. C'est un nom persistant associé à la collection des objets de la classe.

clés (optionnel) : Spécifiées par le mot clé `key`. On peut associer des clés (une ou plusieurs) à un objet. Une clé est une liste d'attributs. Pour chaque clé, il ne peut exister deux objets ayant les mêmes valeurs pour les attributs de la clé. Ce concept est donc similaire aux concepts de clé primaire (`PRIMARY KEY`) et de clé candidate (`UNIQUE`) dans les bases de données relationnelles.

attributs : Spécifiés par le mot clé `attribute`. Le type d'un attribut est un littéral, une classe ou une collection.

relations : Spécifiées par le mot clé `relationship`. Il s'agit d'une relation *bidirectionnelle* avec un ou des objets d'une autre classe. Pour une relation unidirectionnelle, on utilise le concept d'attribut. Si un programme d'application modifie une direction de la relation, le SGBD modifiera l'autre direction, afin que les deux directions soient cohérentes.

opérations : Spécifiées en donnant la signature de l'opération. On indique la nature de chaque paramètre comme suit : `in` pour un paramètre d'entrée, `out` pour un paramètre de sortie, et `inout` pour un paramètre d'entrée-sortie.

Exemple: Voici le schéma relationnel de la bibliothèque. Seules quelques opérations sont énumérées.

```
class TupleLivre ( extent allLivres
  key idLivre
) { attribute    long        idLivre;
  attribute    string      titre;
  attribute    string      auteur;
  attribute    date        dateAcquisition;
  attribute    date        datePret;
  relationship TupleMembre emprunteur
       inverse TupleMembre::prets;
  relationship list<TupleReservations> reservations
       inverse TupleReservation::tupleLivre;
  void preter(in TupleMembre emprunteur, in date datePret);
  void retourner();
  void reserver(in TupleReservation tupleReservation);
  void annulerRes(in TupleReservation tupleReservation);
  date dateRetour();
}

class TupleMembre (
  extent allMembres
  key idMembre
) {
  attribute    long        idMembre;
  attribute    string      nom;
  attribute    long        telephone;
  attribute    short       limitePret;
  relationship set<TupleLivre> prets
       inverse TupleLivre::emprunteur;
  relationship set<TupleReservation> reservations
       inverse TupleReservation::tupleMembre;

}

class TupleReservation (
  extent allReservations
  key idReservation,
      (tupleLivre,tupleMembre)
) {
  attribute    long        idReservation;
  attribute    date        dateAcquisition;
  relationship TupleLivre  tupleLivre
       inverse TupleLivre::reservations;
  relationship TupleMembre tupleMembre
       inverse TupleMembre::reservations;
}
```

11.7 Le langage de définition des requêtes (OQL)

Il permet d'interroger une base de données objet. La syntaxe est inspirée de SQL.

11.7.1 L'énoncé select de base

- Il retourne un `bag` d'objets ou de littéraux.

  ```
  select t.idTitre from t in allLivres where t.auteur =
  'Elmasri';
  ```

  Le symbole `t` est une variable d'itération sur les éléments de la collection `allLivres`. Il est similaire à un alias de table en SQL. Cet énoncé retourne un `bag<string>` de tous les titres des livres de la collection `allLivres` dont l'auteur est égal à `Elmasri`.

  On peut aussi écrire la clause `from` comme suit:

  ```
  from allLivres t
  ```

  ou bien

  ```
  from allLivres as t
  ```

- On peut utiliser le mot clé `distinct` pour éliminer les doublons. Sans le mot clé `distinct`, un énoncé `select` retourne toujours un `bag`. Avec le mot clé `distinct`, un énoncé `select` retourne toujours un `set`.

  ```
  select distinct t.titre from t in allLivres where t.auteur
  = 'Elmasri';
  ```

  Cet énoncé retourne un `set<String>`.

- L'énoncé suivant retourne une collection d'objets.

  ```
  select t from t in allLivres where t.auteur = 'Elmasri';
  ```

  Cet énoncé retourne un `bag<tupleLivre>`.

- On peut sélectionner un sous-ensemble d'attributs d'un objet en utilisant l'opérateur `struct`.

  ```
  select struct (idLivre : t.idLivre,
                 auteur : t.auteur,
                 dateAcquisition : t.dateAcquisition )
  from t in allLivres where t.auteur = 'Elmasri';
  ```

Le résultat est du type

```
bag<struct(idLivre : long,
           auteur : string,
           dateAcquisition : date
          )
   >
```

- On peut ordonner les éléments du résultat. Le type du résultat est alors une liste.

```
select struct (idLivre : t.idLivre,
               auteur : t.auteur,
               dateAcquisition : t.dateAcquisition )
from t in allLivres where t.auteur = 'Elmasri' order by
t.idLivre asc;
```

Le type du résultat est

```
list<struct(idLivre : long,
            auteur : String,
            dateAcquisition : date
           )
    >
```

- Dans la clause `from`, on peut spécifier une expression qui est une collection (un `set`, un `bag`, une `list`, un `array` ou un `dictionay`).

11.7.2 Navigation

- Dans un énoncé `select`, on peut accéder aux objets auxquels un objet est relié, en naviguant à travers les attributs, les relations et les opérations. On peut le faire dans toutes les clauses de l'énoncé, soit `select`, `from` et `where`. On peut tester si un attribut est défini avec le prédicat `is_defined(expresssion)`.

```
select struct(titre : t.idTitre, idEmprunteur :
t.emprunteur.idmembre) from t in allLivres where
is_defined(t.emprunteur);
```

Le type du résultat est `bag<struct(titre : string, idEmprunteur : long)>`

- On peut omettre l'opérateur `struct` s'il n'y a pas deux expressions qui ont le même nom. Le nom de chaque attribut du résultat est le dernier nom de l'expression.

```
select t.idReservation, t.tupleLivre.idLivre,
t.tupleMembre.idMembre from t in allReservation;
```

Le résultat du select est un

```
bag<struct(idReservation : long ,
           idLivre : long,
           idMembre : long
          )
   >
```

- Valeur indéfinie
  - L'opérateur `.` appliqué à une valeur indéfinie retourne une valeur indéfinie.
  - Les opérateurs de comparaison `=, !=, <, >, <= >=` retourne faux si une de leurs opérandes est indéfinie.
  - Le prédicat `is_defined(expresssion)` retourne vrai ssi `expresssion` est définie.
  - Le prédicat `is_undefined(expresssion)` retourne vrai ssi `expresssion` est indéfinie.
- On peut retourner un bag de `struct` comportant des membres non élémentaires.

```
select struct (idMembre : t.idMembre, emprunts :
t.emprunts) from t in allMembres;
```

Le type du résultat est

```
bag<struct(idMembre : long,
           emprunts : set<tupleLivre>
          )
   >
```

- On peut inclure un select dans un select.

```
select struct
    (
    idMembre : t.idMembre,
    titres : (select u.titre from u in t.emprunts)
    )
from t in allMembres;
```

Le type du résultat est `bag<struct(idMembre : long, titres : set<string>)>`.

Il est incorrect d'appliquer un membre à une collection. Par exemple, l'énoncé suivant est syntaxiquement invalide, car l'attribut `titre` est appliqué à un `set`.

```
select struct
    (
    idMembre : t.idMembre,
    titres : t.emprunts.titre
    )
from t in allMembres;
```

- On peut naviguer aussi loin que l'on désire avec les attributs et les relations.

```
select struct
    (
    idMembre : t.idMembre,
    titresRes : (
        select struct
            (
            titre : u.titre,
            idMembreReservations :
                (select v.idMembre from v in u.reservations)
            )
        from u in t.emprunts)
    )
from t in allMembres;
```

Le type du résultat est

```
bag<struct(idMembre : long,
           titresRes : set<struct(titre : string,
                                  idMembreReservations : set<long>)
                           >
          )
   >
```

11.7.3 Opérations sur les collections

On peut appliquer les opérations suivantes aux collections.

- `count(c)` : retourne le nombre d'éléments de la collection `c`. Par exemple, l'énoncé suivant retourne le nombre d'emprunts pour chaque membre.

  ```
  select count(t.emprunts) from t in allMembres;
  ```

  Le type du query est `bag<long>`

- `avg(c)` : retourne la moyenne des valeurs de la collection `c`.

  ```
  avg(select t.count(t.emprunts) from t in allMembres)
  ```

- on a aussi les fonctions `min,max,sum`.

11.7.4 Expressions booléennes quantifiées

Il existe trois types d'expressions booléennes quantifiées.

1. `e in c` : retourne vrai ssi `e` appartient à la collection `c`.
2. `for all v in c : b` : retourne vrai ssi pour tout élément `v` de la collection `c`, la condition `b` est vraie.
3. `exists v in c : b` : retourne vrai ssi il existe un élément `v` de la collection `c` tel que la condition `b` est vraie.

L'énoncé suivant énumère les membres qui ont le plus grand nombre de prêts.

```
select m1 from m1 in allMembres where for all m2 in
allMembres :
          count(m1.prets) >= count(m2.prets);
```

L'énoncé suivant énumère les paires de membres qui ont au moins un livre en commun en réservation. Notez que l'on peut spécifier plusieurs collections dans la clause `from`. Le résultat est le produit cartésien des collections (comme dans le langage SQL).

```
select struct(membre1 : m1, membre2 : m2) from m1 in
allMembres, m2 in allMembres where exists r1 in
m1.reservations :
          exists r2 in m2.reservations :
              r1.tupleLivre = r2.tupleLivre;
```

11.7.5 Les groupes

On peut regrouper les tuples résultants de la clause `from-where` d'un `select`. L'énoncé suivant regroupe les livres par auteur et date d'acquisition.

```
select * from x in allLivres group by auteur : x.auteur,
dateAcquisition : x.dateAcquisition;
```

Le résultat est du type `set` (car il ne peut y avoir de doublon, en regroupant).

```
set<struct(auteur : string,
           dateAcquisition : date,
           partition : bag<struct(x : TupleLivre)>
          )
   >
```

On peut spécifier des expressions dans la clause `select` pour sélectionner ou extraire un sous-ensemble des attributs du résultat.

```
select auteur, dateAcquisition, nbLivre : count(partition)
from x in allLivres group by auteur : x.auteur,
dateAcquisition : x.dateAcquisition;
```

Le résultat est du type suivant.

```
set<struct(auteur : string,
           dateAcquisition : date,
           nbLivre : long
          )
   >
```

Chapter 12

Exemples de BD OO sous Java

12.1 ObjectStore pour Java

12.1.1 Concepts de base

- nous utilisons la version PSE PRO (Personnal Stotage Edition);
- cette version est conçue pour un seul utilisateur;
- un seul programme à la fois en mode mise à jour de la BD;
- plusieurs programmes en concurrence pour accès en lecture.

12.1.2 Étapes usuelles pour l'utilisation d'une BD object store

1. ouverture d'une session
2. création / ouverture de la BD
3. démarrage d'une transaction
4. création / lecture des racines (noms persistants)
5. fin d'une transaction
6. démarrage d'une transaction
7. mise à jour / accès aux objets associés aux noms persistants
8. fin d'une transaction
9. répéter étapes 6 à 8 si nécessaire
10. fermeture de la BD
11. fermeture de la session.

12.1.3 Langage de définition des objets

- il n'y a pas de langage spécifique de définition des objets;
- on utilise la syntaxe de Java pour décrire les objets;
- pas de distinction au niveau syntaxique entre une déclaration d'un objet transitoire ou persistant;
- Object Store s'occupe de stocker et de lire les objets dans une base de données sur disque;
- l'accès à un objet persistant est très similaire à un objet transitoire;
- les objets sont regroupés dans une BD.

12.1.4 Session

- permet d'accéder à une BD;
- une seule BD à la fois durant une session;
- création d'une session

```
private static Session session;
session = Session.create(null, null);
session.join();
```

- fermeture d'une session

```
session.terminate();
```

12.1.5 Base de données

- la classe `Database` permet de créer une base de données;
- une base de données peut contenir n'importe quel type d'objet persistant.
- la BD doit être créée par un programme Java (pas de LDD externe)

```
private static Database db;
db = Database.create(
        dbname,
        ObjectStore.ALL_READ | ObjectStore.ALL_WRITE);
```

  - `dbname` : une string qui contient le chemin d'accès et le nom du fichier de la base de données; elle doit se terminer par `.odb`.
    Exemple: `"biblio.odb"`

– `ObjectStore.ALL_READ | ObjectStore.ALL_WRITE` :
permission d'accès à la base de données (similaire aux permissions de Unix).

Cette commande crée 3 fichiers : `path.odb`, `path.odt`, `path.odf`. Elle ouvre également la BD en mode `UPDATE`. Une seule BD à la fois peut être ouverte dans une session.

- si la BD existe, un programme Java doit tout d'abord l'ouvrir pour y accéder;

```
db = Database.open(dbName, ObjectStore.UPDATE);
```

  – mode `ObjectStore.UPDATE` : mise à jour de la base de données; un seul programme autorisé à la fois;
  – mode `ObjectStore.READONLY` : lecture seulement; plusieurs programmes à la fois.

- pour fermer la base de données :

```
db.close();
```

- pour détruire la base de données :

```
Database.open(dbName, ObjectStore.UPDATE).destroy();
```

12.1.6 Transaction

1. permet d'accéder aux objets persistants;
2. une transaction est exécutée en entier ou bien pas du tout;
3. les modifications aux objets persistants sont sauvegardées sur le disque à la fin de la transaction;
4. les modifications sont visibles pour les autres programmes seulement lorsque la transaction est terminée;
5. une transaction permet de préserver la cohérence des données;
6. démarrage d'une transaction :

```
Transaction tr = Transaction.begin(mode);
```

   - mode = `ObjectStore.UPDATE` : mise à jour de la BD;
   - mode = `ObjectStore.READONLY` : lecture seulement.

7. fin normale d'une transaction

```
tr.commit(mode_de_retention);
```

La figure 12.1 illustre l'impact des différents mode de rétention sur l'usage des objets.

- mode de rétention = `ObjectStore.RETAIN_HOLLOW` :
  signifie que les objets persistants lus durant la transaction seront disponibles pour la transaction suivante; par contre, ils **ne sont pas disponibles entre les transactions**; si on essaie d'accéder à un objet entre deux transactions, on obtient une exception (NoTransactionInProgressException); plus exigeant pour le ramasse-miettes (*garbage collector*).
- mode de rétention = `ObjectStore.RETAIN_STALE` :
  signifie que les objets persistants lus durant la transaction ne seront pas disponibles pour la transaction suivante; pour être utilisés, ils devront être relus à partir d'une racine; de même ils **ne sont pas disponibles entre les transactions**; plus économe en mémoire vive et plus efficace pour le ramasse-miettes.
- mode de rétention = `ObjectStore.RETAIN_READONLY` :
  signifie que les objets persistants lus durant la transaction sont disponibles jusqu'à la prochaine transaction, mais ils ne peuvent être modifiés; moins économe en mémoire vive.
- mode de rétention = `ObjectStore.RETAIN_UPDATE` :
  signifie que les objets persistants lus durant la transaction sont disponibles jusqu'à la prochaine transaction; ils peuvent être modifiés, mais les modifications sont perdues au démarrage de la prochaine transaction; moins économe en mémoire vive;
- mode de rétention = `ObjectStore.RETAIN_TRANSIENT` :
  signifie que les objets persistants lus durant la transaction sont copiés et transformés en objets transitoires. Lors de la transaction suivante, les objets doivent être relus à partir d'une racine.

8. fin anormale d'une transaction :

   ```
   tr.abort(mode_de_retention);
   ```

   on utilise habituellement `ObjectStore.RETAIN_HOLLOW` .

12.1.7 Persistance

1. nom persistant :

   - appelé un « *root* » (une racine) en Object Store; permet de rendre un objet persistant, de le sauvegarder dans une BD, et de le lire de la BD;
   - un nom persistant doit être créé durant une transaction;
   - création

     ```
     db.createRoot("allUsers", allUsers = new OSHashMap());
     ```

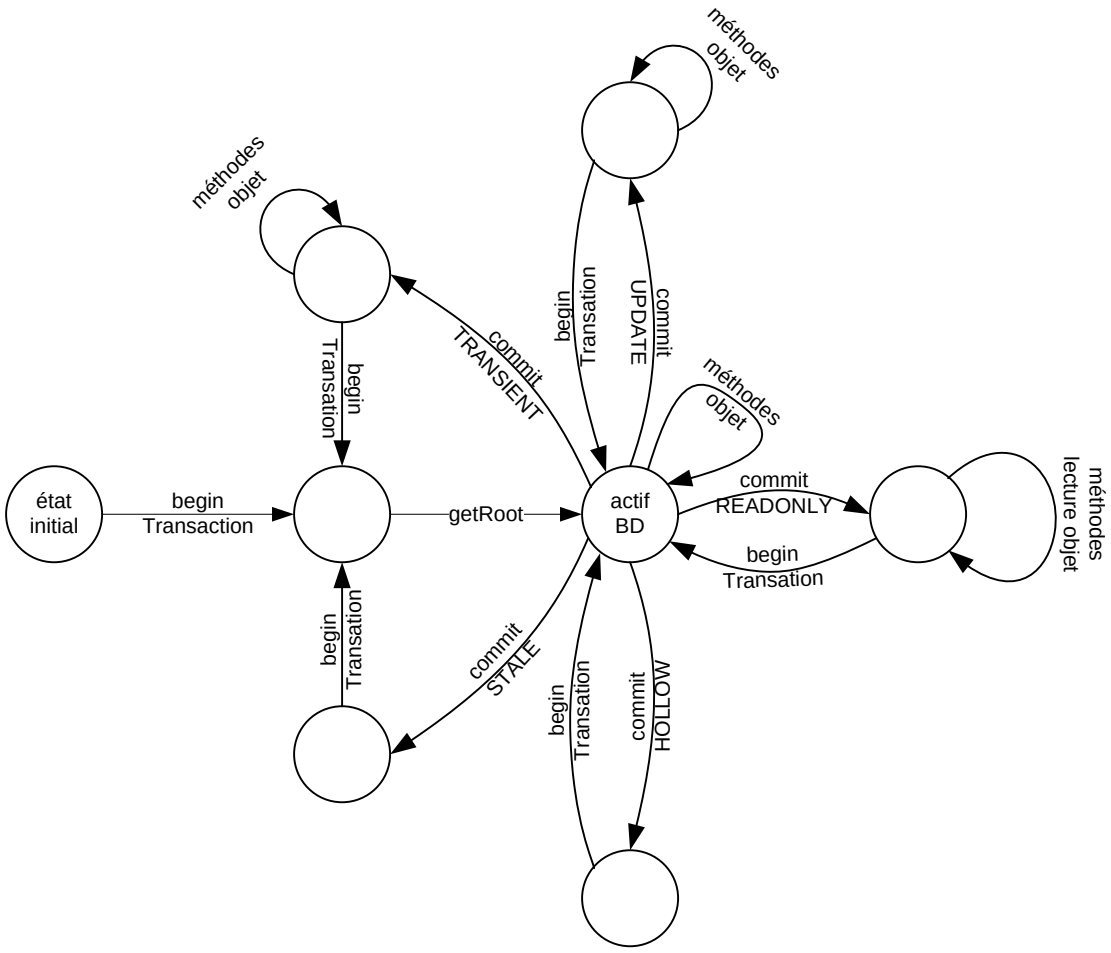

Figure 12.1: Diagramme états-transitions illustrant l'effet des modes de rétention sur l'usage des objets persistants.

  - "allUsers" : nom de la racine (nom persistant)
  - allUsers : objet auquel le nom persistant est associé.
- lecture

  ```
  allUsers = (Map) db.getRoot("allUsers");
  ```

  - (Map) : coercition (conversion) de l'objet associé au nom persistant "allUsers" en un objet de type Map;
- écriture
  - lors d'un commit, les objets ayant un nom persistant ainsi que ceux qu'ils référencent (et qui sont admissibles à la persistance) sont sauvegardés.

2. classe admissible à la persistance (« *persistence capable* »)
   - classe dont les objets peuvent être sauvegardés dans la BD;
   - on peut rendre certaines classes admissibles à la persistance à l'aide du postprocesseur;
   - une version persistante de certaines classes de Java est fournie par Object Store.
     - java.lang.String
     - java.lang.Boolean
     - java.lang.Byte
     - java.lang.Character
     - java.lang.Double
     - java.lang.Float
     - java.lang.Integer
     - java.lang.Long
     - java.lang.Short
     - Array de type primitif, d'Object, de classe admissible à la persistance
     - les classes collections d'Object Store (OSHashMap, OSHashSet, etc).
3. classe devant accéder à des variables d'un objet persistant (« *persistence-aware class* »)
   - une classe qui accède directement à des *variables* d'un objet d'une classe admissible à la persistance, ou des composantes d'un vecteur persistant, doit aussi être traitée par le postprocesseur afin d'y ajouter des annotations pour lire les objets de la BD avant de les modifier;
   - on évite cela en déclarant les variables d'un objet comme étant privées (« *private* »), et en encapsulant l'accès à un objet à l'aide de méthodes. Le postprocesseur s'occupera des annotations pour rendre la classe persistante.
4. état d'un objet persistant
   - actif (« *active* ») : l'objet a été lu de la BD, et son contenu est accessible;

- désactivé (« *stale* »): l'objet a été lu, mais son contenu n'est plus accessible. Pour y accéder, on doit le relire à partir d'une racine durant une transaction;
- creux (« *hollow* »): l'objet a été lu, mais son contenu n'est plus accessible. On peut y accéder durant la prochaine transaction.

5. lors d'un commit, les objets ayant un nom persistant sont sauvegardés, ainsi que les objets admissibles à la persistance auxquels ils réfèrent, transitivement.

6. lorsqu'un objet ayant un nom persistant est lu, les objets admissibles à la persistance qu'il référence sont aussi disponibles; si on accède à une référence d'un objet durant une transaction, Object Store lit l'objet de la base de données.

12.1.8 Collection en Java

1. aussi appelé *container*;

2. on s'en sert typiquement pour regrouper des objets d'une classe;

3. générique: on peut spécifier le type des éléments de la collection

4. plusieurs interfaces du JDK définissent le concept de collection.

   - `Collection<T>` : *bag* (ensemble avec doublons permis)
   - `Set<T>` : ensemble (pas de doublon)
   - `List<T>` : liste (ordonnée)
   - `Map<K,V>` : fonction, donc un ensemble de couples (clé (K), valeur (V)); pas de doublon dans les clés.

5. Object Store fournit une implémentation efficace de ces interfaces pour des collections de grande taille.

Class	**Implements**
OSHashtable	*aucune*
OSHashBag	Collection
OSSmallSet	Set
OSHashSet	Set
OSTreeSet	Set
OSVector	Collection
OSVectorList	List
OSSmallMap	Map
OSHashMap	Map
OSTreeMapByteArray	Map
OSTreeMapDouble	Map
OSTreeMapFloat	Map
OSTreeMapInteger	Map
OSTreeMapLong	Map
OSTreeMapString	Map

12.1.8.1 Collection (Bag)

C'est un multi-ensemble, c'est-à-dire qu'un objet peut apparaître plusieurs fois. Object Store propose une seule implémentation, `OSHashBag`, implémentée avec une table de hachage. Voici les principales méthodes de ce type. La complexité algorithmique est donnée pour les principales méthodes, où n représente le nombre d'éléments dans la collection, c une variable aléatoire qui dépend de la répartition moyenne des éléments dans une table de hachage et k une constante propre à l'implémentation, qui est généralement très petite par rapport à n. Dans le cas normal, c est quasi constant; dans le pire des cas, $c = n$.

- `public OSHashBag<T>(int initSize)` : Constructeur; crée un bag vide dont la taille initiale de la table de hachage est de `initSize` éléments. On peut ajouter plus d'éléments que le nombre `initSize` spécifié; le système ajuste alors la table de hachage en conséquence.

- `public boolean add(T o)` : Ajoute l'objet `o` au bag.
  Complexité: $O(c)$

- `public boolean contains(T o)` : Retourne vrai si l'objet `o` fait partie du bag.
  Complexité: $O(c)$

- `public boolean remove(T o)` : Supprime une occurrence de l'élément `o` dans le bag. Retourne vrai si une occurrence a été supprimée.
  Complexité: $O(c)$

- `public boolean isEmpty()` : Retourne vrai si le bag est vide, faux sinon.
  Complexité: $O(k)$

- `public Iterator iterator()` : Retourne un itérateur sur les éléments du bag.

  Il est n'est pas approprié de modifier une collection durant son parcours avec un itérateur. L'interface `Iterator` fournit une méthode `remove` qui permet de supprimer de la collection l'élément courant. Voir ci-dessous. On peut modifier un objet *référencé* par une collection sans problème.

- `public int size()` : Retourne le nombre d'éléments du bag
  Complexité: $O(k)$

L'interface `Iterator<T>` permet d'itérer sur les éléments d'une collection. Voici les méthodes de `Iterator`.

- `public boolean hasNext()` : Retourne vrai s'il reste un élément à parcourir dans l'itérateur.
  Complexité: $O(k)$

- `public T next()` : Retourne un élément de l'itérateur et avance au prochain.
  Complexité: $O(k)$

- `public void remove()` : Supprime l'élément courant (i.e., celui retourné par `next`) de la collection.
  Complexité: $O(k)$

12.1.8.2 Set

C'est un ensemble, c'est-à-dire qu'un objet ne peut y apparaître plus d'une fois. Object Store propose trois implémentations: `OSSmallSet`, `OSHashSet`, `OSTreeSet`. Voici les principales méthodes de ce type; les méthodes de `Bag` sont aussi disponibles pour les `Set`.

- `public OSSmallSet<T>(int initSize)` : Constructeur; crée un ensemble vide dont la taille initiale de la table de hachage est de `initSize` éléments. On peut ajouter plus d'éléments que le nombre `initSize` spécifié; le système ajuste alors la table de hachage en conséquence. Idéal pour des ensembles de moins de 100 éléments.

- `public OSHashSet<T>(int initSize)` : Similaire à `OSSmallSet`, mais optimisé pour des ensembles de moins de 10000 éléments.

- `public OSTreeSet<T>(Placement placement)` : Constructeur; crée un ensemble vide en utilisant un arbre balancé comme représentation; il faut spécifier le segment de la BD (`placement`) ou sera stocké l'ensemble; la classe `Database` étant une sous-classe de `Placement`, on peut utiliser l'instance de la bd comme paramètre. Idéal pour de très grands ensembles (plus de 10 000 éléments).

- `public boolean add(T o)` : Ajoute l'objet `o` à l'ensemble. Retourne faux si l'objet est déjà présent, sinon retourne vrai.
  Complexité: $O(c)$ pour `OSSmallSet` et `OSHashSet`, $O(log(n))$ pour `OSTreeSet`.

- `public boolean remove(T o)` : Supprime l'objet `o` de l'ensemble. Retourne vrai si l'objet était présent, sinon retourne faux.
  Complexité: $O(c)$ pour `OSSmallSet` et `OSHashSet`, $O(log(n))$ pour `OSTreeSet`.

- `public boolean contains(T o)` : Vérifie l'appartenance de l'objet `o` à l'ensemble. Retourne vrai si l'objet est présent, sinon retourne faux.
  Complexité: $O(c)$ pour `OSSmallSet` et `OSHashSet`, $O(log(n))$ pour `OSTreeSet`.

Voir la documentation pour les autres méthodes de cette `Set`.

12.1.8.3 Liste

Les éléments d'une liste sont ordonnés et numérotés de 0 à n-1, où n est le nombre d'éléments de la liste. On peut utiliser une liste pour représenter une file et une pile. Object Store propose deux implémentations, `OSVector` (implémente l'interface `Collection`) et `OSVectorList` (implémente l'interface `List`). Elles sont identiques, excepté pour la méthode `equals`: `OSVector` implémente `equals` avec l'adresse des objets, donc `l1.equals(l2)` ssi `l1 == l2`; `OSVectorList` implémente `equals` avec l'égalité en contenu comme suit:

$$\texttt{l1.equals(l2)} \Leftrightarrow \texttt{l1.length()} == \texttt{l2.length()} \wedge \forall i : 0 \,..\, \texttt{l1.length()} - 1 : \texttt{l1.get(i).equals(l2.get(i))}$$

- `public OSVector<T>(int initialBufferSize)` : Constructeur; crée une liste vide dont la taille initiale est de `initialBufferSize` éléments. On peut ajouter plus d'éléments que le nombre `initialBufferSize` spécifié; le système ajuste alors la liste en conséquence.

- `public OSVectorList<T>(int initialBufferSize)` : Constructeur; crée une liste vide dont la taille initiale est de `initialBufferSize` éléments, mais qui n'a pas de taille maximale.

- `public boolean add(T o)` : Ajoute l'objet `o` à la fin de la liste.
  Complexité: $O(k)$

- `public boolean add(int index, T o)` : Ajoute l'objet `o` à la position `index`, et décale les autres éléments, à partir de celui à la position `index` avant l'appel, vers la fin. Retourne une exception si l'`index` est invalide `(index < 0 || index > n)`.
  Complexité: $O(n)$

- `public boolean contains(T o)` : Retourne vrai si l'objet `o` fait partie de la liste.
  Complexité: $O(n)$

- `public T get(int index)` : Retourne l'élément à la position `index`. Retourne une exception si l'index est invalide `(index < 0 || index >= n)`.
  Complexité: $O(k)$

- `public boolean remove(T o)` : Supprime la première occurrence de l'élément `o` dans la liste. Retourne vrai si une occurrence a été supprimée.
  Complexité: $O(n)$

- `public T remove(int index)` : Supprime l'élément à la position `index` dans la liste. Les éléments aux positions `index+1` et suivantes sont décalés vers la gauche. Retourne une exception si l'index est invalide `(index < 0 || index >= n)`.
  Complexité: $O(n)$

- `public boolean isEmpty()` : Retourne vrai si la liste est vide, faux sinon.
  Complexité: $O(k)$

- `public Iterator iterator()` : Retourne un itérateur sur les éléments de la liste.

- `public int size()` : Retourne le nombre d'éléments de la liste.
  Complexité: $O(k)$

12.1.8.4 Map

C'est une fonction, donc un ensemble de couples (clé,valeur). Il ne peut y avoir qu'une seule occurrence d'un objet donné comme clé. Une valeur peut être associée à plus d'une clé.

On utilise un map pour chercher rapidement un objet d'une collection d'objets. La clé est typiquement un attribut qui permet d'identifier un objet de manière unique (par exemple, *idLivre* pour un livre, ou *idMembre* pour un membre). La clé doit être une valeur de type référence qui est admissible à la persistance; les types primitifs ne sont pas admissibles. Pour représenter un type primitif, on utilise la classe correspondante dans le package `java.lang`. Par exemple, pour stocker un `int`, on utilise la classe `java.lang.Integer`. Si la clé est formée de plusieurs attributs, on peut utiliser une String contenant la concaténation des

attributs pour former la clé. On peut aussi définir une classee `Cle` dont les attributs sont les éléments de la clé.

Object Store propose trois implémentations : `OSSmallMap`, `OSHashMap`, implémentées avec une table de hachage, et `OSTreeMapxxx` pour les BD de grande taille ($\geq$ 10 000 occurrences), implémentée avec un B-tree, où `xxx` est un des types suivants : ByteArray, Double, Float, Integer, Long, String. Le type `xxx` dénote le type de la clé.

On retrouve les méthodes suivantes dans l'interface `Map`.

- `public OSSmallMap<K,V>(int initialCapacity)` : Constructeur; crée un map vide implémenté avec une table de hachage dont la taille initiale est de `initialCapacity` éléments. On peut ajouter plus d'éléments que le nombre `initialCapacity` spécifié; le système ajuste alors la table en conséquence. `public OSHashMap<K,V>(int initialCapacity)` : Constructeur; crée un map vide implémenté avec une table de hachage dont la taille initiale est de `initialCapacity` éléments. On peut ajouter plus d'éléments que le nombre `initialCapacity` spécifié; le système ajuste alors la table en conséquence.

- `public V put(K key, V value)` : Ajoute le couple (`key`,`value`) au map. Retourne l'objet associé auparavant à la clé `key`.
  Complexité: $O(c)$ pour `OSSmallMap` et `OSHashMap`, $O(log(n))$ pour `OSTreeMapx`

- `public V get(K key)` : Retourne l'élément associé à la clé `key`. Retourne `null` si la clé n'existe pas.
  Complexité: $O(c)$ pour `OSSmallMap` et `OSHashMap`, $O(log(n))$ pour `OSTreeMapx`

- `public V remove(K key)` : Supprime le couple de clé `key` du map. L'objet associé à `key` est retourné en sortie.
  Complexité: $O(c)$ pour `OSSmallMap` et `OSHashMap`, $O(log(n))$ pour `OSTreeMapx`

- `public boolean isEmpty()` : Retourne vrai si le map est vide, faux sinon.
  Complexité: $O(k)$

- `public int size()` : Retourne le nombre de couples (clé,valeur) du map.
  Complexité: $O(k)$

- `public Collection<V> values()` : Retourne une collection des valeurs du map. La collection et le map réfèrent aux mêmes valeurs; donc, une modification d'une valeur de la collection modifie aussi la valeur dans le map, puisqu'il s'agit du même objet.

- `public Set<K> keySet()` : Retourne l'ensemble des clés du map. L'ensemble et le map réfèrent aux mêmes valeurs; donc, une modification de cette clé modifie aussi la valeur dans le map, puisqu'il s'agit du même objet. Il est inapproprié de modifier un objet qui est une clé dans un map si son hashCode n'est pas constant; l'objet peut ne plus être accessible avec les méthodes `get` ou `remove`.

12.1.8.5 Choix d'un type de collection

On peut utiliser les critères du tableau ci-dessous pour choisir un type de collection.

Interface	Classe Implémentation	ordonné	plusieurs occurences même objet	Insertion d'un objet	Suppression d'un objet	appartenance d'un objet	recherche sur un attribut de l'objet	taille maximale recommandée	equals
Collection	OSHashBag	non	oui	c	c	c	n	< 1000	adresse
Set	OSSmallSet	non	non	c	c	c	n	< 100	contenu
Set	OSHashSet	non	non	c	c	c	n	< 10000	contenu
Set	OSTreeSet	non	non	log(n)	log(n)	log(n)	n	< 10^7	contenu
List	OSVector	oui	oui	n	n	n	n index 1	< 10^6	adresse
List	OSVectorList	oui	oui	n	n	n	n index 1	< 10^6	contenu
Map	OSSmallMap	non	clé - non valeur - oui	c	clé - c valeur - n	clé - c valeur - n	n	< 100	contenu
Map	OSHashMap	non	clé - non valeur - oui	c	clé - c valeur - n	clé - c valeur - n	n	< 10000	contenu
Map	OSTreeMapx	non	clé - non valeur - oui	log(n)	clé - log(n) valeur - n	clé - log(n) valeur - n	n	< 10^7	contenu

Légende
n = nb d'éléments de la collection
c = temps constant
Note: toutes ces collections n'ont pas de limite quant à leur capacité; toutefois, elles sont
plus efficaces quand elles contiennent un nombre d'objets inférieur à la taille maximale recommandée

12.1.8.6 Equals, Hashcode et Collection

Les classes implémentées avec une table de hachage, telles ques `OSHashBag`, `OSSmallSet`, `OSHashSet`, `OSSmallMap` et `OSHashMap` utilisent la méthode `hashCode()`, que tout objet dispose par défaut car elle est définie dans la classe `Object`, de laquelle toute classe Java hérite. L'usage d'une table de hachage demande certaines précautions. La figure 12.2 illustre une

table de hachage de dimension 4 et son contenu après l'exécution des instructions suivantes:

```
class MyInt {
private int i;
public MyInt(int i) {this.i = i;}
}

Set<Object> s = new OSHashSet<Object>(4);
s.add(new Integer(0));
s.add(new Integer(4));
s.add(new Integer(0));
s.add(m1 = new MyInt(0));
s.add(m2 = new MyInt(0));
```

Une table de hachage contient un vecteur de taille égale a sa capacité initiale. Un élément de l'ensemble est stocké dans une liste chainée associée à une case du vecteur. Cette case du vecteur est choisie à l'aide la la méthode `hashCode`. Typiquement, un élément `o` est stocké dans la case `o.hashCode() % n`, où `n` est la taille du vecteur; on appelle cette expression une *fonction de hachage*. Puisqu'il est possible que plusieurs objets se retrouvent dans la même case du vecteur suite à l'évaluation de la fonction de hachage, les éléments d'une même case sont stockés dans une liste; on parle alors de *collision* entre éléments. Quand on veut insérer, retirer ou vérifier l'appartenance d'un élément, on évalue cette fonction de hachage pour l'élément et on parcoure ensuite la liste de la case correspondante pour voir si l'élément s'y trouve. Le coût de ces opérations est donc égal à la longueur de la liste (i.e.,au nombre de collisions). Statistiquement, si la valeur du `hashCode` est uniformément distribuée, ce coût devrait être constant et proche de `s.size()/n`. Dans le pire cas, tous les objets de l'ensemble peuvent se retrouver dans la même case; le coût de ces opérations est alors `s.size()`, ce qui est très lent, aussi lent qu'une liste.

Pour que l'utilisation d'une table de hachage soit cohérente, il faut respecter la contrainte suivante:

$$\texttt{o1.equals(o2)} \;\Rightarrow\; \texttt{o1.hashCode() == o2.hashCode()}$$

Donc, si deux objets sont égaux en contenu, alors il doivent avoir le même `hashCode`. Cela permet de visiter une seule case du vecteur et parcourir la liste associée pour vérifier si un objet appartient à l'ensemble. La classe `Object` implémente les méthodes `equals` et `hashCode` avec `==` et l'adresse de l'objet, respectivement, ce qui satisfait la contrainte ci-dessus. Si une classe ne redéfinit pas ces méthodes, alors elle hérite de l implémentation de `Object`. Cela explique donc le comportement du code associé à la figure 12.2. La méthode `hashCode` de la classe `Integer` retourne la valeur de l'entier. Donc les `Integer` 0 et 4 sont stockés dans la case 0, car $0 \bmod 0 = 4 \bmod 4 = 0$. Comme la classe `MyInt` ne redéfinit pas la méthode `hashCode`, elle hérite de la classe `Object`; supposons que l'adresse du premier objet `MyInt` 0, soit `m1`, est 1002 et le deuxième objet `MyInt` 0, soit `m2`, est 1003. Ils sont donc stockés dans les cases $1002 \bmod 4 = 2$ et $1003 \bmod 4 = 3$. La méthode `s.contains(new MyInt(0))` retournera toujours faux, car il n'y a effectivement aucun objet dans l'ensemble avec cette valeur selon la méthode `equals` héritée de `Object`, qui utilise l'adresse de l'objet; comme on crée un nouvel objet lors de l'appel, il aura toujours une nouvelle adresse, forcément unique,

qui n'existera pas dans l'ensemble. Si on veut que deux objets `MyInt` contenant le même entier soient considérés comme égaux, alors il faut redéfinir `equals` et `hashCode`.

```
public int hashCode() {return i;}

public boolean equals(Object o) {
if (o == null) return false;
if (getClass() != o.getClass()) return false;
return (((MyInt) o).i == this.i);
}
```

Si on stocke une `List` dans une table de hachage, il ne faut pas la modifier, car le `hashCode` d'une `List` varie en fonction du contenu de la liste. Ainsi, le dernier appel de `contains` dans le code ci-dessous retourne faux.

```
List<Integer> l = new OSVectorList<Integer>(2);
l.add(1);
s.add(l);
l.add(2);
return s.contains(l); // retourne faux!!!
```

Dans un `OSHashMap`, on peut utiliser une `OSVectorList` pour définir une clé contenant plusieurs valeurs.

```
Map<List<Integer>,String> m = new OSHashMap<List<Integer>,String>(2);
List<Integer> l1 = new OSVectorList<Integer>(2);
l1.add(1);
l1.add(2);
m.put(l1,"a");
List<Integer> l2 = new OSVectorList<Integer>(2);
l2.add(1);
l2.add(2);
return m.get(l2) // retourne "a"
```

Comme `l1` et `l2` sont égales en contenu, elles ont le même `hashCode`.

12.1.9 Traduction d'un modèle entité-relation en un modèle objet

Dans la modélisation des données, on utilise habituellement le diagramme entité-relation pour faire un modèle conceptuel des données d'un système. Si on choisit d'implémenter le système avec une base de données relationnelle, on traduit alors le modèle entité-relation en un modèle relationnel; ces algorithmes de traduction sont décrits dans le cours ift187. Si on choisit d'implémenter le système avec une BD OO comme Object Store, on peut utiliser les algorithmes de traduction suivants. La figure 12.3 illustre un exemple de base qui contient tous les types d'entités et de relations.

- Traduction des entités

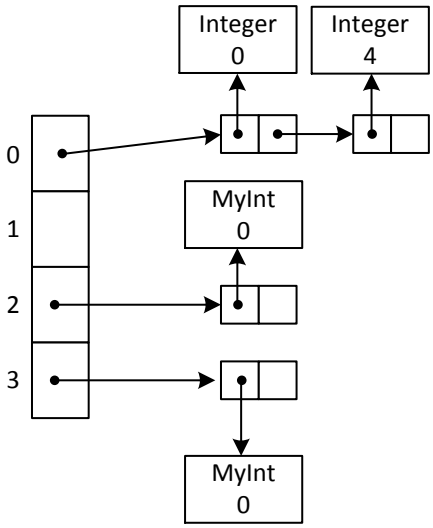

Figure 12.2: Implémentation d'un `Set` avec une table de hachage

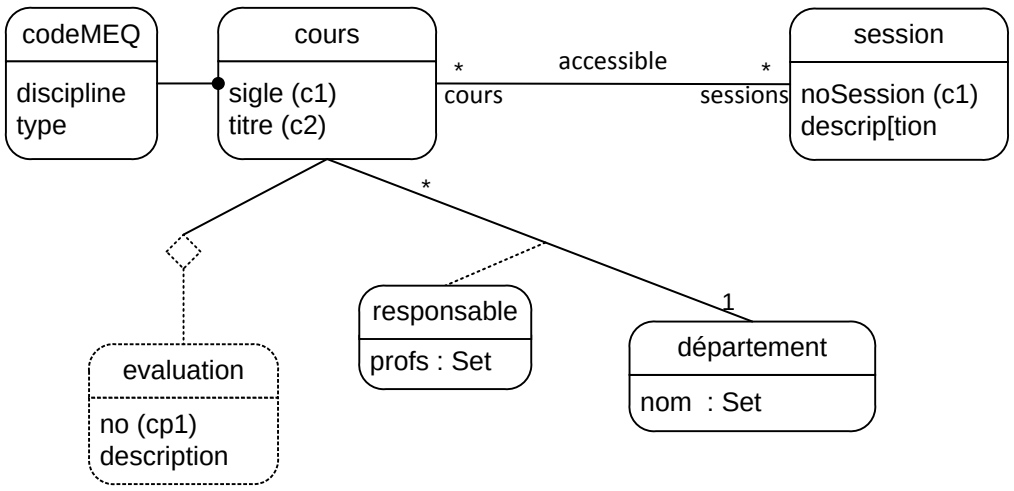

Figure 12.3: Modèle entité-relation illustrant la traduction vers un modèle objet

- **entité** : Une entité (ordinaire) `E` est traduite en deux classes:
  1. Une classe `Tuple_E` pour représenter les instances de l'entité.
  2. Une classe `Coll_E` pour contenir l'ensemble des instances de l'entité. On choisira une collection appropriée en fonction du type d'accès requis. Typiquement, une entité est représenté par un `Map` où la clé est une des clés candidates et la valeur la classe `E` représentant les instances de l'entité.
- **entité faible** : Une entité faible `F` est traduite de manière similaire à une entité ordinaire, sauf que la clé du `Map` de la collection est formée de la clé de l'entité père et de la clé partielle de l'entité faible. On peut soit utiliser une `OSVectorList` pour former une clé, ou bien créer une classe `Cle_F`, en prenant soin de redéfinir les méthodes `equals` et `hashCode` si on utilise un `OSHashMap` ou un `OSSmallMap`. Si on utilise une des classes `OSTreeMapx`, l'option la plus simple est de choisir `OSTreeMapString` et de convertir la clé en une seule chaine de caractères, en concaténant les valeurs des attributs de la clé.

  Une autre alternative consiste à utiliser la classe `Tuple_E` de l'entité père `E` de l'entité faible `F` et d'y ajouter un attribut `Map<K,Tuple_F>`, où `K` désigne la clé de l'entité faible. Pour accéder à une instance de l entité faible, on fait d'abord un get de l'entité père, puis un un `get` sur les instances des entités faibles associées à cette instance de l'entité père.

- Traduction des associations

  - **association binaire sans attribut** : Une association binaire ordinaire `A` entre les entités `E1` et `E2` (sans classe associative) est représentée en ajoutant des attributs aux classes `Tuple_E1` et `Tuple_E2`. Si une instance de `E1` est associée a au plus une instance de `E2`, alors on utilise un attribut `Tuple_E2 r`, où `r` est le nom de rôle (par convention en UML). Si une instance de `E1` est associée a au plus une instance de `E2`, alors il faut choisir une collection appropriée. Typiquement, on utilisera un `Set` s'il n'y a pas d'ordre sur les instances associées (par exemple les prêts d'un membre ne sont pas ordonnés) ou une `List` si les instances sont ordonnées (par exemple les réservations d'un livre dont stockés selon l'ordre d'arrivée). Le choix de la collection dépend toujours des accès qui sont effectués; ce n'est que lorsqu'on a terminé la conception des méthodes de l'application que l'on sait réellement quel type d'accès est requis.

    Comme on navigue généralement une association dans les deux directions, il faut aussi ajouter un attribut dans `Tuple_E2` en pour permettre d'accéder aux instances de `E1` associées à une instance de `E2`, en utilisant la même stratégie.
  - **association binaire avec classe associative** : Une association binaire `A` entre les entités `E1` et `E2` et comprenant une classe associative (généralement appelée aussi `A`) est représentée comme suit. On crée une classe `Tuple_A` qui comprend ses attributs propres ainsi que des attributs vers les entités associées, selon la cardinalité de l'association, exactement comme dans une association binaire sans attributs. On ajoute dans les classes `Tuple_E1` et `Tuple_E2` des attributs pour accéder aux instances de la classe associatives.

– **association n-aire** : il faut créer une classe associative et la traiter comme une classe associative binaire.

La traduction en Java du modèle entité-relation de la figure 12.3 est donnée ci-dessous. On remarque qu'il y a quatre classes `Coll_`e pour chaque entité e (faible ou ordinaire). Il n'y a pas de classe pour stocker une collection d'instances de CodeMEQ, car ce n'est pas une entité, mais simplement une classe pour stocker un attribut complexe de l'entité cours; cela est signifié par une association d'aggrégation représentée par une boule noire à l'extrémité de l'association du coté de l'entité cours. La clé du `Map` dans `Coll_Evaluation` est une `List<Object>`, car on doit pouvoir y stocker une `String` pour le sigle et un `Integer` pour le no de l'évaluation; cela nous force donc à utiliser le type le plus général qui comprend à la fois `String` et `Integer`; seule la classe `Object` satisfait cette contrainte. Finalement, la classe `TupleResponsable` utilise un `Set<String>` pour stocker les profs responsables d'un cours, puisqu'il y a potentiellement plusieurs professeurs responsables d'un cours.

```
public classe Coll_Cours {
    private Map<String,TupleCours> lesCours;
...}

public classe Coll_Session {
    private Map<String,TupleSession> lesSessions;
...}


public classe Coll_Departement {
    private Map<String,TupleDepartement> lesDepartements;
...}

public classe Coll_Evaluation {
    private Map<List<Object>,TupleEvaluation> lesEvaluations;
...}

public class TupleCours {
    /* attributs propres */
    private String sigle;
    private String titre;
    private CodeMEQ codeMEQ
    /* associations */
    private Set<TupleSession> sessions;
    private TupleResponsable responsable;
    private Map<Integer,TupleEvaluation> lesEvaluations;
...}

public class TupleSession {
    /* attributs propres */
```

```
    private String noSession;
    private String description;
    /* associations */
    private Set<TupleCours> cours;
...}

public class TupleDepartement {
    /* attributs propres */
    private String nom;
    /* associations */
    private Set<TupleResponsable> responsablesCours;
...}

public class TupleResponsable {
    /* attributs propres */
    private Set<String> profs;
    /* associations */
    private TupleCours cours;
    private TupleDepartement departement;
...}

public class TupleEvaluation {
    /* attributs propres */
    private int no;
    /* associations */
    private TupleCours cours;
...}

public class CodeMEQ {
    /* attributs propres */
    private String discipline;
    private String type;
...}
```

12.1.10 Query

- on peut exécuter un query sur un objet persistant des classes `OSHashBag`, `OSHashSet`, `OSTreeSet`, `OSVector` et `OSVectorList` (package `com.odi.util.Collection`)
- un query s'applique à un seul objet; il consiste en une expression booléenne; cette expression peut référer à des variables ou des méthodes publiques de la classe des objets de la collection; il s'agit d'une condition exprimée avec une syntaxe similaire à Java.
- il y a deux étapes

– création d'un objet `Query`

```
Query query = new Query(TupleLivre.class, "getAuteur() == \"a1\"");
```

Ce query permet de sélectionner des objets de la classe `TupleLivre`. Le premier paramètre, `TupleLivre.class` est un nom de classe suivi de l'attribut `class`. Le deuxième paramètre est une `String` qui contient le texte de la requête. On y appelle la méthode `getAuteur()` et on détermine si le résultat de l'appel est égal à la chaîne de caractères `a1`. Pour signifier qu'il s'agit d'une chaîne de caractères, on doit l'entourer de deux `"`. Comme cette chaîne fait partie du texte de la requête, on utilise `\"` pour indiquer au compilateur java que le symbole `"` fait partie du texte de la requête (et qu'il ne représente pas la fin du deuxième paramètre du constructeur).

On peut utiliser les opérateurs suivants pour exprimer des conditions.

* arithmétique : `+ / - * \%`
* relationnel binaire : `> < <= >= == !=`
* concatenation de `String` : `+`
* condition (négation, et, ou): `! && ||`
* coercition (*cast*) : `(type)`

On peut appliquer les opérateurs relationnels binaires à des `String`.

– exécution d'un query : Un query ne peut s'appliquer qu'aux objets de type `OSHashBag`, `OSHashSet`, `OSTreeSet`, `OSVector` et `OSVectorList`. Si on désire l'exécuter sur un `Map`, il faut tout d'abord extraire du `Map` les valeurs qu'il contient avec la méthode `values`. On peut ensuite appliquer la méthode `select` au query en lui passant en paramètre la collection des objets. Cette méthode retourne une collection d'objets de la classe identifiée lors de la création du query (dans notre exemple, des objets de la classe `Tuplelivre`).

```
Collection livreQuery = query.select(allLivres.values());
```

– impression du résultat : On peut utiliser un `Iterator` pour parcourir les objets de cette collection. Dans l'exemple ci-dessous, on imprime les objets en utilisant la méthode `afficher` définie dans la classe `TupleLivre`.

```
Iterator livreIter = livreQuery.iterator();

while (livreIter.hasNext()) {
    ((TupleLivre) livreIter.next()).afficher();
    }
```

• query avec variable (paramètre)

1. création d'un objet pour contenir les variables;

```
FreeVariables freeV = new FreeVariables();
```

2. définition des variables;

```
freeV.put("x", String.class);
```

Le premier argument est une `String` qui donne le nom de la variable. Le deuxième argument est un nom de classe (voir la liste des objets de la classe `java.lang` à la page 77). La méthode `put` n'accepte qu'une valeur de type `String` comme premier argument, et une classe comme deuxième argument;

3. création du query avec les variables :

```
Query query = new Query(TupleLivre.class, "getAuteur() == x", freeV);
```

4. création d'un objet pour contenir les valeurs des variables :

```
FreeVariableBindings freeVB = new FreeVariableBindings();
```

5. affectation d'une valeur à chaque variable;

```
freeVB.put("x", auteur);
```

6. exécution du query.

```
Collection queryResults = query.select(allLivres.values(), freeVB);
```

Exemple: Le code source suivant recherche les livres dont l'auteur est "a1".

```
FreeVariables freeV = new FreeVariables();
freeV.put("x", String.class);
Query query = new Query(TupleLivre.class, "getName() == x", freeV);
FreeVariableBindings freeVB = new FreeVariableBindings();
freeVB.put("x", "a1");
Collection queryResults = query.select(allLivres.values(), freeVB);
```

- filtre pour les `String` : On peut utiliser l'opérateur `~~` qui ressemble à l'opérateur `like` de SQL. Il prend à gauche une expression de type `String` et à droite un patron. Ce patron peut contenir les opérateurs suivants

Opérateur	Description
`?`	Dénote n'importe quel caractère
`*`	chaîne de 0 ou plusieurs caractères
`[`	réservé
`]`	réservé
`(`	réservé
`)`	réservé
`|`	réservé
`&?`	?
`&*`	*
`&[`	[
`&]`	]
`&(`	(
`&)`	)
`&|`	|
`&&`	&
`&i`	permet une comparaison indépendante de la casse dans le reste du patron

Dans l'exemple ci-dessous, on sélectionne tous les livres qui ont comme auteur une `String` commençant par `?foo` en majuscule ou en minuscule, suivie ensuite de n'importe quel caractère, suivie ensuite de `*` et finalement suivie d'une chaîne quelconque (possiblement vide).

```
new Query(TupleLivre.class,"getAuteur() ~~ \"&i&?foo?&**\"");
```

Pour d'autres exemples, voir le fichier `TestQuery.java` sur le site web du cours. Ce programme utilise les classes définies pour le tp3.

- index : il est possible d'utiliser un index pour accélérer le traitement d'un query ou trier le résultat. Les index sont un peu plus difficiles à utiliser (il faut les mettre à jour manuellement lors de la mise à jour d'un objet).

12.1.10.1 Tri du résultat d'un query

Un query retourne une collection. On peut trier une collection et mettre le résultat dans une liste grâce à la méthode générique `Collections.sort`. Il en existe deux versions:

```
sort(List<T> list)
```

```
sort(List<T> list, Comparator<? super T> c)
```

La première version trie la liste selon l'ordre *naturel* des objets de la classe `T`. Cet ordre dit "naturel" est défini par la méthode `compareTo`. Pour que le compilateur puisse vérifier que la classe `T` défini bien cette méthode, la classe `T` doit implémenter l interface `Comparable`, qui contient seulement cette méthode. Par exemple. si on veut trier des livres selon l'ordre naturel des livres, on doit avoir ceci dans la classe `TupleLivre`

```
public class TupleLivre implements Comparable<TupleLivre> {
...
public int compareTo(TupleLivre o)
{
return this.getIdLivre() - o.getIdLivre();
}
```

La méthode `o1.compareTo(o2)` retourne un entier :

- $= 0$, si les deux objets sont considérés comme égaux selon cet ordre;
- < 0, si `o1` est considéré inférieur à `o2` selon cet ordre;
- > 0, si `o1` est considéré supérieur à `o2` selon cet ordre.

Pour la classe `TupleLivre` décrite ci-dessus, nous avons choisi comme ordre naturel l'ordre croissant sur le no de livre. Si on veut comme ordre naturel l'auteur en ordre croissant et le titre en ordre décroissant, on peut implémenter `compareTo` comme suit:

```
public int compareTo(TupleLivre o)
{
if (o1.auteur.equals(o2.auteur))
    // ordre décroissant de titre pour un même auteur
    return o2.titre.compareTo(o1.titre);
else
    // ordre croissant d'auteur
    return o1.auteur.compareTo(o2.auteur);
}
```

Comme il est souvent nécessaire d'avoir plusieurs façons de trier une collection d'objets, la deuxième version de `Collections.sort` permet de spécifier un `Comparator`, plutôt que d'utiliser l'ordre naturel. Cette interface prescrit une méthode `compare` similaire à `compareTo`.

```
public class CompTupleLivreTitre implements Comparator<TupleLivre> {
    public int compare(TupleLivre o1, TupleLivre o2)
    {
    return o1.getTitre().compareTo(o2.getTitre());
    }
}
```

Voici comment on utilise les deux versions de la méthode `sort`.

```
// tri selon l'ordre naturel de TupleLivre
LinkedList<TupleLivre> l = new LinkedList<TupleLivre>(lesLivres.values());
Collections.sort(l);
Iterator<TupleLivre> livreIter = l.iterator();

while (livreIter.hasNext()) {
```

```
    livreIter.next().afficher();
}

// tri selon l'ordre spécifié par le comparator CompTupleLivreTitre
Collections.sort(l, new CompTupleLivreTitre());
livreIter = l.iterator();

while (livreIter.hasNext()) {
    livreIter.next().afficher();
}
```

12.1.11 Compilation et postprocesseur

- on compile les programmes java comme s'ils étaient des programmes ordinaires;
- on exécute ensuite le postprocesseur sur les fichiers `.class` des classes qui doivent être admissibles à la persistance;
- il faut traiter ensemble les classes qui sont reliées entre elles

  `osjcfp -inplace -dest . Interest.class User.class`

  - remplace les fichiers `.class` par une version annotée qui gère la persistance;

  Si plusieurs classes doivent être traitées, on peut exécuter le postprocesseur lui donnant un fichier en paramètre qui contient la liste des classes à traiter. Il faut traiter toutes les classes en même temps.

- Par exemple, voici le contenu du fichier `cfpargs.txt`

  ```
  -inplace -dest .
  -pc TupleLivre.class TupleMembre.class TupleReservation.class
  ```

  On appelle ensuite le postprocesseur avec ce fichier.

  `osjcfp @cfpargs.txt`

12.1.12 Quelques commandes utilitaires

- ramasse-miettes (« *garbage collector* »): `osjgcdb <database_name>.odb`

  Il élimine tous les objets qui ne sont pas accessibles à partir d'une racine.

- afficher informations à propos d'une BD : `osjshowdb <database_name>.odb`

  Il affiche les informations suivantes.

  - noms persistants (racines)
  - objets détruits
  - nombre d'objets par type d'objet

12.1.13 Analogie entre BD relationnelle et Object Store

- pour avoir l'équivalent d'une table, il faut
  - définir une classe avec les attributs;
  - rendre cette classe admissible à la persistance avec le postprocesseur;
  - définir une variable de type collection (`Set`, `Map`, etc) pour contenir tous les objets de la classe; on utilise typiquement un `Map` avec les attributs de clé primaire comme clé du `Map`.
  - donner un nom persistant à cette variable (c'est le nom de la table).
- un objet correspond à un tuple d'une relation
- ajout d'un tuple
  - démarrer une transaction;
  - faire un getRoot sur la racine de la table pour lire la collection;
  - ajouter l'objet dans la collection;
  - commit de la transaction.

  note: on peut faire le getRoot est une seule fois dans une transaction de la méthode d'initialisation du programme, avec un commit `RETAIN_HOLLOW`
- modification d'un tuple
  - démarrer une transaction;
  - faire un getRoot sur la racine de la table pour lire la collection;
  - obtenir l'objet de la collection;
  - modifier l'objet;
  - commit de la transaction.
- destruction d'un tuple
  - démarrer une transaction;
  - faire un getRoot sur la racine de la table pour lire la collection;
  - éliminer l'objet de la collection, et de tous les autres attributs où il est référencé.
  - commit de la transaction

  Cette approche ne détruit pas physiquement l'objet de la BD. Mais comme il n'est plus référencé par un objet ayant un nom persistant, il sera éliminé par le ramasse-miette d'Object Store.

  Si on veut immédiatement éliminer physiquement un objet de la BD après avoir éliminé toutes les références à cet objet, on utilise la méthode `ObjectStore.destroy(o)`. On peut dissocier un objet d'un nom persistant en utilisant la méthode `db.setRoot`

`(nomPersistant, null)`. Notez que cette méthode ne supprime pas l'objet associé à la racine. Le nom persistant peut ensuite être associé à un autre objet en utilisant encore la méthode `setRoot`. On peut supprimer un nom persistant avec la méthode `db.destroyRoot (nomPersistant)`.

- il n'y a pas de concept de contrainte d'intégrité en Object Store;
- il n'y a pas de concept d'énoncé `select` en Object Store;
- il n'y a pas d'énoncé `alter table` en Object Store. Il faut manuellement convertir la BD en utilisant la sérialisation de Java.

12.2 FastObjects pour Java avec JDO

12.2.1 Introduction

- FastObjects est une base de données objet satisfaisant la norme JDO (Java Data Objects)
- nous utilisons la version j1, du domaine public, qui comporte certaines limitations (mono-utilisateur, index à un seul attribut);

12.2.2 Étapes usuelles pour l'utilisation d'une BD FastObjects

1. rendre les classes admissibles à la persistance en traitant leur fichier .class avec un post-processeur.
2. ouverture d'une connexion et création / ouverture de la BD avec un `PersistenceManager`
3. démarrage d'une transaction
4. lecture des objets persistants à partir d'un extent
5. rendre un objet persistant
6. fin d'une transaction
7. fermeture de la connexion via le `PersistenceManager`.

12.2.3 Langage de définition des objets

- il n'y a pas de langage spécifique de définition des objets;
- on utilise la syntaxe de Java pour décrire les objets;
- pas de distinction au niveau syntaxique entre une déclaration d'un objet transitoire ou persistant;

- FastObjects s'occupe de stocker et de lire les objets dans une base de données sur disque;
- l'accès à un objet persistant est très similaire à un objet transitoire;
- tous les objets sont regroupés dans une seule BD.

12.2.4 Compilation et post-processeur

- Pour qu'un objet puisse être stocké dans une BD objet, il faut que sa classe soit traitée afin d'être admissible à la persistance.
- On compile les classes java à rendre admissibles à la persistance comme si elles étaient des classes ordinaires. On les traite ensuite avec un post-processeur.
- Le post-processeur modifie le bytecode de la classe c'est-à-dire le fichier `.class`, pour y insérer du code de lecture et d'écriture d'un objet. Le fichier `.class` est donc ré-écrit.
- Après chaque compilation d'un `.java`, il faut donc exécuter le post-processeur
- Chaque classe à traiter avec le post-processeur doit être décrite par un fichier `<nomDeClasse>.jdo`. Ce fichier, écrit en format XML, donne certaines propriétés de la classe. Dans le cadre du cours, nous nous limitons à y décrire les index, qui sont utilisés pour accéder rapidement à un objet de l'`extent` associé à la classe. Voici un exemple, le fichier `TupleLivre.jdo` pour la classe `TupleLivre.java`.

```
<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE jdo SYSTEM
"jdo.dtd"> <jdo>
  <package name="">
    <class name="TupleLivre">
      <extension key="index" value="indexIdLivre" vendor-name="FastObjects">
        <extension key="member" value="idLivre" vendor-name="FastObjects"/>
        <extension key="unique" value="true" vendor-name="FastObjects"/>
      </extension>
      <extension key="index" value="indexAuteur" vendor-name="FastObjects">
        <extension key="member" value="auteur" vendor-name="FastObjects"/>
      </extension>
    </class>
  </package>
</jdo>
```

Ce fichier `.jdo` définit un index, appelé `indexIdLivre`, pour l'extent de la classe `TupleLivre`. Cet index porte sur l'attribut `idLivre` et il est `unique`, c'est-à-dire qu'il ne pourra y avoir deux objets de la classe `TupleLivre` ayant la même valeur pour `idLivre`. Un index unique est similaire à une clé primaire d'une table dans une BD relationnelle. Ce fichier `.jdo` définit un deuxième index, appelé `indexAuteur`, portant sur l'attribut `auteur` de la classe `TupleLivre`. Cet index n'est pas unique, donc il

pourra y avoir deux livres ayant le même auteur. La version `j1` que nous utilisons pour le cours, qui est gratuite et du domaine public, ne permet pas de définir des index à plus d'un attribut. Les versions complètes de FastObjects le permettent.

La norme JDO impose l'existence d'un extent pour chaque classe. Un extent contient tous les objets de la BD pour une classe donnée. Si une classe a des sous-classes, il est possible de mettre tous les objets dans un seul extent, celui de la classe générale. La section 12.2.9 décrit comment on peut lire un objet d'un extent. Lorsqu'on rend un objet persistant, il est automatiquement ajouté à l'extent de sa classe (ou de sa super-classe). Lorsqu'on supprime un objet persistant, il est automatiquement supprimé de son extent (voir section 12.2.8).

- On exécute le post-processeur sur les fichiers `.class` des classes qui doivent être admissibles à la persistance; il faut traiter ensemble les classes qui sont reliées entre elles. Voici la commande à exécuter sous DOS ou UNIX pour appeler le post-processeur.

  ```
  ptj -enhance
  ```

- Si les attributs d'une classe font référence à d'autres classes, ces classes doivent aussi être traitées par le post-processeur, afin de les rendre admissibles à la persistance. Les attributs de type primitif (boolean, byte, short, int, long, char, float, double) sont automatiquement admissibles à la persistance.

- On doit déclarer tous les attributs d'une classe admissible à la persistance comme étant `private`.

- Les attributs de type `static` ne sont jamais stockés avec un objet; en effet, comme ce sont des attributs de classe, il n'est pas logique de les stocker avec un objet. Pour stocker ces attributs, il faut leur donner un nom persistant (voir section 12.2.8).

- Comme on ré-utilise souvent certaines classes du JDK, FastObjects offre une version post-processé de plusieurs classes utilitaires du JDK.

  - `java.lang`:
    `Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `String`
  - `java.util`:
    `Locale`, `Date`, `Collection`, `Vector`, `List`, `LinkedList`, `ArrayList`, `Set`, `HashSet`, `SortedSet`, `TreeSet`, `Map`, `HashMap`, `Hashtable`, `SortedMap`, `TreeMap`
  - `java.math`:
    `BigDecimal`, `BigInteger`

12.2.5 Persistence Manager

- joue le rôle d'une connexion à la BD;

```
try
    {
    java.util.Properties pmfProps = new java.util.Properties();

    // Obtenir un PersistenceManagerFactory et d{\'e}finir le nom de la BD
    pmfProps.put(
        "javax.jdo.PersistenceManagerFactoryClass",
        "com.poet.jdo.PersistenceManagerFactories" );
    pmfProps.put(
    // url de la BD; on utilise ici le mode local
        "javax.jdo.option.ConnectionURL",
        "fastobjects://LOCAL/" + dbName + ".j1" );
    PersistenceManagerFactory pmf =
        JDOHelper.getPersistenceManagerFactory( pmfProps );

    // Obtenir un PersistenceManager
    try
        {pm = pmf.getPersistenceManager();}
    catch ( JDOUserException e )
        {
        //cr{\'e}ation de la BD si elle n'existe pas.
        DatabaseAdministration.create(pmf.getConnectionURL(), null);
        pm = pmf.getPersistenceManager();
        }
    }
catch (Exception e)
    {
    System.out.println(e);
    throw new BiblioException("Impossible d'ouvrir la connexion");
    }
```

- fermeture d'une connexion

```
if ( pm.currentTransaction().isActive() )
    pm.currentTransaction().rollback();
pm.close();
```

12.2.6 Base de données

- une base de données peut contenir n'importe quel type d'objet persistant.
- la BD doit être créée par un programme Java (voir section 12.2.5); il pas de LDD externe comme en SQL.

```
DatabaseAdministration.create(
```

```
"fastobjects://LOCAL/" + dbName + ".j1",
null);
```

Cette commande crée un fichier `<dbName>.j1` dans le répertoire courant où le programme Java a été démarré.

- pour fermer la base de données, on ferme simplement le `PersistenceManager`

```
pm.close();
```

12.2.7 Transaction

1. permet d'accéder aux objets persistants;
2. on ne peut accéder aux objets persistants que durant une transaction.
3. une transaction est exécutée en entier ou bien pas du tout;
4. les modifications aux objets persistants sont sauvegardées dans la base de données à la fin de la transaction;
5. les modifications sont visibles pour les autres programmes seulement lorsque la transaction est terminée;
6. une transaction permet de préserver la cohérence des données;
7. format typique d'une transaction

```
pm.currentTransaction().begin();
try {
    // validation et maj des objets
    ...
    pm.currentTransaction().commit();
    }
catch (Exception e)
    {
    if (pm.currentTransaction().isActive())
        pm.currentTransaction().rollback();
    throw e;
    }
```

12.2.8 Persistance

1. un objet est persistant lorsqu'il est stocké dans une base de données;
2. pour qu'on puisse le stocker dans une base de données, il faut d'abord traiter sa classe avec le post-processeur (voir section 12.2.4);

3. pour rendre un objet persistant, il faut qu'au moins une des deux conditions suivantes soit satisfaite :

   - l'objet est rendu persistant en exécutant la méthode suivante :

     `pm.makePersistent( obj );`

     On peut aussi affecter un nom à un objet lors de l'appel de cette méthode.

     `PersistenceManagers.makePersistent(pm, obj, "nom" )`

   - l'objet est référencé par un objet persistant; c'est ce qu'on appelle la *persistance transitive*.

4. pour supprimer un objet persistant d'une base de données, on utilise la méthode suivante :

   ```
   pm.deletePersistent( obj );
   ```

   Il faut aussi enlever toute référence à cet objet dans les autres objets.

5. lors d'un commit, les objets persistants sont sauvegardés dans la base de données et ajoutés automatiquement à leur extent.

6. valeur `null` des attributs des types suivants :

   Si est un attribut d'une classe admissible à la persistance est de type `java.lang.Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Double`, and `Float`, une valeur `null` est convertie en la valeur 0 dans ce type lorsque l'objet est écrit dans la BD. Pour un attribut de type Boolean, la valeur `null` est remplacée par `false`.

   Lorsque l'objet sera lu de la base de données, la valeur `null` aura donc disparu. Il est donc préférable que les constructeurs de classes admissibles à la persistance initialisent eux-mêmes ces attributs à la valeur désirée, pour éviter les surprises.

   Pour les autres types, les valeurs `null` sont préservées lors de l'écriture.

12.2.9 Lecture d'un objet d'une BD

On peut utiliser un des mécanismes suivants pour lire un objet d'une BD.

1. en utilisant l'*extent* de la classe de cet objet et les index définis sur cette classe à l'aide du fichier `.jdo` (voir section 12.2.4)

   ```
   Extent allLivres = pm.getExtent(TupleLivre.class, true);
   Iterator iter = Extents.iterator(allLivres, "indexIdLivre");
   TupleLivre t = null;
   if (Extents.findKey(iter, new Integer(idLivre)))
       t = (TupleLivre) iter.next();
   allLivres.close(iter);
   return t;
   ```

2. en utilisant le nom persistant de l'objet :

```
PersistenceManagers.findObject(pm, "tm" );
```

3. en utilisant un query sur l'extent de la classe

```
public void listerLivresRetard(Date dateVisee)
{
GregorianCalendar gc = new GregorianCalendar();
gc.setTime(dateVisee);
gc.add(Calendar.DATE, -14);
Date dateLimitePret = gc.getTime();
Extent allLivres = pm.getExtent(TupleLivre.class, true);
Query qry = pm.newQuery();
qry.declareImports("import java.util.Date");
qry.declareParameters("java.util.Date dateLimite");
qry.setFilter("datePret < dateLimite");
qry.setCandidates(allLivres);
Collection res = (Collection) qry.execute(dateLimitePret);
Iterator iter = res.iterator();
System.out.println("Liste des livres en retard");
while (iter.hasNext())
    {
    TupleLivre t = (TupleLivre) iter.next();
    double diff = (dateLimitePret.getTime() -
        t.getDatePret().getTime())/(24*60*60*1000);
    System.out.println(t + " Retard = " + diff);
    }
}
```

4. en utilisant une propriété (méthode ou attribut) d'un objet déjà lu.

5. en itérant sur tous les objets de l'extent, ce qui n'est pas très rapide.

```
Extent allLivres = pm.getExtent(TupleLivre.class, true);
Iterator iter = allLivres.iterator();
while (iter.hasNext())
    {
    ...
    }
```

12.2.10 Collection en Java

Contrairement à ObjectStore, Fastobjects utilise les implémentations du JDK; elles ne sont donc pas efficaces pour des collections de grande taille. Il vaut mieux utiliser un extent pour stocker une collection de grande taille (plusieurs milliers d'objets).

12.2.11 Analogie entre BD relationnelle et BD objets

- Pour avoir l'équivalent d'une table, il faut :
  - définir une classe avec les attributs;
  - rendre cette classe admissible à la persistance avec le post-processeur;
  - utiliser un extent (le plus efficace en FastObjects pour des collections de grande taille) ou bien définir une variable de type collection (`Set`, `Map`, etc) pour contenir tous les objets de la classe (si la collection est petite);
  - si on utilise une collection, donner un nom persistant à cet objet;
- un objet correspond à un tuple d'une relation
- ajout d'un tuple
  - démarrer une transaction;
  - ajouter l'objet dans l'extent avec un appel à `makePersistent`;
  - commit de la transaction.
- modification d'un tuple
  - démarrer une transaction;
  - obtenir l'objet de la collection (via l'extent et un index, ou bien via un query).
  - modifier l'objet;
  - commit de la transaction.
- suppression d'un tuple
  - démarrer une transaction;
  - éliminer toute référence à cet objet dans les autres objets
  - faire un `deletePersistent` sur l'objet
  - commit de la transaction
- Le seul concept de contrainte d'intégrité en FastObjects est l'index unique.
- Il n'y a pas de concept d'énoncé `select` en FastObjects; on utilise un `query`.
- Il n'y a pas d'énoncé `alter table` en FastObjects. On change simplement la définition de la classe et on utilise le post-processeur pour modifier les objets existants dans la base de données.

```
ptj -enhance -update -database <DatabaseName>
```

Chapter 27

Le langage XML

27.1 Introduction

- XML (eXtensible Markup Language) est une norme industrielle, indépendante des plateformes (UNIX, WIN, LINUX, etc) pour la représentation des données (n'importe quel type de données : texte, entier, structure de données, etc).
- SGML (Standard Generalized Markup Language) père "spirituel" de XML; langage très puissant permettant de définir des documents très complexes; trop puissant et exigeant pour les applications web, on décida d'en faire une version plus simple, XML.
- HTML (Hypertext Markup Language) est une application de SGML pour le web; ses instructions sont interprétables par un fureteur comme Firefox pour afficher le contenu dans une forme prédéterminée.
- XML vs HTML : HTML comporte des balises (*tag*) prédéterminés; XML permet de définir et d'utiliser n'importe quelle balise.
- DTD (*document type definition*) permet de définir les balises d'un document XML. Un document XML peut contenir un DTD ou bien faire référence à un DTD. Un document XML est *valide* s'il satisfait le format défini dans son DTD associé. Un document XML est dit *bien formé* s'il respecte la syntaxe du langage XML. Un document valide est nécessairement bien formé.
- Un schéma XML permet aussi de définir les balises d'un document XML. Toutefois, il permet de spécifier des contraintes plus fortes, en introduisant la notion de type de données comme *string*, *date*, *integer*, *float*. On peut donc spécifier des types plus forts qu'avec un DTD, ce qui permet de faire des validations plus fortes d'un document XML.
- analyseur syntaxique XML (*XML parser*) permet de vérifier si un document est bien formé et s'il est valide (si un DTD ou un schéma XML lui est associé).
- DOM (Document Object Model) : permet de générer une représentation objets, sous forme d'un arbre, pour un document XML.

- Permet de générer des définitions de classe à partir d'un schéma XML. On peut ensuite représenter un document XML en instanciant ces classes avec l'aide d'un analyseur syntaxique.
- XSL (Extensible Style sheet Language) permet de définir une traduction d'un fichier XML en d'autres formats, comme HTML ou PDF. Avec XSL, on peut donc spécifier comment afficher un document XML en passant par une représentation HTML du contenu.
- XMI (XML Metadata Interchange) est un vocabulaire XML qui permet de représenter en XML une spécification UML d'un système. Par exemple, il est possible de stocker une spécification d'un système conçue avec l'outil Rational Rose dans un fichier de format XML.
- Quelques sites sur XML
  - `http://www.xml.org`
  - `http://java.sun.com/xml`
  - `http://www.jxml.com`

27.2 Structure d'un document XML

27.2.1 Caractéristiques générales

- Un document XML est dépendant de la casse (comme en Java ou en C++); par exemple, les noms `Membre` et `membre` sont considérés comme distincts.
- Un document XML peut être rédigé en utilisant un jeux de caractères particulier; il n'est pas restreint à l'ASCII.
- Un commentaire débute par la balise `<!--` et se termine par la balise `-->`

  ```
  <!-- Voici un exemple de commentaire -->
  ```

  Un commentaire peut être inséré n'importe où dans un document XML.

27.2.2 Entête

Un document XML débute par la ligne suivante:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
```

Cette ligne indique

- `version="1.0"` : à quelle du version du standard XML le document se conforme;
- `encoding="ISO-8859-1"` : quel jeux de caractères est utilisé dans le document;

- – `standalone="yes"`" : si le document est indépendant d'autres documents.
  – `standalone="no"`" : si le document est dépendant d'autres documents (par exemple, le document réfère à un DTD)

27.2.3 Corps

Le corps d'un document XML est formé d'un (et un seul) *élément*. Cet élément peut se décomposer en un nombre arbitraire de sous-éléments; un sous-élément peut aussi être décomposé en sous-éléments, récursivement. L'élément principal est appelé la racine du document XML.

Un élément x est défini par une balise de début `<x>` et une balise de fin `</x>`. Par exemple, l'élément `membre` est défini par les balises suivants.

```
<membre> ... </membre>
```

Le nom d'une balise (par exemple `membre`) doit débuter par une lettre, et peut être suivi d'une suite de lettres ou de chiffres (comme un identificateur en Java).

Un élément peut comporter des attributs. Ils sont notés à même la balise de début. Par exemple, l'élément `membre` suivant comporte trois attributs : `idMembre`, `nom`, `citation`.

```
<membre
  idMembre="1"
  nom="Robert D'Astous"
  citation='il a dit:"Quoi?!?"'
  >
</membre>
```

La valeur d'un attribut est délimitée soit par des doubles guillemets (`"`), soit par des simples guillemets (`'`). La valeur d'un attribut **ne doit pas** contenir les caractères suivants : `<`, `&`. On peut inclure un double guillemets (`"`) si la valeur est délimitée par des simples guillemets, et vice-versa. Pour inclure de manière arbitraire les caractères `<`, `&`, `"`, `'`, on utilise une *entité*. En XML, une entité est reconnue par un analyseur syntaxique XML et substituée par la chaîne de caractères correspondante, un peu comme les *escape sequence* de C ou Java. Les entités de la figure 27.1 sont prédéfinies en XML. On peut aussi définir ses propres entités. En utilisant ces entités, on peut écrire l'entité membre comme suit et lui ajouter l'attribut `condition` qui contient la valeur `x < y && y > x`.

```
<membre
  idMembre="1"
  nom='Robert D&amp;quot;Astous'
  citation="il a dit:&amp;apos;Quoi?!?&amp;apos;"
  condition="x &amp;lt; y &amp;amp;&amp;amp; y &amp;gt; x"
  >
</membre>
```

Entité	Caractère substitué
&amp;	&
&lt;	<
&gt;	>
&quot;	’
&apos;	"

Figure 27.1: Entités prédéfinies de XML

On appelle *contenu d'un élément* les informations apparaissant entre la balise de début et la balise de fin d'un élément. Le contenu peut inclure du texte ou d'autres éléments, que l'on appelle alors des sous-éléments.

L'élément `membre` peut aussi être décrit à l'aide de sous-éléments, plutôt que des attributs comme ce fut le cas dans l'exemple précédent.

```
<membre>
  <idMembre>1</idMembre>
  <nom>Robert D'Astous</nom>
  <citation>il a dit:"Quoi?!?"</citation>
  <condition>x &lt; y &amp;amp; y > x</condition>
</membre>
```

Le texte contenu dans un élément **ne doit pas** contenir les caractères `<` et `&` ni la chaîne de caractères `]]>`[1]. On peut utiliser les entités pour inclure de manière arbitraire ces caractères.

Si un élément x ne contient pas de sous-élément, on peut alors le définir par une seule balise se terminant par `/>` (au lieu de simplement `>`) et omettre la balise de fin `</x>`. Par exemple, on peut réécrire l'exemple du membre défini par des attributs comme suit, car il ne comporte aucun sous-élément.

```
<membre
  idMembre="1"
  nom="Robert D'Astous"
  citation='il a dit:"Quoi?!?"'
  condition="x &lt; y &amp;amp; y &gt; x"
  />
```

La figure 27.2 illustre une bibliothèque comportant deux livres et un membre. L'élément `biblio` en est la racine. La figure 27.3 reprend les mêmes données que celles de l'exemple précédent, mais représentées sous formes d'attributs d'un élément. La figure 27.4 représente l'expression arithmétique $(x + 5) * (y - (6 + z))$ en XML. On remarque qu'une expression est composée d'une étiquette qui contient l'opérateur et de sous-expressions qui contiennent les opérandes de l'opérateur. On note que ce fichier XML représente l'expression sous forme d'un arbre comme à la figure 27.5.

[1]Cette chaîne sert de balise de fin pour un type particulier de contenu appelé CDATA.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>

<biblio>
  <!-- livre 1 -->
  <livre>
    <idLivre>1</idLivre>
    <titre>Le matou</titre>
    <auteur>Yves Beauchemin</auteur>
    <dateAcquisition>1999-12-31</dateAcquisition>
  </livre>

  <!-- livre 2 -->
  <livre>
    <idLivre>2</idLivre>
    <titre>Le principe du geyser</titre>
    <auteur>Stephane Bourguignon</auteur>
    <dateAcquisition>2001-02-28</dateAcquisition>
    <idMembre>1</idMembre>
    <datePret>2002-03-13</datePret>
  </livre>

  <!-- membre 1 -->
  <membre>
    <idMembre>1</idMembre>
    <nom>Macha Limonchink</nom>
    <telephone>5143328970</telephone>
    <limitePret>10</limitePret>
    <nbPret>1</nbPret>
  </membre>
</biblio>
```

Figure 27.2: Un document XML décrivant une bibliothèque de deux livres et un membre

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>

<biblio>
  <!-- livre 1 -->
  <livre
    idLivre="1"
    titre="Le matou"
    auteur="Yves Beauchemin"
    dateAcquisition="1999-12-31"
  />

  <!-- livre 2 -->
  <livre
    idLivre="2"
    titre="Le principe du geyser"
    auteur="Stephane Bourguignon"
    dateAcquisition="2001-02-28"
    idMembre="1"
    datePret="2002-03-13"
  />

  <!-- membre 1 -->
  <membre
    idMembre="1"
    nom="Macha Limonchink"
    telephone="5143328970"
    limitePret="10"
    nbPret="1"
  />
</biblio>
```

Figure 27.3: Un document XML décrivant une bibliothèque avec des attributs

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>

<!--
    exemple d'expression arithmetique representee en XML
    (x+5)*(y-(6+z))
-->

<expression etiquette="*">
  <expression etiquette="+">
    <expression etiquette="x"/>
    <expression etiquette="5"/>
  </expression>
  <expression etiquette="-">
    <expression etiquette="y"/>
    <expression etiquette="+">
      <expression etiquette="6"/>
      <expression etiquette="z"/>
    </expression>
  </expression>
</expression>
```

Figure 27.4: Un document XML décrivant l'expression arithmétique $(x + 5) * (y - (6 + z))$

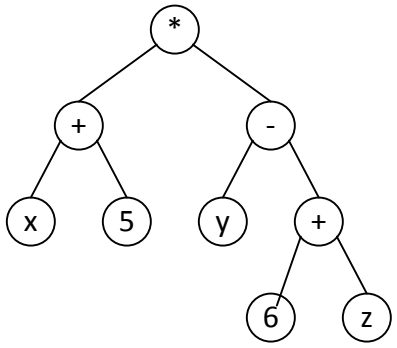

Figure 27.5: Arbre de l'expression arithmétique $(x + 5) * (y - (6 + z))$

```
 1:<?xml version='1.0' encoding='ISO-8859-1'?>
 2:<!-- DTD de la bibliotheque -->
 3:<!ELEMENT biblio (livre | membre)*>
 4:
 5:<!-- livre -->
 6:<!ELEMENT livre (idLivre, titre, auteur, dateAcquisition,
 7:                                  idMembre?, datePret?)>
 8:<!ELEMENT idLivre (#PCDATA)>
 9:<!ELEMENT titre (#PCDATA)>
10:<!ELEMENT auteur (#PCDATA)>
11:<!ELEMENT dateAcquisition (#PCDATA)>
12:<!ELEMENT datePret (#PCDATA)>
13:
14:<!-- membre -->
15:<!ELEMENT membre (idMembre, nom, telephone, limitePret, nbPret)>
16:<!ELEMENT idMembre (#PCDATA)>
17:<!ELEMENT nom (#PCDATA)>
18:<!ELEMENT telephone (#PCDATA)>
19:<!ELEMENT limitePret (#PCDATA)>
20:<!ELEMENT nbPret (#PCDATA)>
```

Figure 27.6: Un DTD pour le document XML de la figure 27.2

27.3 DTD

Un DTD permet de définir un type de document XML qui spécifie les balises attendues et leur structure. Un document XML est dit valide s'il satisfait les règles décrites dans le DTD qu'il référence. Le langage Java comporte des librairies permettant de vérifier si un document XML est valide.

Le DTD de la figure 27.6 définit le type d'un document XML pour une bibliothèque décrite avec des éléments; le document XML de la figure 27.2 est valide par rapport à ce DTD. Notons que les numéros de ligne apparaissant dans l'exemple ne font pas partie du DTD; ils ont été ajoutés pour faire référence à des lignes particulières du DTD afin de l'expliquer. L'entête d'un DTD (ligne 1) est similaire à celle d'un document XML. Le reste du DTD décrit les éléments admissibles. On décrit d'abord l'élément racine, ses attributs et les sous-éléments qui le composent. La balise `<!ELEMENT` permet de définir la structure d'un élément (c-à-d ses sous-éléments). La ligne 3 indique que la racine du document XML est l'élément `biblio`. Cet élément est composé d'une suite, potentiellement vide, (dénotée par l'opérateur `*`) de sous-éléments qui sont soit des éléments `livre`, soit des éléments `membres` (dénoté par l'opérateur `|`). La déclaration de la structure d'un élément XML est donnée par une expression construite à l'aide des opérateurs de la figure 27.7. Les opérateurs `*`, `+`, `|`, `,` sont n-aires et les expressions e peuvent être récursives. Le DTD de la figure 27.8 définit les documents XML contenant une expression arithmétique comme celui de la figure 27.4. On note que l'élément `expression` est récursif.

Opérateur	Définition
`EMPTY`	aucun contenu dans l'élément, donc élément avec une seule balise $< x/>$
`#PCDATA`	suite de caractères
x	un sous-élément x
e`?`	zéro ou une occurrence de l'expression e
e`*`	zéro ou plusieurs occurrences de l'expression e
e`+`	une ou plusieurs occurrences de l'expression e
e_1 `|` ... `|` e_n	choix entre les expressions $e_1 \dots e_n$
e_1 `,` ... `,` e_n	séquence des expressions $e_1, \dots, e_n$

Figure 27.7: Opérateurs pour définir un élément dans une DTD

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!-- Expression arithmetique  -->
<!ELEMENT expression (expression)*>
<!ATTLIST expression
    etiquette CDATA #REQUIRED
    >
```

Figure 27.8: Un DTD pour le document XML de la figure 27.4

Une élément qui comporte seulement du texte est défini à l'aide du mot clé `#PCDATA` (ex: les lignes 6 à 12 et 15 à 20 de la figure 27.6). Un élément qui comporte à la fois du texte et des sous-éléments est dit *mixte* et il est défini comme suit:

`<!ELEMENT elementAvecTexte (#PCDATA |` x_1 `|` ... `|` x_n`)*>`

Notons qu'un élément mixte ne peut inclure de sous-expressions.

Les attributs d'un élément sont décrits à l'aide de la balise `<!ATTLIST`. Le DTD de la figure 27.9 définit le type d'un document XML pour une bibliothèque décrite avec des éléments; le document XML de la figure 27.3 est valide par rapport à ce DTD. Un attribut est défini par un expression de la forme

$$nomAttribut \quad Type \quad ValeurDefaut$$

Le $Type$ peut être l'une des valeurs décrites à la figure 27.10. La $ValeurDefaut$ détermine si l'attribut est optionnel ou obligatoire et quelle est la valeur par défaut si l'attribut n'est pas spécifié. La figure 27.11 décrit ces valeurs. L'exemple de la figure 27.12 illustre l'usage de chaque $ValeurDefaut$. Les attributs `nom` et `sexe` sont obligatoires. Les autres attributs sont optionnels. Si les attributs `etat` et `origine` ne sont pas spécifiés, un analyseur syntaxique du document XML retournera les valeurs `vivant` et `terrien`, respectivement, pour ces attributs. La valeur de l'attribut `nom` est unique dans tout le document XML. L'attribut `parrain` doit être une valeur spécifiée dans un ID d'un autre élément du document XML. L'attribut `origine` ne peut prendre que la valeur `terrien`. Le document XML de la figure 27.13 est conforme au DTD de la figure 27.12. La balise `<!DOCTYPE` permet de référer au DTD auquel

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!-- DTD de la bibliotheque -->
<!ELEMENT biblio (livre | membre)*>

<!-- livre -->
<!ELEMENT livre EMPTY>
<!ATTLIST livre
    idLivre CDATA #REQUIRED
    titre CDATA #REQUIRED
    auteur CDATA #REQUIRED
    dateAcquisition CDATA #REQUIRED
    idMembre CDATA #IMPLIED
    datePret CDATA #IMPLIED
    >

<!-- membre -->
<!ELEMENT membre EMPTY>
<!ATTLIST membre
    idMembre CDATA #REQUIRED
    nom CDATA #REQUIRED
    telephone CDATA #REQUIRED
    limitePret CDATA #REQUIRED
    nbPret CDATA #REQUIRED
    >
```

Figure 27.9: Un DTD pour le document XML de la figure 27.3

Type	Signification
`CDATA`	suite de caractères
`ID`	nom unique pour tout le document XML
`IDREF`	référence à un ID
(v_1 `|` ... `|` v_n)	énumération de valeurs possibles pour l'attribut

Figure 27.10: Type des attributs dans un DTD

$ValeurDefaut$	Signification
`#REQUIRED`	l'attribut doit toujours être spécifié
`#IMPLIED`	l'attribut est optionnel
`#FIXED` v	l'attribut est optionnel; sa valeur ne peut être que v
v	l'attribut est optionnel; sa valeur par défaut est v

Figure 27.11: Spécification des valeurs par défaut des attributs dans un DTD

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!ELEMENT terre (personne)*>
<!ELEMENT personne EMPTY>
<!ATTLIST personne
    nom ID #REQUIRED
    surnom CDATA #IMPLIED
    etat (vivant | mort ) "vivant"
    parrain IDREF #IMPLIED
    sexe (masculin | feminin ) #REQUIRED
    origine CDATA #FIXED "terrien"
    >
```

Figure 27.12: Un DTD illustrant la définition d'attributs

le document XML doit se conformer; elle indique que le DTD est dans le fichier `terre.dtd` et l'élément racine est `terre`; il faut alors indiquer dans la première ligne du document XML qu'il dépend d'un autre document en indiquant `standalone="no"`.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE terre SYSTEM "terre.dtd">
<terre>
  <personne
    nom="Paul"
    surnom="polo"
    etat="mort"
    sexe="masculin"
    origine="terrien"
  />
  <personne
    nom="Jean"
    parrain="Paul"
    sexe="masculin"
  />
</terre>
```

Figure 27.13: Un document XML conforme au DTD de la figure 27.12

27.4 Schéma XML

Un schéma XML permet de définir la structure d'un document XML. Un schema définit des *types* et associe un type à chaque élément. Un type décrit la structure d'un élément, c'est-à-dire ses attributs et ses sous-éléments. Il y a deux sortes de types : *simple* et *complexe*. Un type simple détermine soit un élément n'ayant ni attribut et ni sous-élément, ou soit un

Type Simple	**Exemples de valeur** (exemples séparées par des virgules)
`string`	abc12
`integer`	..., -1, 0, 1, ...
`long`	-9223372036854775808, ..., -1, 0, 1, ..., 9223372036854775807
`int`	-2147483648, ..., -1, 0, 1, ..., 2147483647
`short`	-32768, ..., -1, 0, 1, ..., 32767
`byte`	-128, ...,-1, 0, 1, ..., 127
`decimal`	-1.23, 0, 123.4, 1000.00
`float`	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN (point flottant 32 bits)
`double`	-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN (point flottant 64 bits)
`boolean`	true, false, 1, 0
`duration`	P1Y2M3DT10H30M12.3S (ce qui vaut 1 an, 2 mois, 3 jours, 10 heures, 30 minutes, et 12.3 secondes)
`dateTime`	1999-05-31T13:20:00.000-05:00 (31 mai 1999 à 13:20:00, -5 heures par rapport à GMT)
`date`	1999-05-31
`time`	13:20:00.000, 13:20:00.000-05:00

Tableau 27.1: Les types simples prédéfinis de schéma XML

attribut d'un élément. Il existe des types simples prédéfinis pour la plupart des types de base généralement utilisés dans les applications informatiques. Le tableau 27.1 énumère les plus communs. On peut aussi définir des types simples à partir d'un type simple existant.

Les types complexes dénotent des éléments qui ont des attributs ou des sous-éléments. Il existe trois principaux constructeurs de types complexes : `sequence` (la séquence), `choice` (le choix) et `all` (un *ensemble* de sous-éléments, c'est-à-dire que chaque sous-élément apparaît au plus une fois et dans un ordre arbitraire). Le nombre d'occurrences peut être spécifié pour les éléments d'une séquence et d'un choix, ce qui offre un mécanisme similaire aux opérateurs `*`, `+` et `?` des DTD.

Un schéma XML utilise la notion de *namespace*, qui est similaire à la notion de *package* en Java. Un *namespace* dénote des éléments XML. Un fichier XML pourrait faire référence à deux schémas XML; par exemple, si une application doit communiquer avec deux compagnies différentes avec un seul fichier XML, ce fichier XML devrait utiliser des éléments provenant de deux schémas XML distincts. Pour distinguer ces éléments ayant le même nom dans chaque schéma, on utilise un préfixe associé au *namespace* du schéma.

27.4.1 Entête d'un schéma XML

Un schéma XML débute par les lignes suivantes.

```
<?xml version="1.0" encoding='ISO-8859-1'?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="urn:ift287:biblio:element"
            xmlns="urn:ift287:biblio:element"
            elementFormDefault="qualified"
```

```
            >
..
</xsd:schema>
```

La première ligne indique, comme pour tout fichier XML, la version XML utilisée ainsi que le jeu de caractères utilisé pour l'encodage du schéma XML. La deuxième ligne contient la balise d'ouverture `<xsd:schema">`, qui indique qu'on définit un schéma XML. Comme les schémas XML adhèrent à la syntaxe de XML (ce que les DTD ne font pas), une balise de fermeture de cet élément doit apparaître à la fin du fichier. Le nom de ces balises utilise le préfixe `xsd`. Ce préfixe est au choix du spécifieur; il est déterminé par l'attribut `xmlns:`*xsd*. Ainsi, tous les éléments définit dans le fichier XML `http://www.w3.org/2001/XMLSchema` (la norme schéma XML qui définit les éléments servant à construire un schéma XML, comme `sequence` et `choice`) seront utilisées dans ce schéma avec un préfixe `xsd`. Un autre préfixe aurait pu être choisi, par exemple `toto`, et il serait déclaré ainsi:

```
<toto:schema xmlns:toto="http://www.w3.org/2001/XMLSchema"
...
</toto:schema>
```

L'attribut `targetNamespace="urn:ift287"` définit le *namespace* du schéma XML; ce nom sera utilisé par les fichiers XML pour référencer ce schéma. L'attribut `xmlns="urn:ift287"` indique que les éléments définis dans le schéma XML sont, par défaut, associé au *targetNamespace* et sont référencés sans utiliser de préfixe. L'attribut `elementFormDefault= "qualified"` indique que les éléments d'un fichier XML conforme devront toujours être qualifié par un préfixe, à moins qu'on indique que le *namespace* du fichier XML soit, par défaut, celui du schéma. C'est ce que nous utiliserons en général dans nos exemples.

Voici comment un fichier XML ferait référence à ce schéma XML.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<biblio xmlns="urn:ift287:biblio:element"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="urn:ift287:biblio:element
                         file:./biblio-element.xsd">
```

La deuxième ligne contient la balise de début du fichier XML, soit `biblio`, ainsi que la référence à un schéma XML, grâce à l'attribut `xmlns="urn:ift287:biblio:element"`. La troisième ligne indique que les autres attributs du fichier XML proviennent d'un autre *namespace*, soit le standard

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

Les quatrième et cinquième lignes indiquent que le schéma XML identifié par le *namespace* `urn:ift287:biblio:element` se trouve dans le fichier `file:./biblio-element.xsd`.

27.4.2 Définition de type d'éléments

Après l'entête d'un schéma XML, on associe un type à l'élément racine du schéma XML comme suit:

```
<xsd:element name="biblio" type="Biblio"/>
```

Cet élément déclare un élément `biblio` dont la structure est définie par le type `Biblio`. Par convention, nous utilisons comme nom de type le nom de l'élément, mais débutant par une majuscule. Le type `Biblio` est ensuite défini comme suit:

```
<xsd:complexType name="Biblio">
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="livre" type="Livre"  />
        <xsd:element name="membre" type="Membre"  />
    </xsd:choice>
</xsd:complexType>
```

Le type `Biblio` est défini avec le constructeur `choice`, qui dénote un choix, exactement comme l'opérateur `|` dans un DTD. Les attributs `minOccurs` et `maxOccurs` indiquent le nombre d'occurrences du choix. Les composantes du choix peuvent être des éléments, ou une expression formée avec les constructeurs de type `choice`, `sequence` et `all`, récursivement. Pour décrire le sens du type `Biblio`, voici l'expression correspondante dans une DTD:

```
(livre | membre)*
```

On constate que la syntaxe des schémas XML est moins concise que celle des DTD (`;-)`).

Les opérateurs `?`, `*` et `+` des DTD sont représentés par les attributs `minOccurs` et `maxOccurs` dans un schéma XML. Ces attributs peuvent être spécifiés pour les balises `choice`, `sequence` et `element`. S'ils ne sont pas spécifiés, leur valeur par défaut est 1. Voici la correspondance

Opérateurs DTD	**attributs schéma XML**
`+`	`minOccurs="1" maxOccurs="unbounded"`
`*`	`minOccurs="0" maxOccurs="unbounded"`
`?`	`minOccurs="0" maxOccurs="1"`

Naturellement, on peut aussi spécifier `minOccurs="`m`" maxOccurs="`n`"`, avec $m \leq n$, pour indiquer un intervalle particulier.

À titre d'exemples d'usage des types, voir `biblio-element.xsd` et `biblio-attribut.xsd` sur la page web du cours.

27.4.3 Définition des attributs d'un élément

Un élément comportant des attributs est défini comme suit:

```
...
<xsd:element name="livre" type="Livre"  />
...
<xsd:complexType name="Livre">

    ... sous-éléments ...
```

```
    <xsd:attribute name="idLivre"         type="xsd:integer" use="required"/>
    <xsd:attribute name="titre"           type="xsd:string"  use="required"/>
    <xsd:attribute name="auteur"          type="xsd:string"  use="required"/>
    <xsd:attribute name="dateAcquisition" type="xsd:date"    use="required"/>
    <xsd:attribute name="idMembre"        type="xsd:integer" use="optional"/>
    <xsd:attribute name="datePret"        type="xsd:date"    use="optional"/>
</xsd:complexType>
```

La balise `attribute` ne peut apparaître que dans un `complexType`. L'attribut `type` est un type simple, par exemple un des types prédéfinis du tableau 27.1 ou un nouveau type simple. L'attribut `use` indique si l'attribut défini est obligatoire (`required`) ou optionnel (`optional`).

27.4.4 Définition de type par extension

On peut définir un nouveau type en référant à un type existant et en ajoutant des contraintes supplémentaires. Voici un exemple pour l'attribut `nbPret` de l'élément `membre`, qui réfère au nouveau type simple `integerMinMax`.

```
<xsd:complexType name="Membre">
    ...
    <xsd:attribute name="nbPret"     type="integerMinMax" use="required"/>
    ...
</xsd:complexType>

<xsd:simpleType name="integerMinMax">
    <xsd:restriction base="xsd:integer">
        <xsd:minInclusive value="0" />
        <xsd:maxInclusive value="10" />
    </xsd:restriction>
</xsd:simpleType>
```

La balise `simpleType` permet de définir un nouveau type simple. La balise `restriction` permet de référer à un type existant et d'y ajouter des contraintes. Ici, on définit une borne inférieure et supérieure avec les balises `minInclusive` et `maxInclusive`, que l'on appelle des facettes (*facets*). Voici quelques restrictions par type de base. La première permet de se restreindre aux valeurs `s1` à `sn`.

```
<xsd:restriction base="xsd:string">
    <xsd:enumeration value="s1" />
    ...
    <xsd:enumeration value="sn" />
</xsd:restriction>
```

On peut fixer la longueur d'une `string` comme suit:

```
<xsd:restriction base="xsd:string">
    <xsd:length value="10" />
</xsd:restriction>
```

ou bien en spécifiant un intervalle pour la longueur.

```
<xsd:restriction base="xsd:string">
    <xsd:minLength value="8" />
    <xsd:maxLength value="10" />
</xsd:restriction>
```

On peut aussi utiliser une expression régulière avec la balise `pattern`. Pour plus d'information, se référer à la norme des schéma XML sous la rubrique *facets*

Voici quelques restrictions pour un nombre.

```
<xsd:restriction base="xsd:decimal">
    <xsd:totalDigits value="8" />
    <xsd:fractionDigits value="10" />
</xsd:restriction>
```

On peut aussi utiliser les facettes suivantes pour un nombre.

pattern
enumeration
maxInclusive
maxExclusive
minInclusive
minExclusive

Il est aussi possible de définir un type complexe par extension. Voir la norme des schémas XML.

27.4.5 Type anonyme

Au lieu de définir un élément ou un attribut en référant à un type, on peut directement inclure le type dans l'élément ou l'attribut, sans associer un nom à ce type. Par exemple, on pourrait définir l'élément `biblio` comme suit:

```
<xsd:element name="biblio"/>
  <xsd:complexType>
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
          <xsd:element name="livre" type="Livre"  />
          <xsd:element name="membre" type="Membre"  />
      </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

On peut définir un attribut avec un type anonyme comme suit:

```
<xsd:attribute name="limitePret">
      <xsd:simpleType>
          <xsd:restriction base="xsd:integer">
              <xsd:minExclusive value="0" />
              <xsd:maxInclusive value="10" />
          </xsd:restriction>
      </xsd:simpleType>
</xsd:attribute>
```

27.5 Analyse syntaxique d'un document XML

Un analyseur syntaxique (*parser*) permet de vérifier si un document XML est bien formé et valide, s'il fait référence à un DTD ou un schéma XML. Naturellement, il permet aussi d'extraire des éléments du fichier XML pour en faire certains traitements (ex: mise à jour d'une base de données, affichage du contenu à l'écran, etc). Il existe une librairie dans le JDK pour faire l'analyse syntaxique d'un document XML: `javax.xml.parsers`. On y propose deux analyseurs : SAXP et DOM. SAXP effectue une lecture séquentielle du fichier XML et appelle des méthodes définies par l'utilisateur pour traiter chaque partie du fichier (balise, caractère, etc). DOM charge tout le fichier XML en mémoire vive sous la forme d'un arbre. SAXP est plus économe en mémoire vive, mais est plus difficile d'usage que DOM.

27.5.1 SAXP

SAXP est fondé sur les classes suivantes: `SAXParserFactory`, `SAXParser`, `DefaultHandler`. La classe `SAXParserFactory` permet de créer un *parser*. Un *parser* utilise un `DefaultHandler` pour déterminer le traitement à effectuer sur les éléments. Les instructions suivantes permettent de faire l'analyse syntaxique d'un document XML.

```
01:SAXParserFactory factory = SAXParserFactory.newInstance();
02:factory.setValidating(true);
03:SAXParser saxParser = factory.newSAXParser();
04:DefaultHandler handler = new ParserSimple();
05:saxParser.parse( new File(nomFichierXML), handler);
```

La ligne 02 indique que le `SAXParserFactory` doit valider le fichier XML, c'est-à-dire vérifier s'il est conforme au DTD référencé dans le fichier XML. La méthode `parse` (ligne 05) de la classe `SAXParser` prend comme paramètre le fichier XML à analyser et un objet qui est une instance de `DefaultHandler`. On utilise ici un objet de type `ParserSimple`, qui est une sous-classe de la classe `DefaultHandler`. La méthode `parse` appelle des méthodes de la classe `ParserSimple` selon les informations lues dans le fichier XML. Il faut donc définir dans la classe `ParserSimple` le traitement que l'on désire effectuer. Les méthodes suivantes sont appelées par la méthode `parse`.

- `public void startDocument()`

  Appelée au début de la lecture du document.

- `public void endDocument()`

  Appelée à la fin de la lecture du document.

- ```
  public void startElement(String namespaceURI,
                           String lName,
                           String qName,
                           Attributes attrs)
  ```

  Appelée à la fin de la lecture d'une balise de début d'un élément. Les deux premiers paramètres sont utilisés pour gérer les cas où un document XML réfère à plusieurs DTD (notion *XML name space*). Nous n'utiliserons pas cette notion dans le cadre du cours. Le paramètre `qname` contient le nom de l'élément. Le paramètre `attrs`, de type `Attributes`, contient la liste des attributs de l'élément. Les attributs y sont numérotés à partir de 0. La méthode `getLength()` retourne le nombre d'attributs. La méthode `getQName(i)` retourne le nom du $i^{\text{ième}}$ attribut de la liste; la méthode `getValue(i)` retourne sa valeur. Toutefois, comme il est possible de spécifier les attributs dans n'importe quel ordre, on ne peut connaître à l'avance l'index d'un attribut donné. On peut aussi accéder à la valeur d'un attribut de `attrs` en utilisant son nom : `getValue(nomAttribut)`

  Les attributs optionnels (`#IMPLIED`) qui ne sont pas mentionnés dans l'élément ne font naturellement pas partie de la liste. Les attributs optionnels ayant une valeur par défaut spécifiée dans le DTD et qui ne sont pas mentionnés dans l'élément sont inclus dans la liste avec leur valeur par défaut (d'où l'intérêt de spécifier une valeur par défaut dans le DTD).

- ```
  public void endElement(String namespaceURI,
                         String lName,
                         String qName)
  ```

  Appelée à la fin de la lecture d'une balise de fin d'un élément (même si cette balise est omise et remplacée par un `/>` dans la balise de début).

- `public void characters(char buf[], int offset, int len)`

  Appelée, une ou plusieurs fois, pour traiter le contenu d'un élément. Si le document XML fait référence à un DTD, seules les suites de caractères correspondant à des éléments de type `#PCDATA` sont retournés.

- Les méthodes suivantes sont appelées lorsque des erreurs sont détectées dans le document XML.

  ```
  public void warning(SAXParseException e)
  public void error(SAXParseException e)
  public void fatalError(SAXParseException e)
  ```

Pour des exemples de programmes utilisant la classe SAXParser, on peut se référer aux programmes `ParserSimple.java`, `ChargeurTableAttribut.java` et `ChargeurTableElement.java`. Le premier ne fait qu'afficher le contenu d'un document XML à l'écran. Vous pouvez l'utiliser pour vérifier si un document XML est bien formé. On l'appelle comme suit:

```
java ParserSimple <fichier>.xml [-v]
```

L'option -v permet de vérifier si un document XML est valide (s'il fait référence à un DTD). Le programme `ChargeurTableAttribut.java` permet de charger une table dans une BD relationnelle à partir du contenu d'un document XML. Pour chaque sous-élément de la racine, il insère un tuple dans la table correspondant au nom de l'élément. Les attributs du sous-élément représentent les valeurs des attributs du tuple. On l'appelle comme suit:

```
java ChargeurTableAttribut <fichier>.xml <serveur> <userid> <pw> [-v]
```

Le paramètre `<serveur>` peut prendre les valeurs suivantes (telles que définies dans la classe `ConnexionJDBC.java`; modifiez cette classe pour l'adapter à votre configuration locale).

- `sti` : BD oracle du STI;
- `oracleSL` : BD oracle Server installée localement;
- `oracleLite` : BD oracle Lite installée localement;
- autre : BD access installée localement.

Vous pouvez l'utiliser avec le document XML `biblio-attribut-v1.xml`. Le programme `ChargeurTableElement.java` est très similaire; il suppose toutefois que chaque attribut d'un tuple est défini par un sous-élément de l'élément représentant un tuple. On peut l'exécuter comme suit.

```
java ChargeurTableElement <fichier>.xml <serveur> <userid> <pw> [-v]
```

Vous pouvez l'utiliser avec le document XML `biblio-element-v1.xml`.

27.5.2 DOM

Le DOM permet de traduire un document XML en une représentation sous forme d'un arbre. La figure 27.14 représente le DOM du document XML de la figure 27.4; cette représentation graphique est obtenu à l'aide d'une autre librairie de Java, SWING. La racine de l'arbre représente un objet de type `Document`. Ce noeud comporte ensuite un fils pour le type de document et un autre pour le commentaire apparaissant avant l'élément racine. Il y a un noeud pour chaque commentaire. Finalement, il y a un noeud pour l'élément racine du document XML. Ce noeud a été sélectionné par l'utilisateur, ce qui fait que ses attributs sont affichés dans la partie droite de la fenêtre. Ce noeud comporte un fils pour chaque partie de son contenu. Le premier fils représente les caractères situés entre la balise `<expression etiquette="*">` (racine du document XML) et son premier sous-élément (la balise `<expression etiquette="+">`). On remarque ainsi une première différence avec le `SAXParser`. Ce dernier omet les caractères qui ne sont pas des `#PCDATA` tel que défini dans le DTD, alors que **tous** les caractères et **tous** les commentaires peuvent être inclus dans un DOM. On peut toutefois indiquer à l'analyseur syntaxique d'ignorer les noeuds de type texte (`Node.TEXT_NODE`) contenant seulement des espaces ou les noeuds de commentaires (`Node.COMMENT_NODE`). Le dernier fils dénote le texte situé entre la balise marquant la fin de

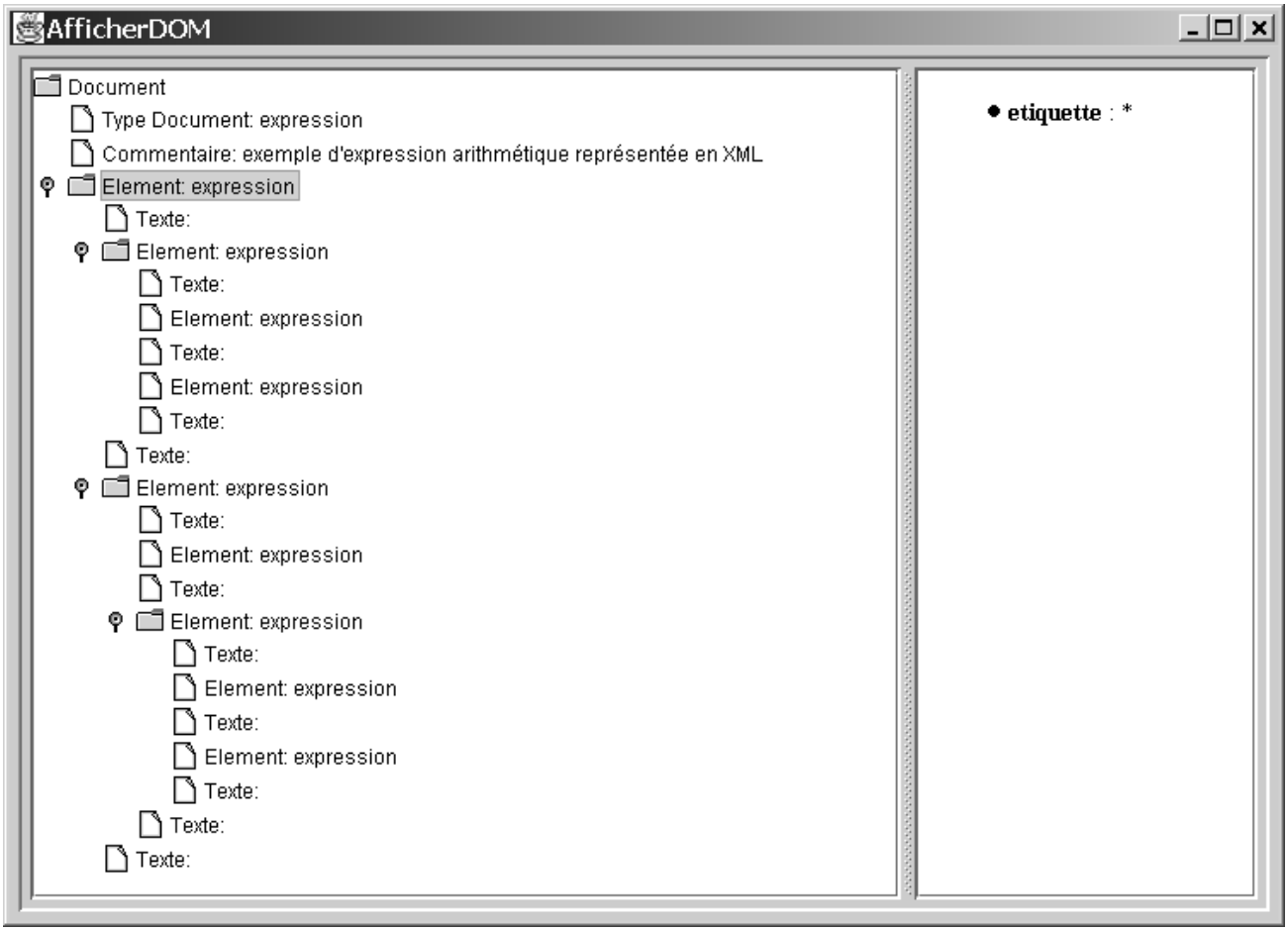

Figure 27.14: Le DOM correspondant au document XML de la figure 27.4

l'élément `<expression etiquette="+">` et la balise marquant le début de l'élément suivant (`<expression etiquette="-">`). Et ainsi de suite pour le reste de l'arbre. On peut afficher cet arbre en exécutant le programme suivant.

```
java AfficherDOM expression.xml
```

Le programme AfficherDOM illustre les principales méthodes du DOM. Le code source suivant permet de créer un DOM.

```
DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();
factory.setValidating(true);
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.parse( new File(argv[0]) );
```

On y utilise trois classes. Un objet de la classe `DocumentBuilderFactory` est utiliser pour créer un objet de type `DocumentBuilder`. Ce dernier peut faire analyse syntaxique d'un document XML et générer un objet de type `Document`. Pour indiquer à l'analyseur syntaxique d'ignorer les noeuds de type texte (`Node.TEXT_NODE`) contenant seulement des espaces ou les noeuds de commentaires (`Node.COMMENT_NODE`) en initialisant l'objet `factory` comme suit:

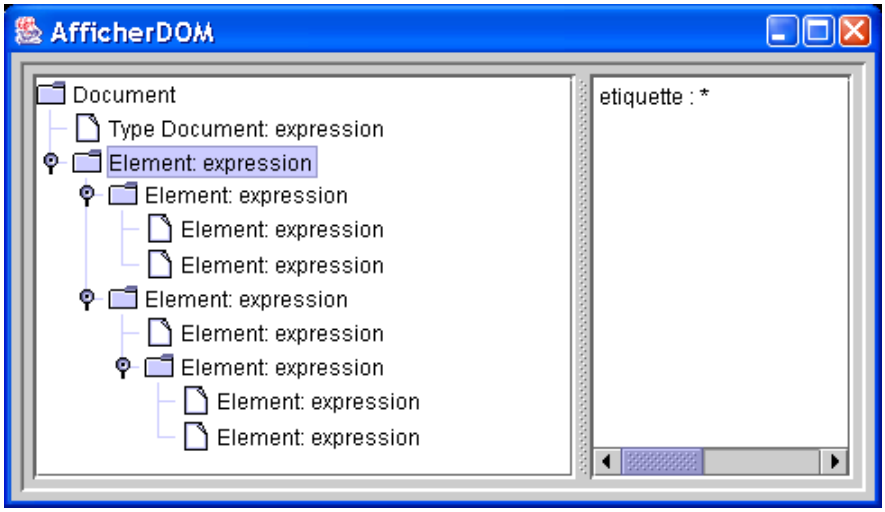

Figure 27.15: Le DOM correspondant au document XML de la figure 27.4 avec omission des commentaires et noeuds ne contenant que des espaces

```
DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();
factory.setValidating(true);
factory.setIgnoringElementContentWhitespace(true);
factory.setIgnoringComments(true);
Document document = builder.parse( new File(argv[0]) );
```

La figure 27.15 illustre le DOM du document XML de la figure 27.4 où les noeuds représentant les commentaires et le texte ne contenant que des espaces sont omis.

L'objet `document` est un arbre; l'interface `Document` est une sous-interface (extension) de l'interface `Node`. On peut naviguer à travers un arbre en utilisant les méthodes suivantes sur un objet de type `Node`.

- `getFirstChild()` : retourne le premier fils d'un noeud;
- `getNextSibling()` : retourne le frère d'un noeud;
- `getNodeType()` : retourne le type d'un noeud; la valeur retournée est l'une des constantes suivantes :
  - `Node.ATTRIBUTE_NODE` : attribut;
  - `Node.COMMENT_NODE` : commentaire;
  - `Node.DOCUMENT_NODE` : document;
  - `Node.DOCUMENT_TYPE_NODE` : type du document;
  - `Node.ELEMENT_NODE` : element;
  - `Node.TEXT_NODE` : texte.
- `getNodeName()` : retourne le nom d'un élément ou d'un attribut;

- `getNodeValue()` : retourne la valeur d'un attribut ou le texte d'un noeud représentant une suite de caractères ou d'un commentaire;
- `getAttributes()` : retourne un `NamedNodeMap`; cet objet est une liste d'attributs de l'élément correspondant au noeud.[2]

On utilise trois méthodes pour extraire les attributs d'un `NamedNodeMap`. Les attributs sont numérotés à partir de 0.

- `item(int i)` : retourne le `i`$^{\text{ième}}$ attribut de la liste; l'objet retourné est de type `Node`;
- `getLength()` : retourne le nombre d'attributs de la liste;
- `getNamedItem(String nom)` : retourne l'attribut de nom `nom` dans la liste; l'objet retourné est de type `Node`.

On peut extraire le nom et la valeur d'un attribut avec les méthodes `getNodeName` et `getNodeValue`. Si `n` est un objet de type `Node`, on extrait la valeur d'un de ses attributs nommé `att1` de la manière suivante:

```
Node n;
...
n.getAttributes().getNamedItem("att1").getNodeValue()
```

Comme cela est un peu long, il existe aussi une interface `Element`, qui est une extension de `Node`. Elle offre une méthode `getAttribute` permettant d'extraire directement la valeur d'un attribut d'un élément. Toutefois, comme on utilise généralement une variable de type `Node` pour parcourir un DOM, il faut au préalable faire un vérification de type (*cast*) pour passer d'une expression de type `Node` à `Element`.

```
((Element) n).getAttribute("att1")
```

La navigation à travers un arbre est illustrée par la figure 27.16. On note que, par souci de simplicité, les noeuds de type texte ont été supprimés de la figure. Les appels de méthode suivants illustre la navigation.

- n_1`.getFirstChild() ==` n_2
- n_1`.getNextSibling() == null`
- n_2`.getFirstChild() ==` n_4
- n_2`.getNextSibling() ==` n_3
- n_3`.getNextSibling() == null`
- n_4`.getFirstChild() == null`

[2]On remarque que les attributs d'un noeud **ne sont pas** représentés par des noeuds de l'arbre; on ne peut donc pas accéder aux attributs via la méthode `getFirstChild()`

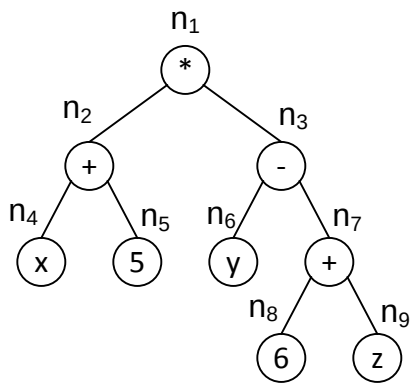

Figure 27.16: Le DOM correspondant au document XML de la figure 27.4

- n_4.getNextSibling() == n_5
- n_5.getNextSibling() == null
- n_1.getNodeName() == "expression"
- n_1.getAttributes().item(0).getNodeName() == "etiquette"
- n_1.getAttributes().item(0).getNodeValue() == "*"

Les programmes `ChargeurTableDOMAttribut.java` et `ChargeurTableDOMElement.java` illustrent l'usage du DOM pour charger une table d'une BD relationnelle. La structure attendue des documents XML est la même que pour les programmes `ChargeurTable-Attribut.java` et `ChargeurTableElement.java`, respectivement.

27.6 Génération d'un document XML

On peut créer un fichier XML de plusieurs façons. La plus élémentaire est de générer le fichier XML "*à la main*" en utilisant simplement les primitives d'écriture dans un fichier. Il faudra alors faire attention à la gestion des entités pour s'assurer que les symboles réservés pour la syntaxe XML sont correctement utilisés (voir figure 27.1). Pour ouvrir un fichier en écriture, on peut procéder comme suit:

```
FileOutputStream output = new FileOutputStream(<nomDuFichier>);
PrintStream out = new PrintStream(output);
```

On peut ensuite écrire dans `out` en utilisant les méthodes `print` et `println`.

On peut aussi créer un document XML à partir d'un DOM. Pour ce faire, il s'agit d'utiliser les transformations XML définies dans le package `javax.xml.transform` du JDK. La gestion des entités et de la syntaxe de XML en général (attribut, balise) est ainsi simplifiée. Voici comment on procède. On suppose ici que l'objet `document` contient le DOM à transformer en un fichier XML et que `out` dénote le fichier de sortie (voir définition de `out` paragraphe précédent).

```
TransformerFactory tFactory =
    TransformerFactory.newInstance();
Transformer transformer = tFactory.newTransformer();
transformer.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, <nomDuDTD>);
DOMSource source = new DOMSource(document);
StreamResult result = new StreamResult(out);
transformer.transform(source, result);
```

XSLT transforme automatiquement les caractères réservés (e.g., `<` et `&`) en entités dans le fichier de sortie `out`.

27.6.1 Construire un DOM

On peut créer un objet de type `Document` (i.e., un DOM) à partir d'un objet de type `DocumentBuilder`.

```
DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();
Document document =
    factory.newDocumentBuilder().newDocument();
```

Ensuite, on peut créer des noeuds en utilisant des méthodes de l'interface `Document`. Les noeuds sont reliés pour former un arbre en utilisant les méthodes de l'interface `Node`.

Les méthodes suivantes de l'interface `Document` permettent de créer les différents types de noeud d'un DOM.

- e = `document.createElement(<nomElement>)` : crée un élément e; cet élément doit ensuite être ajouté comme sous-élément d'un noeud x (de type `Element` ou `Document`) avec la méthode x`.appendChild(`e`)`. *Attention* : la méthode `createElement` n'ajoute pas le noeud créé comme un fils de `document`; il faut absolument utiliser une deuxième méthode, comme `appendChild`, pour insérer ce noeud dans l'arbre de `document`.

- c = `document.createComment(<texteDuCommentaire>)` : crée un noeud c contenant un commentaire; ce commentaire peut ensuite être ajouté comme contenu d'un autre noeud x (de type `Element` ou `Document`) avec la méthode x`.appendChild(`c`)`.

- t = `document.createTextNode(<texte>)` : crée un noeud t contenant seulement du texte; ce texte peut ensuite être ajouté comme contenu d'un autre noeud x (de type `Element` ou `Document`) avec la méthode x`.appendChild(`t`)`.

Pour ajouter un attribut a de valeur v à un élément x, on utilise la méthode

$$x\texttt{.setAttribute(}a\texttt{,}v\texttt{)}$$

Pour plus de détails, voir les exemples `EcrireXMLBiblioAttribut.java` et `EcrireXML-BiblioElement.java`.

Puisque XSLT transforme automatiquement les caractères réservés (e.g., `<` et `&`) en entités (e.g., `<` et `&`), on spécifie la valeur des attributs et le contenu textuel des éléments sans utiliser les entités, ce qui simplifie la création du DOM.

Chapter 27

Application WEB : Servlet et JSP

27.1 Application WEB avec Servlet

Une application WEB offre une interface utilisateur via un fureteur comme Microsoft Internet Explorer, Netscape ou Mozilla. L'utilisateur interagit avec l'application en utilisant des pages HTML générées dynamiquement par un serveur WEB. Ces pages comportent des formulaires qui permettent à l'utilisateur d'envoyer des requêtes au serveur. L'application s'exécute sur le serveur pour traiter ces requêtes. Nous utiliserons un serveur de type servlet-JSP. Dans cette section, nous décrivons les mécanismes offerts par ce type de serveur et nous illustrerons un système typique en utilisant l'exemple de la bibliothèque vu au chapitre 24.

27.1.1 Serveur servlet-JSP

Un serveur de type servlet-JSP permet de traiter les requêtes provenant des clients. Un client est un fureteur. Un serveur reçoit deux types de requêtes : exécution d'un servlet ou exécution d'une page JSP.

Un *servlet* est un programme Java; il est exécuté par le serveur pour traiter une requête d'un client; il reçoit en paramètre la requête du client; il produit en sortie une réponse. La réponse est une page web qui sera affichée par le client. Cette page est écrite en HTML et peut aussi contenir d'autres types d'instructions qui sont compréhensibles par le client (e.g., du Javascript, des macros Flash, etc). Dans le cadre du cours, nous n'utiliserons que du HTML.

Une page JSP est un mélange de code HTML et d'éléments de script JSP. Ces éléments de script comportent du code Java qui sera exécuté par le serveur pour générer une page HTML complète qui sera ensuite envoyée au client. Lorsqu'une page HTML est appelée la première fois, elle est traduite par le serveur en un servlet. Cette traduction est assez simple : le code HTML est converti en instructions Java qui écrivent ce code HTML sur le fichier de sortie envoyé au client. Les éléments de script JSP sont aussi traduits en Java.

Le *Java Community Process* produit des spécifications pour les serveurs servlet-JSP. Un premier document définit les servlets et un deuxième définit JSP. Il existe plusieurs organisations (entreprise et communauté source libre) qui produisent des serveurs satisfaisant ces spécifications. Dans le cadre du cours, nous utiliserons Tomcat, développé par le groupe

Apache, qui est du domaine public. Au niveau commercial, on retrouve WEBSphere, JBoss, Caucho Resin, Macromedia JRUN et plusieurs autres.

Un serveur gère l'accès aux *applications*. Dans les spécifications de servlet-JSP, une application est appelée un *context*. Chaque application est définie par un fichier de configuration `web.xml` qui décrit les ressources utilisées par l'application : les paramètres d'initialisation de l'application, l'identification des ressources externes comme les bases de données, les programmes a exécuter au démarrage et à la terminaison de l'application, les programmes à exécuter au démarrage et à la terminaison d'une session et les servlets composant l'application. Dans Tomcat, chaque application réside dans un sous-répertoire du répertoire `webapps`. Ce sous-répertoire contient toutes les composantes de l'application : pages JSP, pages HTML, servlets, fichier de configuration `web.xml`, images et autres objets référencés par les pages.

Toutes les classes utilisées pour les servlets sont définies dans le package `javax.servlet`. Ce package ne fait pas partie du JDK 1.6; il est toutefois fourni avec le serveur Tomcat.

27.1.2 Cycle de vie d'une application

Au démarrage, Tomcat initialise toutes les applications qu'il contient (i.e., tous les sous-répertoire de `webaps`) en créant un objet *context* (de l'interface `ServletContext`) pour chacune et en exécutant les programmes devant être exécutés au démarrage de chaque application, tel que spécifié dans le fichier de configuration `web.xml` de l'application.

À ce point, le serveur est prêt à répondre aux requêtes provenant des clients. À la réception d'une requête, le serveur exécute le servlet correspondant (rappel : une page JSP est traduite en un servlet lorsqu'elle est appelée la première fois).

Lors de la réception d'une requête, le serveur crée une *session* (de l'interface `HttpSession`) si le servlet y fait référence. Une session permet de conserver des informations relatives au dialogue entre un client et un serveur. Un *dialogue* est une suite de requêtes effectuées par un client. Une session est créée lors de la première requête d'un client. Elle est supprimée soit par un servlet (par exemple, lorsque le client effectue une requête pour se déconnecter de l'application) ou par le serveur au bout d'un certain temps d'inactivité (*timeout*).

La gestion des sessions n'est pas triviale, car le serveur n'a pas de contrôle sur le client. Généralement, un client accède à une application web en utilisant d'abord une page d'enregistrement ou d'ouverture de session (communément appelée un *login*). Par exemple, un site bancaire demande d'abord au client de s'identifier en entrant son identificateur (*userid*) et son mot de passe. Ensuite, il lui affiche un menu où il peut avoir accès à ses comptes de banques et à ses cartes de crédit. Toutefois, le serveur ne peut contraindre le client à faire une transaction de sortie (*logout*). Il faut donc un autre mécanisme pour fermer une session; le serveur utilise alors un compteur (*timer*) qui mesure le temps d'inactivité d'une session. Lorsque cette valeur atteint une certaine limite, la session est supprimée par le serveur. Avant de supprimer une session, le serveur peut appeler une méthode d'une classe définie dans l'application pour terminer proprement le traitement entrepris durant le dialogue avec le client (par exemple, fermeture des connexions avec la BD, mise à jour du profil du client dans la BD, sauvegarde dans la BD des informations relatives à la transaction commerciale en cours afin de permettre au client de poursuivre facilement la transaction en cours lors d'une prochaine session). Le travail à faire dépend de la nature de l'application.

Au minimum, il faut s'assurer que les ressources utilisées par l'application pour cette session sont libérées, afin de ne pas surcharger le serveur servlet-JSP ou le serveur de BD.

Comme une application peut recevoir des requêtes de plusieurs clients en même temps, le serveur permet de gérer plusieurs sessions en parallèle. Une session contient typiquement des informations sur le dialogue en cours avec un client. Par exemple, dans un site de commerce électronique, la session contient l'identificateur du client, les items à acheter choisis par le client jusqu'à ce point et toute autre information nécessaire par l'application pour assurer un dialogue simple avec l'utilisateur. Par exemple, un achat électronique peut nécessiter *plusieurs requêtes* afin de choisir les items à acheter, spécifier le mode d'expédition, de facturation, et de paiement, avant de conclure la transaction commerciale. L'application doit être conçue de manière à prévoir tout arrêt abrupte du dialogue avec le client (pour cause d'abandon par le client, de panne de réseau ou de panne de serveur).

Tel que mentionné initialement, la requête d'un client est traitée par un servlet. Le servlet peut identifier le client ayant soumis la requête en référant à la session associée à la requête. C'est le serveur qui détermine la session associée à une requête. Le servlet peut ensuite répondre correctement à la requête en utilisant les informations contenues dans la session et dans la requête.

Une requête contient généralement des *paramètres* qui sont entrés par l'utilisateur dans un *formulaire* de la page web d'où origine la requête. Un formulaire est exprimé dans un élément du fichier html. Ce formulaire indique le servlet (ou la page JSP) devant traiter la requête du côté du serveur.

27.1.3 Comparaison entre un programme mono-utilisateur et une application WEB

Nous présentons le concept d'une application WEB en le comparant à l'architecture client-serveur d'un programme d'application présentée au chapitre 24 et en illustrant avec le même exemple, soit un système de gestion de bibliothèque. Dans la suite ce chapitre, nous appellerons `Biblio24` le programme `Biblio` vu au chapitre 24.

Au chapitre 24, le programme `Biblio24` est appelé sur une ligne de commande par l'utilisateur. Ce programme ne traite qu'une seule transaction a la fois, toujours pour le même utilisateur. Il est conçu de manière à préserver l'intégrité de la base de données en utilisant les mécanismes de transactions du SGBD (commit et rollback). Plusieurs copies du programme `Biblio24` peuvent s'exécuter en parallèle, soit sur le même ordinateur ou sur des ordinateurs différents. Le programme `Biblio24` s'occupe de lire les transactions au clavier, d'extraire les paramètres et d'appeler une méthode d'un gestionnaire de transactions pour traiter la transaction. Il affiche ensuite une réponse très primitive au client.

Une application web opère dans un contexte différent. Son interface est une page web affichée par un fureteur. Le client soumet une requête au serveur servlet-JSP via le réseau internet. Le serveur servlet-JSP reçoit la requête et appelle une méthode du servlet pour traiter la requête et envoyer la réponse au client. Plusieurs clients peuvent utiliser l'application en parallèle. Il y a peu de travail à faire pour intégrer l'application `Biblio24` dans une application WEB. Il s'agit tout simplement de la convertir pour pouvoir traiter plusieurs requêtes en parallèle.

`Biblio24` utilise une seule connexion et une seule instance des gestionnaires de transaction et de tables; toutes ces variables sont déclarées comme des variables de classe de `Biblio24`. Si on veut traiter plusieurs clients en parallèle, il ne faut pas qu'une connexion soit partagée par deux clients lors de l'exécution d'une transaction; cela mettrait en peril l'intégrité des données. Une solution simple consiste à définir la connexion et les gestionnaires comme des *variables d'instance*; dans la suite du texte, nous appellerons `BiblioWeb` cette nouvelle classe inspirée de `Biblio24`. Dans l'application web, on créera une instance de `BiblioWeb` pour chaque client. C'est la solution que nous utiliserons, par souci de simplicité. Toutefois, cette solution n'utilise pas très efficacement les ressources du serveur, car deux clients pourraient partager une instance de `BiblioWeb`, à condition que deux transactions ne s'exécutent pas en même temps sur la même instance de `BiblioWeb`. Nous explorerons cette solution à la section 27.2

`Biblio24` effectue ses propres lectures au clavier pour obtenir les paramètres d'une transaction. Dans une application WEB, cette fonction sera attribuée au servlet. Le servlet fera l'extraction des paramètres de la requête, validera leur type, et appellera ensuite une méthode d'un gestionnaire de transactions de `BiblioWeb` pour effectuer la transaction. Pour afficher le résultat d'une transaction, on utilisera une page JSP. La force de JSP réside dans la capacité de spécifier facilement une page dynamique : la partie statique de la page est écrite en HTML standard; la partie dynamique utilise les éléments de script de JSP entrelacés avec du code HTML.

`Biblio24` n'a qu'un seul menu de base. À chaque transaction, il faut ré-entrer les mêmes informations, comme le numéro du membre. Dans une application WEB, on utilise généralement plusieurs menus pour naviguer à travers l'application. Les informations comme le numéro de membre ne sont entrées qu'une seule fois, lorsque le membre s'enregistre (*login*) auprès du système. Par la suite, l'application se rappelle du numéro de membre et l'utilise directement pour effectuer certaines transactions. La gestion des menus et la navigation sera aussi confiée aux servlets; l'affichage d'un menu sera confié aux pages JSP.

Le démarrage de l'application `Biblio24` est effectué sur une ligne de commande DOS ou UNIX. Elle se termine par l'exécution de `exit` par le client. Le démarrage de l'application WEB est effectué au démarrage du serveur servlet-JSP. Lors du démarrage de `Biblio24`, on ouvre une connexion et on crée les gestionnaires de connexions, de transactions et de tables. Dans une application WEB, ce travail peut être effectué de plusieurs façons. Précédemment, nous avons mentionné que nous utiliserions une instance de `BiblioWeb` pour chaque client. Cette instance sera créée lors du démarrage d'une session avec un client. Elle sera fermée lors de la fermeture de la session avec le client. Dans la section 27.2, nous verrons une alternative à cette approche.

27.1.4 Application

Une application dans un serveur servlet-JSP est représentée par un objet de l'interface `ServletContext`. Dans la documentation des serveurs servlet-JSP, on appelle aussi une application un *contexte*. Cet objet est créé par le serveur lors de son démarrage. Il est supprimé lors de l'arrêt du serveur.

Un contexte possède des *paramètres d'initialisation* et des *attributs*. Un paramètre d'initialisation a un nom, de type `String`, et une valeur, de type `String`. Le nom et la

valeur d'un paramètre sont définis dans le fichier `web.xml`. Les paramètres d'initialisation ne peuvent être modifiés; ils sont gérés par le serveur seulement. Un attribut d'un contexte a aussi un nom, de type `String`, et une valeur, de type `Object`. On crée un attribut avec la méthode `setAttribute(<nom>,<valeur>)`; on utilise aussi cette méthode pour le modifier. On obtient la valeur d'un attribut avec la méthode `getAttribute(<nom>)`. Comme elle est générale, elle retourne un objet de type `Object`; il faut donc faire un *cast* pour utiliser les propriétés de l'objet retourné. Si l'attribut cherché n'existe pas dans le contexte, la valeur `null` est retournée.

On accède à un contexte de trois manières.

- Au démarrage de l'application : si le fichier `web.xml` déclare un *listener* pour le contexte, c'est-à-dire une classe implémentant l'interface `ServletContextListener`, le serveur crée un objet de cette classe et appelle sa méthode `contextInitialized`. On spécifie donc dans cette méthode le traitement particulier que l'on désire effectuer au démarrage de l'application (e.g., allocation de connexions avec la BD, initialisation de certains objets globaux, etc). On sauvegarde ces différentes ressources dans des attributs du contexte.

- Dans un servlet : on appelle la méthode `getServletContext()` du servlet.

- À la terminaison de l'application : la méthode `contextDestroyed` du *listener* pour le contexte est appelée. On spécifie donc dans cette méthode le traitement particulier que l'on désire effectuer à la terminaison de l'application (e.g., fermer et désallouer les ressources acquises par l'application).

Pour voir un exemple de *listener* de contexte, consultez la classe `BiblioContextListener.java` de la page web du cours.

27.1.5 Servlet

Tel que mentionné précédemment, un servlet est exécuté par le serveur servlet-JSP pour traiter une requête d'un client. Le servlet doit donc :

- extraire les paramètres de la requête;

- valider les paramètres de la requête;

- faire le traitement (maj, calculs) requis pour la requête;

- formater en HTML la réponse à la requête.

Pour calculer la réponse à la requête, le servlet fera appel, en général, à un autre objet, par exemple, une instance de la classe `BiblioWeb`. Pour formatter la réponse en HTML, il fera appel en général à une page JSP, qui offre plus de flexibilité.

Un servlet est une classe qui est une extension de la classe `HttpServlet`. À la reception d'une requête d'un client, le serveur servlet-JSP appelle la méthode `doGet` ou la méthode `doPost` d'un objet du servlet indiqué dans la requête du client. Par exemple, le code HTML ci-dessous demande au serveur d'appeler la méthode `doPost` d'un servlet de la classe `Login`.

```
<FORM ACTION="Login" METHOD="POST">
```

Ce servlet est instancié par le serveur servlet-JSP; le serveur peut instancier autant de servlets qu'il le désire afin de répondre à plusieurs requêtes en parallèle. Le code HTML ci-dessous demande au serveur d'appeler la méthode `doGet`.

```
<FORM ACTION="Login" METHOD="GET">
```

Le code HTML ci-dessous utilise la méthode `doGet` car il s'agit d'un lien hypertexte ordinaire.

```
<a href="Logout">Sortir</a>
```

Pour le servlet, il n'y a pas de différence entre les deux méthodes. Pour le client, la méthode `GET` transmet la requête au serveur en un seul bloc, qui comprend le nom du servlet et les paramètres du formulaire HTML. La méthode `POST` transmet la requête en deux blocs; le nom du servlet dans le premier et les paramètres dans le deuxième; en conséquence, les paramètres ne sont pas affichés dans la barre d'adresse du fureteur. Par exemple, si la méthode utilisée est `GET`, le fureteur affiche l'adresse lorsque la réponse du serveur servlet-JSP est reçue.

```
http://localhost:8080/biblio/Login?userIdOracle=ift287_01&
motDePasseOracle=minou&serveur=sti
```

Pour un formulaire d'enregistrement, cela entraîne que le mot de passe entré apparaît dans la fenêtre, ce qui n'est pas désirable. Avec la méthode `POST`, le fureteur affiche simplement le nom du servlet appelé.

```
http://localhost:8080/biblio/Login
```

Toutefois, cela ne constitue pas un gage de confidentialité, puisque le mot de passe est envoyé sur le réseau sans être encrypté. Pour des communications sécurisées, il faut utiliser le protocole `https` au lieu du protocole `http`. Dans le cadre du cours, nous utiliserons simplement le protocole `http`. Généralement, la méthode `doGet` d'un servlet appellera simplement la méthode `doPost`, ou vice-versa.

Un servlet a des paramètres d'initialisation qui peuvent être spécifié dans le fichier `web.xml`. Leur valeur est définie à la création du servlet par le serveur. Dans le cadre du cours, nous n'utiliserons pas ces paramètres.

Pour voir des exemples de servlets, consultez la page web du cours; elle en comporte plusieurs (`Login`, `Logout`, `SelectionMembre`, `Emprunt`, et `ListePretMembre`).

27.1.6 Session

Une session dans un serveur servlet-JSP est un objet de l'interface `HttpSession`. Une session permet de sauvegarder des informations concernant le dialogue entre un client et une application. Une session est crée lors que le servlet le demande. Un servlet obtient une session en utilisant la méthode `getSession()` sur la requête reçue du client (le paramètre `request` de la méthode `doGet` ou la méthode `doPost`). Dans une page JSP, on réfère à la session en utilisant la variable `session` dans le code Java.

Une session possède des *attributs*. Ils sont gérés exactement de la même manière que les attributs d'un contexte. Un attribut a un nom, de type `String`, et une valeur, de type `Object`. On crée un attribut avec la méthode `setAttribute(<nom>,<valeur>)`; on utilise aussi cette méthode pour le modifier. On obtient la valeur d'un attribut avec la méthode `getAttribute(<nom>)`. Si l'attribut cherché n'existe pas dans la session, la valeur `null` est retournée.

On utilise les attributs d'une session pour sauvegarder les informations entrées par client que l'on désire conserver d'une page à l'autre dans le dialogue, afin de pas demander au client de les entrer une deuxième fois. On s'en sert aussi pour garder des informations sur l'état de la conversation avec le client (par exemple, pour afficher une barre d'historique, gérer les retours en arrière, donner des informations contextuelles).

La création d'une session dépend malheureusement du fureteur utilisé. Lorsqu'un serveur servlet-JSP interagit avec Mozilla, une seule session est créée, peut importe le nombre de fenêtres Mozilla démarrées par l'utilisateur. Avec Microsoft Internet Explorer, il peut y avoir plusieurs sessions (une session pour chaque fenêtre démarrée avec le menu DÉMARRER de Windows; il n'y a pas de nouvelle session pour les fenêtres démarrées par le menu FICHIER→NOUVELLE FENÊTRE d'Internet Explorer).

Le cycle de vie d'une session est le suivant.

- Création lors du premier appel de `getSession()`
- Suppression lorsque la méthode `invalidate()` de la session est appelée. Cette méthode peut être appelée par un servlet pour terminer un dialogue (par exemple lorsque le client fait un *logout*). Elle peut aussi être appelée par le serveur servlet-JSP lorsque la session a été inactive trop longtemps. Si le fichier `web.xml` déclare un *listener* de session, c'est-à-dire une classe implémentant l'interface `HttpSessionListener`, le serveur crée un objet de cette classe et appelle sa méthode `sessionDestroyed`. On spécifie donc dans cette méthode le traitement particulier que l'on désire effectuer à la terminaison de la session (e.g., mise à jour d'une BD pour enregistrer l'état du dialogue avec le client, afin de permettre une reprise rapide du dialogue lors de sa prochaine session, fermeture de connexions avec la BD, etc). La figure 27.1 illustre le cas où le serveur met fin à la session suite à l'expiration du délais d'inactivité.

Pour voir l'usage des sessions, consultez l'exemple de la page web du cours (plusiers classes et JSP l'utilisent). Pour un exemple de *listener* de session, voir `BiblioSessionListener.java`.

27.1.7 HTML

Le langage HTML (Hypertext Markup Language) est une application de SGML pour le web; ses instructions sont interprétables par un fureteur pour afficher du texte dans une forme prédéterminée. HTML comprend plusieurs types de balises. Nous n'énumérons ici que les plus importantes. Il existe plusieurs éditeurs HTML qui épargnent au concepteur de page web la gestion des détails de chaque balise. Les balises HTML, contrairement à celle de XML, sont indépendantes de la casse. De plus, certaines balises ne requièrent pas une balise de fin.

Voici les principales balises utilisées dans les exemples du cours.

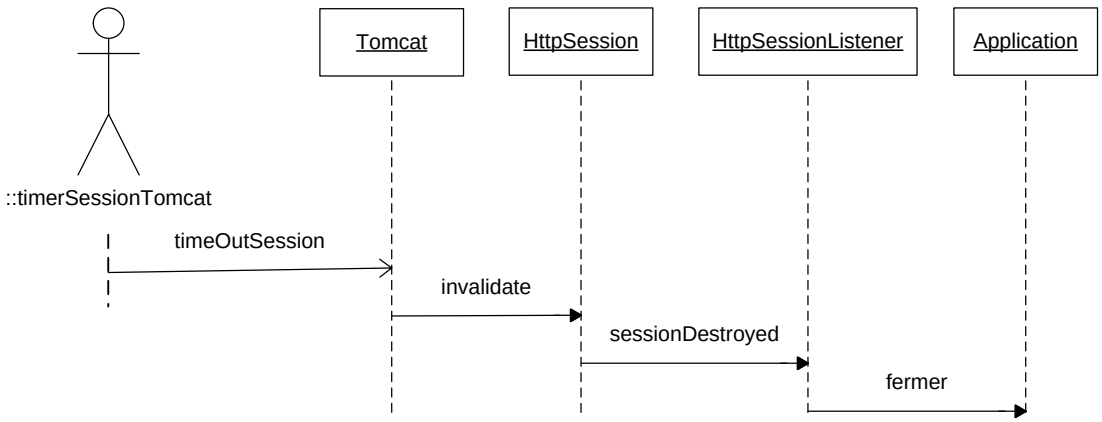

Figure 27.1: Diagramme de séquence pour le traitement l'expiration d'une session

- `<html> </html>` : c'est l'élément racine d'un document HTML.
- `<head> </head>` : contient de l'information générale pour la page; ces informations sont données par des sous-éléments comme `meta` et `title`. Elle n'apparaissent pas dans la fenêtre du fureteur.
- `<meta>` : permet d'indiquer différentes propriétés de la page comme le jeu de caractères utilisés, l'auteur, les mots clés pour l'indexation de la page par les moteurs de recherche, etc.
- `<title> </title>` : détermine le titre de la page; ce titre apparaît dans la barre supérieure de la fenêtre du fureteur lorsque la page est affichée.
- `<body> </body>` : contient le corps de la page; seuls les éléments du corps sont affichés dans la fenêtre du fureteur.
- `<h`i`> </h`i`>` : définit un titre de niveau i ($1 \leq i \leq 6$). Un titre s'affiche sur une seule ligne en caractères plus gros.
- `<p></p>` : définit un paragraphe. En HTML, un texte libre (sans balise) est mis en forme comme un seul paragraphe. Les lignes blanches et les espaces entre les mots sont éliminés par le fureteur lors de l'affichage. Pour faire des paragraphes, on utilise cette balise.
- `<br>` : dénote une fin de ligne dans un paragraphe.
- `<ul> </ul>` : définit une liste avec des •.
- `<ol> </ol>` : définit une liste numérotée.
- `<li> </li>` : définit un item d'une liste.
- `<table> </table>` : définit une table.
- `<tr> </tr>` : définit une ligne d'une table.

```
<html>
<head>
<meta content="text/html; charset=ISO-8859-1" http-equiv="content-type">
<meta name="author" content="Marc Frappier">
<title>Exemple simple de code HTML</title>
</head>
<body>
<h1> Exemple simple de page HTML</h1>
<!-- Voici la balise de debut de la liste -->
<ul>
    <li>
    Item 1 :
    Voici la date courante : 2004-04-04 09:40
    </li>
    <li>
    Item 2 :
    Voici la date courante : 2004-04-04 09:40
    </li>
</ul>
</body>
</html>
```

Figure 27.2: Une page HTML simple

- `<td> </td>` : définit une cellule d'une ligne.
- `<center> </center>` : permet de centrer un paragraphe, une image, une table, etc.
- `<a href="Logout">Sortir</a>` : permet de définir un lien hypertexte. Dans cet exemple, on appelle la page `Logout` en cliquant sur le texte `Sortir` dans la page; l'adresse de la page est donnée en mode relatif, ce qui facilite la maintenance des pages web. L'adresse complète de la page est obtenue en utilisant le l'adresse la page présentement affichée. Par exemple, supposons que `serveur/dir1/.../dirn/pagecourante.html` constitue l'adresse internet complète de la page présentement affichée dans le fureteur; en cliquant sur `Sortir`, le fureteur appelle la `Logout` page `serveur/dir1/.../dirn/Logout`.

  Si l'adresse donnée dans l'attribut `href` commence par un `/`, par exemple, `/chemin/Logout`, alors ce `/` dénote le répertoire racine du **serveur**. L'adresse complète de la page dénotée est `serveur/chemin/Logout`.

La figure 27.2 illustre un exemple simple de page HTML. La figure 27.3 donne le résultat de l'affichage de cette page dans Mozilla.

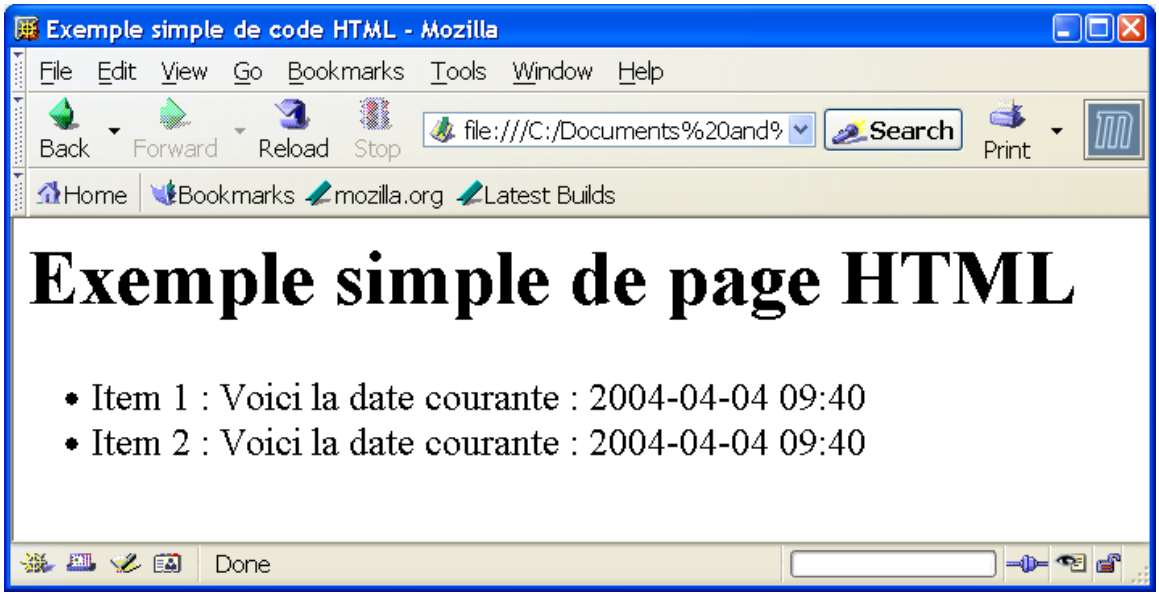

Figure 27.3: Le résultat de l'affichage de la page de la figure 27.2

27.1.8 Requête et formulaire HTML

Une requête dans un serveur servlet-JSP est un objet de l'interface `HttpServletRequest`. Elle est passée en paramètre à la méthode `doGet` et à la méthode `doPost`.

```
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
    throws ServletException, IOException
```

L'objet `request` dénote la requête; il contient les paramètres entrés par l'utilisateur dans un formulaire HTML. Il contient aussi une référence à la session dont elle fait partie. Finalement, une requête comporte des attributs, comme une session et un contexte, afin de faciliter la communication entre un servlet et une page JSP dédié au formatage de la réponse à une requête. On crée un attribut avec la méthode `setAttribute(<nom>,<valeur>)`; on utilise aussi cette méthode pour le modifier. On obtient la valeur d'un attribut avec la méthode `getAttribute(<nom>)`. Si l'attribut cherché n'existe pas dans la requête, la valeur `null` est retournée.

La gestion des paramètres d'une requête est généralement simple, mais parfois complexe pour les formulaires élaborés comportant des listes. Les paramètres d'une requêtes sont définis dans un formulaire HTML. Les paramètres sont transmis avec leur valeur par le client lors de la requête. Le servlet peut extraire un paramètre de la requête avec la méthode

```
String valeur = request.getParameter("nomDuParam")
```

Si le paramètre n'existe pas dans la requête, la valeur `null` est retournée. Si le paramètre contient un nombre ou une date, il faut alors convertir la valeur, de type `String`, en une valeur du bon type avec les méthodes de conversion de Java (e.g., `Interger.parseInt(valeur)`, `java.sql.Date.valueOf(valeur)`, `java.sql.Timestamp.valueOf(valeur)`).

Un formulaire HTML est défini par les balises `<FORM ...>` et `</FORM>`. Il existe plusieurs balises pour définir les champs d'un formulaire. Voici les plus importantes. Pour plus d'information, voir la page web du cours.

- bouton de soumission d'une requête (*submit*)

  ```
  <INPUT TYPE="SUBMIT" NAME="nomDuBouton"
  VALUE="texteAfficheSurBouton">
  ```

  Il peut y avoir plusieurs bouton *submit* dans une page. Lorsque l'utilisateur clique sur un de ces boutons, la requête est soumise au serveur avec les paramètres du formulaire. Pour le bouton *submit* sélectionné par l'utilisateur, le client retourne un paramètre de nom `nomDuBouton` et de valeur `texteAfficheSurBouton`. Les autres boutons ne sont pas retournés.

- champs

  ```
  <INPUT TYPE="TEXT" NAME="nomDuParam" VALUE="valeurInitiale">
  ```

  Le client retourne un paramètre de nom `nomDuParam`. La valeur par défaut du paramètre dans la page web est `valeurInitiale`.

- mot de passe

  ```
  <INPUT TYPE="PASSWORD" NAME="nomDuParam">
  ```

  Le client retourne un paramètre de nom `nomDuParam`. Lors de la saisie, les caractères ne sont pas affichés. À utiliser avec la méthode `POST`, pour éviter que le mot de passe apparaisse dans la barre d'adresse du fureteur.

  ```
  <FORM ACTION="Login" METHOD="POST">
  ```

- liste déroulante avec choix d'une seule valeur

  ```
  <SELECT NAME="nomDuParam">
    <OPTION VALUE="valeur">texte affiche dans la liste
    <OPTION VALUE="sti" SELECTED>sti
    <OPTION VALUE="postgres">postgres
    <OPTION VALUE="oracleSL">oracleSL
    <OPTION VALUE="oracleLite">oracleLite
    <OPTION VALUE="access">MS Access
  </SELECT>
  ```

  Le client retourne un paramètre de nom `nomDuParam` et la valeur choisie par l'utilisateur.

- bouton de radio

```
<INPUT TYPE="RADIO" NAME="nomDuParam" VALUE="valeur1">
<INPUT TYPE="RADIO" NAME="nomDuParam" VALUE="valeur2">
...
<INPUT TYPE="RADIO" NAME="nomDuParam" VALUE="valeurn">
```

Le client retourne un, et un seul, paramètre de nom `nomDuParam` et la valeur $\texttt{valeur}_x$ choisie par l'utilisateur (i.e., le bouton choisi).

- champs masqué

```
<INPUT TYPE="HIDDEN" NAME="nomDuParam" VALUE="valeur">
```

Un champs masqué n'apparaît pas à l'écran (toutefois, il est visible dans le source HTML). Le fureteur retourne *toujours* un paramètre pour un champs masqué; son nom est `nomDuParam` et sa valeur `valeur`. Utile pour identifier précisément un élément d'une liste dans laquelle l'utilisateur choisit un élément. On peut stocker dans un champs masqué les attributs de cet élément afin de pouvoir l'identifier complètement (par exemple, tous les attributs d'une clé primaire, afin de retrouver cet élément dans la base de données).

Un servlet peut appeler une page JSP pour formater la réponse à une requête en procédant comme suit.

```
RequestDispatcher dispatcher =
  request.getRequestDispatcher("/WEB-INF/listePretMembre.jsp");
dispatcher.forward(request, response);
```

Dans cet exemple, l'adresse de la page commence par un `/`. Le serveur servlet-JSP interprète ce symbole comme le répertoire principal de l'application web courante, et non par rapport à la racine du serveur, comme c'est le cas en HTML. Par souci de simplicité, nous utiliserons le répertoire `WEB-INF` pour stocker toutes les pages JSP de l'application, sauf celle de démarrage, qui apparaît dans le répertoire principal de l'application. Les pages de `WEB-INF` ne sont pas accessible aux clients. Seul un servlet de l'application y a accès. Cela permet de garder un certain contrôle sur l'accès aux pages JSP de l'application.

La figure 27.4 illustre par un diagramme de séquence les objets impliqués dans le démarrage d'une application et le traitement d'une requête.

27.1.9 JSP

Une page JSP est un mélange de code HTML et de code Java. Le serveur servlet-JSP traduit une page JSP en un servlet lors du premier appel d'une page. Par la suite, il appelle directement le servlet correspondant à la page.

Le code Java est exprimé entre des balises `<%` et `%>`. Le tableau ci-dessous décrit les principales balises.

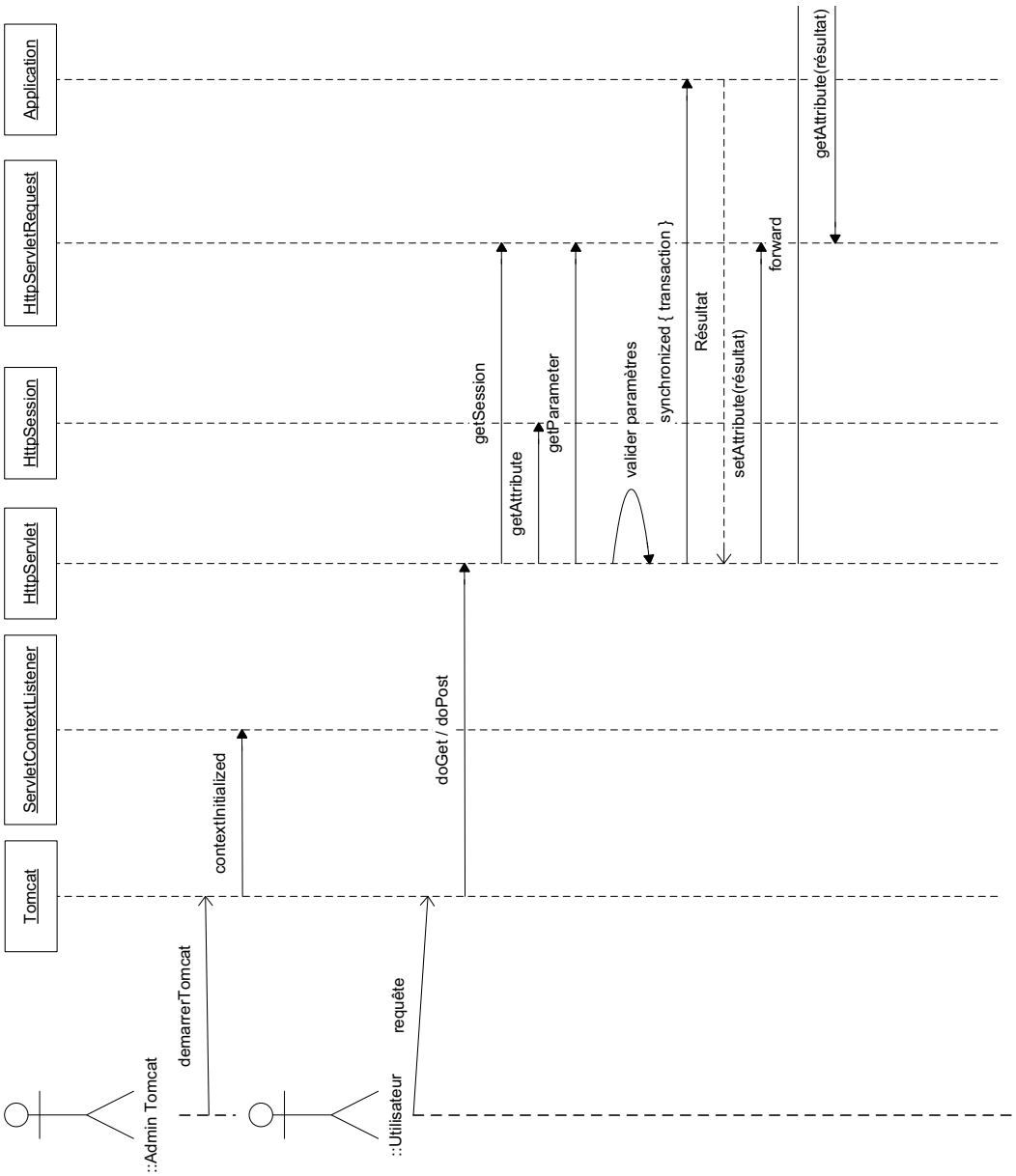

Figure 27.4: Diagramme de séquence du traitement d'une requête

`<%@ page import="`package$_1$, ..., package$_n$`" %>`	Déclaration des packages utilisés par le code Java de la page JSP.
`<%= expression java %>`	L'expression Java est évaluée et insérée dans le code HTML.
`<% code java %>`	Le code Java est exécuté; on peut entralacer du code java et du code HTML en entourant le code java des balises `<%` et `%>`. Voir exemples `messageErreur.jsp` et `listePretMembre.jsp` sur le site web du cours.
`<jsp:include page="/chemin/page.jsp" />`	Appel d'une autre page JSP.
`<%-- commentaire --%>`	Un commentaire JSP; ne fera pas partie du HTML généré.
`<!-- commentaire -->`	Un commentaire HTML. S'il contient des éléments de script JSP, ils seront exécutés
`<\%`	À utiliser pour générer `<%` dans la page HTML.
`%\>`	À utiliser pour générer `%>` dans la page HTML.
`\'`	Pour mettre un ' dans un attribut.
`\"`	Pour mettre un " dans un attribut.

La figure 27.5 présente un exemple simple de page JSP qui génère le code HTML présenté à la figure 27.2.

27.1.10 Contrôle du flot des pages

Un servlet peut être appelé directement par l'utilisateur sans nécessairement suivre l'ordre prévu par les pages. Par exemple, un utilisateur peut définir un signet sur une page et la rappeler, ou bien modifier les paramètres de ce signet. Il faut donc bien vérifier que l'appel d'un servlet est effectué dans des conditions valides. On peut stocker dans une session une variable d'état qui indique l'état actuel de la conversation et qui conditionne l'exécution d'un servlet. La figure 27.6 illustre comment ce contrôle de flot est fait pour l'exemple de la bibliothèque. La figure **??** l' illustre sous forme d'un automate de Mealy.

27.1.11 Traitement de la concurrence

Plusieurs clients peuvent utiliser une application web en même temps. En conséquence, un servlet peut être appelé pour traiter plusieurs requêtes en concurrence. Chaque requête est traitée par un *thread*. Un *thread* est un fil d'exécution. L'interpréteur Java peut exécuter plusieurs *thread* en même temps, un peu comme dans Windows où plusieurs applications peuvent exécuter en même temps. Par exemple, vous pouvez imprimer un document tout en continuant à utiliser un traitement de texte. Le système d'exploitation partage alors le temps d'exécution du processeur entre les différents processus (ou *thread*) concurrents.

```
<%-- Declaration des packages utilises --%>
<%@ page import="java.util.*,java.text.*" %>

<%-- Declaration de variables Java --%>
<%
SimpleDateFormat formatAMJHM = new SimpleDateFormat("yyyy-MM-dd HH:mm");
%>

<%-- Definition d'une page HTML contenant seulement une liste --%>
<html>
<head>
<meta content="text/html; charset=ISO-8859-1" http-equiv="content-type">
<meta name="author" content="Marc Frappier">
<title>Page d'exemple de code JSP</title>
</head>
<body>
<h1> Exemple simple de liste generee avec JSP</h1>
<!-- Voici la balise de debut de la liste -->
<ul>
<%-- Iteration pour creer 2 items dans la liste avec la date courante --%>
<%
  for (int i=1; i <= 2; i++)
    {
%>
<%-- L'execution de la boucle genere 2 items de la liste --%>
    <li>
    Item <%= i%> :
    Voici la date courante : <%= formatAMJHM.format(new Date())%>
    </li>
<%
    }
%>
</ul>
</body>
</html>
```

Figure 27.5: Une page JSP générant la page HTML de la figure 27.2

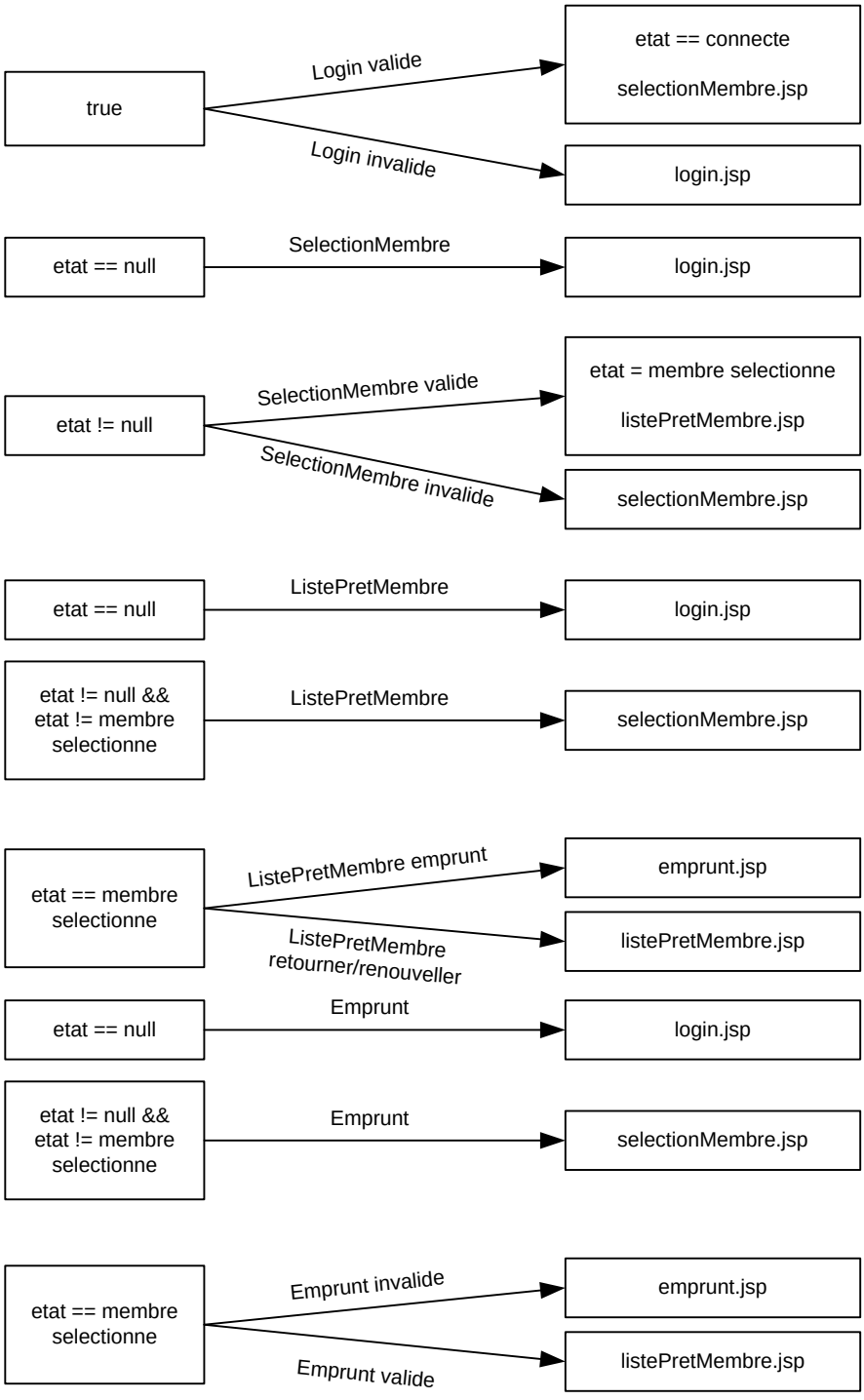

Figure 27.6: Diagramme états-transitions pour le contrôle de flot des pages

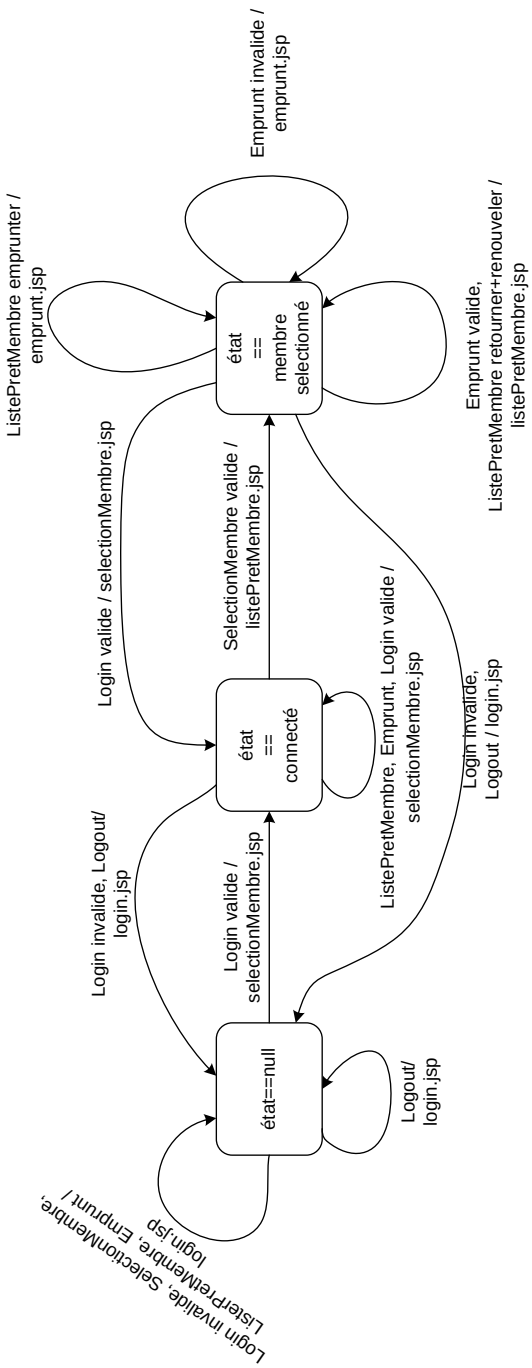

Figure 27.7: Automate de Mealy pour le contrôle de flot des pages

27.1.11.1 Accès concurrent aux ressources partagées

Le traitement en concurrence des requêtes par le serveur rend la programmation d'une application web un peu plus complexe que celle d'une application mono-utilisateur comme `Biblio24`. La difficulté réside dans l'accès concurrent en écriture à des ressources partagées. Les exemples typiques de ressources dans le cadre de ce cours sont des variables de classe, des variables d'instance et une connexion avec une base de données. Ce problème de partage de ressources est similaire au problème rencontré au chapitre 24 pour le traitement des transactions en concurrence dans un système de gestion de bases de données. Si deux *threads* accèdent en concurrence à la même ressource, les modifications de l'un peuvent interférer avec les modifications de l'autre, et résulter en une valeur incorrecte dans la ressource partagée. Par exemple, supposons que le serveur démarre deux *threads* en concurrence pour traiter deux requêtes de dépôt qui portent sur le même compte de banque. Chaque *thread* doit exécuter la méthode `depot` suivante :

```
void depot(Compte compte, int montant);
{
int solde = compte.getSolde();
compte.setSolde(solde + montant);
}
```

Les variables locales de la méthode `depot`, soient `montant` et `solde`, sont propres à chaque *thread*; elles ne sont donc pas partagées entre les *threads*. Le paramètre `compte` est aussi propre à chaque *thread*, sauf que c'est une référence à un objet. Dans notre exemple, on examine le cas où les deux *threads* réfèrent au même compte, donc les modifications faites par chaque *thread* s'effectuent sur le même objet compte. Comme le système d'exploitation partage le temps d'exécution entre les deux *threads*, la séquence d'événements suivante est possible. Supposons que le solde du compte est de 1 000 $ au début de l'exécution. Le premier *thread* exécute l'appel `depot(c,10)` et le deuxième `depot(c,500)`.

1. Le premier *thread* lit le solde du compte, qui est de 1 000 $. Sa variable locale `solde` vaut donc 1 000.

2. L'interpréteur Java suspend le premier *thread* et permet au second *thread* de s'exécuter. Le second *thread* lit le solde du compte, qui est de 1 000 $. Il affecte la valeur 1 500 au solde du compte et termine son exécution.

3. L'interpréteur Java permet au premier *thread* de poursuivre son exécution. Sa variable locale solde vaut toujours 1 000, même si la valeur actuelle du solde du compte est maintenant de 1 500 $. Il affecte la valeur 1 010 au solde du compte pour compléter son dépôt.

La valeur finale du solde est donc 1 010 $. Si les deux *threads* s'étaient exécuté l'un après l'autre au lieu de s'exécuter en concurrence, la valeur du solde serait de 1 510 $. Le propriétaire du compte a donc perdu 500 $ à cause de la concurrence entre les deux *threads*.

Notons que la réécriture de la méthode `depot` en utilisant un seul énoncé comme suit n'est pas une solution.

```
void depot(Compte compte, int montant);
{
compte.setSolde(compte.getSolde() + montant);
}
```

Le système d'exploitation peut très bien interrompre l'exécution après `getSolde()` et avant l'addition.

Il ne faut pas sous-estimer la probabilité de l'occurrence du problème de mise à jour en concurrence. Il est arrivé à quelques reprises dans le passé que certains sites de commerce électronique ont facturé des transactions de clients différents sur la même carte de crédit.

27.1.11.2 Mécanisme Java de contrôle de la concurrence

Java propose une solution très simple pour gérer la concurrence : le verrou (*lock*). Chaque objet possède un attribut que l'on appelle un verrou. L'énoncé

```
synchronized (o) { <traitement> }
```

tente d'obtenir le verrou de l'objet `o`. Si le verrou est déjà alloué à un autre *thread*, l'énoncé `synchronized` attend que ce *thread* libère le verrou. Lorsque le verrou est obtenu, l'exécution de `traitement` débute. À la fin de `traitement`, le verrou est libéré.

Ainsi, les *threads* ayant à partager un objet en écriture peuvent utiliser l'énoncé `synchronized` pour s'assurer l'exclusivité d'un objet pendant un traitement. Pour que cette exclusivité soit garantie, il faut que chaque *thread* utilise l'énoncé `synchronized`; un *thread* qui n'utilise pas l'énoncé `synchronized` peut très bien modifier l'objet. Un verrou n'empêche pas la modification d'un objet; un verrou influence seulement l'exécution de l'énoncé `synchronized`[1]; le `<traitement >` démarre seulement quand le verrou est obtenu.

Le problème du dépôt dans un compte de banque est facilement résolu par l'usage de `synchronized`. Voici la version améliorée de cette méthode.

```
void depot(Compte compte, int montant);
{
synchronized (compte)
  {
  int solde = compte.getSolde();
  compte.setSolde(solde + montant);
  }
}
```

Bien entendu, les autres méthodes qui modifient le compte devraient aussi utiliser `synchronized`, si elle peuvent s'exécuter en concurrence avec `depot`; sinon le problème de corruption de données pourrait survenir.

[1] L'énoncé `wait` utilise aussi le verrou d'un objet.

27.1.11.3 Ressources partagées par les servlets

La spécification de JCP pour les serveurs de servlet-JSP indique qu'un serveur peut traiter les requêtes en concurrence en utilisant soit la même instance de servlet dans chaque *thread*, soit plusieurs instances du même servlet. Les attributs d'une instance de servlet peuvent donc être accédés en concurrence par plusieurs *threads*. La session et le contexte sont aussi accessibles en concurrence. Par contre, la requête traitée par un servlet lui est propre.

Comme la session est accessible en concurrence par plusieurs servlet, notre programme `BiblioWeb` doit s'assurer que deux requêtes provenant du même client (donc de la même session) n'utilise pas la connexion avec la base de données en concurrence. Ainsi, dans le traitement d'une requête, chaque servlet de `BiblioWeb` utilise l'énoncé `synchronized` avant de faire une transaction sur une instance de `biblio`.

Voir `Emprunt.java` et `ListePretMembre.java` pour un exemple détaillé.

27.2 Partage de connexions

Le programme `BiblioWeb` créée une instance de `Biblio` pour chaque utilisateur. Si 500 personnes se connectent en même temps au système, il y aura alors 500 connexions ouvertes dans la base de données. Pourtant, bien peu de connexions seront utilisées en même temps, car un utilisateur entre peu de transaction à la minute. En pratique, pour un instant donné, probablement que seulement 20 connexions seront utilisées, les autres étant en attente d'une transaction de l'utilisateur. Ainsi, 500 connexions sont ouvertes, alors que seulement 20 au maximum sont utilisées en même temps.

Pour décharger le serveur de base de données, on peut partager les connexions entre plusieurs clients. On doit toutefois respecter les conditions suivantes :

- une requête d'un client a l'exclusivité d'une connexion durant l'exécution de sa transaction;
- sa transaction se termine par un `commit` ou un `rollback`.

Notre programme `Biblio` utilise des `PreparedStatement` pour effectuer ses requêtes, par souci d'efficacité. Un `PreparedStatement` est associé à une connexion. Si on partage les connexions, il faut donc s'assurer d'utiliser les `PreparedStatement` associés à la connexion. En conséquence, le plus simple est de partager les instances de `Biblio`, chaque instance de `biblio` assurant la correspondance `PreparedStatement` et connexion.

Plusieurs serveurs servlet-JSP offrent des gestionnaires de partage de ressources. Il est préférable de les utiliser, car le développement d'un tel gestionnaire n'est pas simple.

Voici le scénario typique pour le partage de ressources entre des clients d'une application.

- Au démarrage de l'application par le serveur servlet-JSP, le `ContextListener` de l'application alloue un certain nombre de ressources (par exemple, connexions ou instances de `Biblio`). Ce nombre peut être spécifié dans le fichier `web.xml`), ainsi que d'autres paramètres comme l'adresse du serveur de BD, le pilote JDBC à utiliser, etc.

- Lors du traitement d'une requête, un servlet demande une ressource. Il utilise cette ressource pour traiter la requête. Finalement, il libère la ressource lorsque le traitement est terminé.
- À la terminaison de l'application, le `ContextListener` ferme correctement toutes les ressources créés.

Le partage de ressources est assez complexe et sujet à plusieurs erreurs de programmation. Voici les erreurs typiques.

- Un servlet oublie de libérer une ressource à la fin de son traitement.
- Un servlet oublie de fermer les objets associés à la ressource obtenue (e.g., les `Statement`, les `ResultSet`).
- Le serveur de BD ferme une connexion (suite à une erreur fatale interne, un problème de communication, un trop long délais d'inactivité, etc).

Un gestionnaire efficace de partage de ressources doit détecter ces erreurs et tenter de les corriger.