EXERCICE JAVA

Thèmes : Bases de données, accès JDBC, classes et interfaces Java, architectures 2 couches
Niveau : intermédiaire

Lectures conseillées :

- ✗ **[1] : Apprentissage du langage Java** [http://tahe.developpez.com/java/cours]
- ✗ **[2] : Spring IoC** [http://tahe.developpez.com/java/springioc]

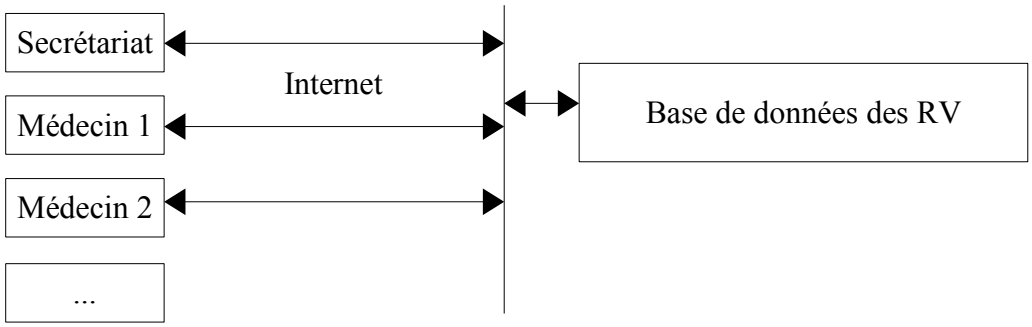
Le texte est long parce qu'il est nécessaire de détailler l'environnement dans lequel prennent place les questions. Ces questions ne sont pas particulièrement difficiles. Le sujet mesure donc aussi bien votre capacité à comprendre un problème qu'à le traiter.

1 Le problème

Une société de services en informatique [ISTIA-IAIE] désire proposer un service de prise de rendez-vous. Le premier marché visé est celui des médecins travaillant seuls. Ceux-ci n'ont en général pas de secrétariat. Les clients désirant prendre rendez-vous téléphonent alors directement au médecin. Celui-ci est donc dérangé fréquemment au cours d'une journée ce qui diminue sa disponibilité à ses patients. La société [ISTIA-IAIE] souhaite leur proposer un service de prise de rendez-vous fonctionnant sur le principe suivant :

- un service secrétariat assure les prises de RV pour un grand nombre de médecins. Ce service peut être réduit à une unique personne. Le salaire de celle-ci est mutualisé entre tous les médecins utilisant le service de RV.
- le service secrétariat et tous les médecins sont reliés à Internet
- les RV sont enregistrés dans une base de données centralisée, accessible par Internet, par le secrétariat et les médecins
- la prise de RV est normalement faite par le secrétariat. Elle peut être faite également par les médecins eux-mêmes. C'est le cas notamment lorsqu'à la fin d'une consultation, le médecin fixe lui-même un nouveau RV à son patient.

L'architecture du service de prise de RV est le suivant :

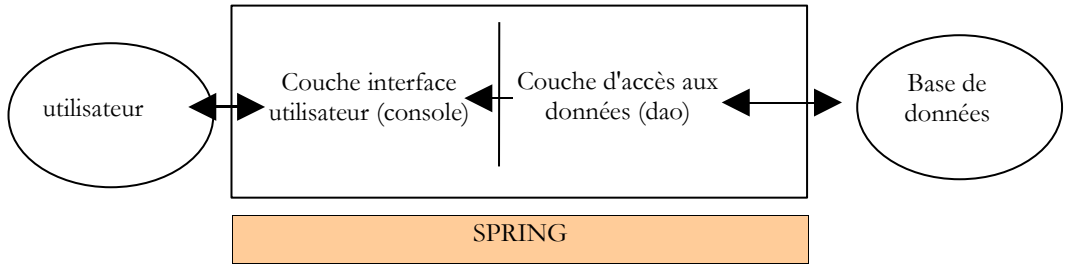


Les médecins gagnent en efficacité s'ils n'ont plus à gérer les RV. S'ils sont suffisamment nombreux, leur contribution aux frais de fonctionnement du secrétariat sera faible.

La société [ISTIA-IAIE] décide de confier à un stagiaire l'élaboration d'une première maquette du service, une application console basique. C'est cette maquette que nous nous proposons de réaliser ici.

2 Les éléments de l'application

L'application [RdvMedecins] aura une architecture à deux couches :

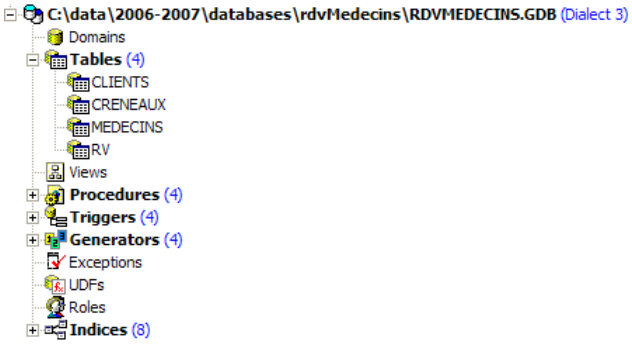


- les deux couches sont rendues indépendantes grâce à l'utilisation d'interfaces Java,
- l'intégration des couches est réalisée par Spring.

La couche [console] est destinée à être remplacée ultérieurement par une couche [web].

2.1 La base de données

La base de données qu'on appellera [RDVMEDECINS] est une base de données Firebird avec quatre tables :



2.1.1 La table [MEDECINS]

Elle contient des informations sur les médecins gérés par l'application [RdvMedecins].

PK	Field Name	Field Type	Domain	Size
1	ID	INTEGER		
	NOM	VARCHAR		30
	PRENOM	VARCHAR		30
	TITRE	VARCHAR		5

ID	NOM	PRENOM	TITRE
1	PELISSIER	Marie	Mme
2	BROMARD	Jacques	Mr
3	JANDOT	Philippe	Mr
4	JACQUEMOT	Justine	Melle

- ID : n° identifiant le médecin - clé primaire de la table
- NOM : le nom du médecin
- PRENOM : son prénom
- TITRE : son titre (Melle, Mme, Mr)

2.1.2 La table [CLIENTS]

Les clients des différents médecins sont enregistrés dans la table [CLIENTS] :

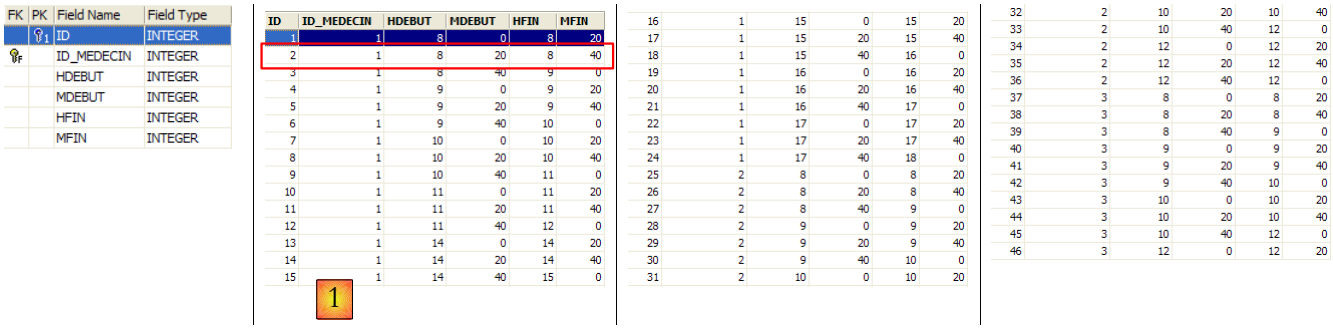
PK	Field Name	Field Type	Domain	Size	Sc
1	ID	INTEGER			
	NOM	VARCHAR		30	
	PRENOM	VARCHAR		30	
	TITRE	VARCHAR		5	

ID	NOM	PRENOM	TITRE
1	MARTIN	Jules	Mr
2	GERMAN	Christine	Mme
3	JACQUARD	Jules	Mr
4	BISTROU	Brigitte	Melle

- ID : n° identifiant le client - clé primaire de la table
- NOM : le nom du client
- PRENOM : son prénom
- TITRE : son titre (Melle, Mme, Mr)

2.1.3 La table [CRENEAUX]

Elle liste les créneaux horaires où les RV sont possibles :



FK	PK	Field Name	Field Type
	1	ID	INTEGER
F		ID_MEDECIN	INTEGER
		HDEBUT	INTEGER
		MDEBUT	INTEGER
		HFIN	INTEGER
		MFIN	INTEGER

ID	ID_MEDECIN	HDEBUT	MDEBUT	HFIN	MFIN
1	1	8	0	8	20
2	1	8	20	8	40
3	1	8	40	9	0
4	1	9	0	9	20
5	1	9	20	9	40
6	1	9	40	10	0
7	1	10	0	10	20
8	1	10	20	10	40
9	1	10	40	11	0
10	1	11	0	11	20
11	1	11	20	11	40
12	1	11	40	12	0
13	1	14	0	14	20
14	1	14	20	14	40
15	1	14	40	15	0
16	1	15	0	15	20
17	1	15	20	15	40
18	1	15	40	16	0
19	1	16	0	16	20
20	1	16	20	16	40
21	1	16	40	17	0
22	1	17	0	17	20
23	1	17	20	17	40
24	1	17	40	18	0
25	2	8	0	8	20
26	2	8	20	8	40
27	2	8	40	9	0
28	2	9	0	9	20
29	2	9	20	9	40
30	2	9	40	10	0
31	2	10	0	10	20
32	2	10	20	10	40
33	2	10	40	12	0
34	2	12	0	12	20
35	2	12	20	12	40
36	2	12	40	12	0
37	3	8	0	8	20
38	3	8	20	8	40
39	3	8	40	9	0
40	3	9	0	9	20
41	3	9	20	9	40
42	3	9	40	10	0
43	3	10	0	10	20
44	3	10	20	10	40
45	3	10	40	12	0
46	3	12	0	12	20

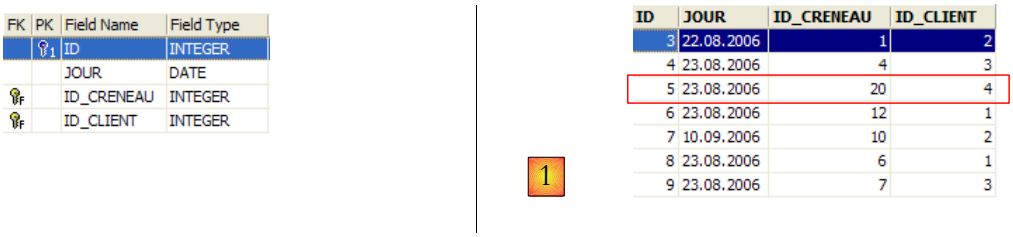
1

- ID : n° identifiant le créneau horaire - clé primaire de la table (ligne 8)
- ID_MEDECIN : n° identifiant le médecin auquel appartient ce créneau – clé étrangère sur la colonne MEDECINS(ID).
- HDEBUT : heure début créneau
- MDEBUT : minutes début créneau
- HFIN : heure fin créneau
- MFIN : minutes fin créneau

La seconde ligne de la table [CRENEAUX] (cf [1] ci-dessus) indique, par exemple, que le créneau n° 2 commence à 8 h 20 et se termine à 8 h 40 et appartient au médecin n° 1 (Mme Marie PELISSIER).

2.1.4 La table [RV]

Elle liste les RV pris pour chaque médecin :



FK	PK	Field Name	Field Type
	1	ID	INTEGER
		JOUR	DATE
F		ID_CRENEAU	INTEGER
F		ID_CLIENT	INTEGER

ID	JOUR	ID_CRENEAU	ID_CLIENT
3	22.08.2006	1	2
4	23.08.2006	4	3
5	23.08.2006	20	4
6	23.08.2006	12	1
7	10.09.2006	10	2
8	23.08.2006	6	1
9	23.08.2006	7	3

1

- ID : n° identifiant le RV de façon unique – clé primaire
- JOUR : jour du RV
- ID_CRENEAU : créneau horaire du RV - clé étrangère sur le champ [ID] de la table [CRENEAUX] – fixe à la fois le créneau horaire et le médecin concerné.
- ID_CLIENT : n° du client pour qui est faite la réservation – clé étrangère sur le champ [ID] de la table [CLIENTS]

Cette table a une contrainte d'unicité sur les valeurs des colonnes jointes (JOUR, ID_CRENEAU) :

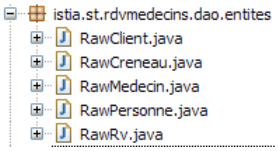
```
ALTER TABLE RV ADD CONSTRAINT UNQ1_RV UNIQUE (JOUR, ID_CRENEAU);
```

Si une ligne de la table[RV] a la valeur (JOUR1, ID_CRENEAU1) pour les colonnes (JOUR, ID_CRENEAU), cette valeur ne peut se retrouver nulle part ailleurs. Sinon, cela signifierait que deux RV ont été pris au même moment pour le même médecin. D'un point de vue programmation Java, le pilote JDBC de la base lance une *SQLException* lorsque ce cas se produit.

La ligne 3 (cf [1] ci-dessus) signifie qu'un RV a été pris pour le créneau n° 20 et le client n° 4 le 23/08/2006. La table [CRENEAUX] nous apprend que le créneau n° 20 correspond au créneau horaire 16 h 20 - 16 h 40 et appartient au médecin n° 1 (Mme Marie PELISSIER). La table [CLIENTS] nous apprend que le client n° 4 est Melle Brigitte BISTROU.

2.2 Les objets de l'application

Les objets manipulés par la couche [dao] de l'application [RdvMedecins] reflètent les lignes des tables de la base de données. Ce sont les suivants :



2.2.1 Classe [RawPersonne]

La classe [Personne] représente une ligne de la table [MEDECINS] ou de la table [CLIENTS]. Son code est le suivant :

```
1.  package istia.st.rdvmedecins.dao.entites;
2.
3.  public class RawPersonne {
4.          // caractéristiques d'une personne
5.          private int id;
6.          private String titre;
7.          private String nom;
8.          private String prenom;
9.
10.         // constructeur par défaut
11.         public RawPersonne(){
12.
13.         }
14.
15.         // constructeur avec paramètres
16.         public RawPersonne(int id, String titre, String nom, String prenom){
17. ...
18.         }
19.
20.         // constructeur par recopie
21.         public RawPersonne(RawPersonne personne){
22. ...
23.         }
24.
25.         // toString
26.         public String toString(){
27.                 return "["+id+","+titre+","+prenom+","+nom+"]";
28.         }
29.
30.         // setters and getters
31. ...
32.
33. }
```

- ligne 5 : le n° de la personne
- ligne 6 : son titre (Mr, Mme, Melle)
- ligne 7 : son nom
- ligne 8 : son prénom
- lignes 16-18 : le constructeur permettant d'initialiser une instance [Personne] avec les quatre informations nécessaires
- lignes 21-23 : le constructeur permettant d'initialiser une instance [Personne] avec les valeurs (id, titre, nom, prenom) d'une autre personne.
- lignes 26-28 : la méthode [toString] de la classe.
- lignes 30 et au-delà : les méthodes get / set associées à chacun des champs privés de la classe.

2.2.2 Classe [RawMedecin]

La classe [RawMedecin] représente une ligne de la table [MEDECINS]. Son code est le suivant :

```
1.  package istia.st.rdvmedecins.dao.entites;
2.
3.  public class RawMedecin extends RawPersonne{
4.          // constructeur par défaut
5.          public RawMedecin(){
6.          }
7.
8.          // constructeur avec paramètres
9.          public RawMedecin(int id, String titre, String nom, String prenom){
10.                 // parent
11.                 super(id,titre,nom,prenom);
12.         }
13.
14.         // constructeur par recopie
15.         public RawMedecin(RawMedecin medecin){
16.                 // parent
17.                 super(medecin);
18.         }
19.
20. }
```

- ligne 3 : la classe dérive de la classe [Personne] et en a donc toutes les caractéristiques.

2.2.3 Classe [RawClient]

La classe [RawClient] représente une ligne de la table [CLIENTS]. Son code est le suivant :

```
1.  package istia.st.rdvmedecins.dao.entites;
2.
3.  public class RawClient extends RawPersonne{
4.          // constructeur par défaut
5.          public RawClient(){
6.          }
7.
8.          // constructeur avec paramètres
9.          public RawClient(int id, String titre, String nom, String prenom){
10.                 // parent
11.                 super(id,titre,nom,prenom);
12.         }
13.
14.         // constructeur par recopie
15.         public RawClient(RawClient client){
16.                 // parent
17.                 super(client);
18.         }
19. }
```

- ligne 3 : la classe dérive de la classe [Personne] et en a donc toutes les caractéristiques.

2.2.4 Classe [RawCreneau]

La classe [RawCreneau] représente une ligne de la table [CRENEAUX]. Son code est le suivant :

```
1.  package istia.st.rdvmedecins.dao.entites;
2.
3.  public class RawCreneau {
4.
5.          // caractéristiques
6.          private int id;
7.          private int idMedecin;
8.          private int hDebut;
9.          private int mDebut;
10.         private int hFin;
11.         private int mFin;
12.
13.         // constructeur par défaut
14.         public RawCreneau() {
15.
16.         }
17.
18.         // constructeur avec paramètres
19.         public RawCreneau(int id, int idMedecin, int hDebut, int mDebut, int hFin,
20.                         int mFin) {
21. ...
22.         }
23.
24.         // constructeur par recopie
25.         public RawCreneau(RawCreneau creneau) {
26. ...
27.         }
28.
29.         // toString
30.         public String toString() {
31.                 return "[" + id + "," + idMedecin + "," + hDebut + ":" + mDebut + ","
32.                                 + hFin + ":" + mFin + "]";
33.         }
34.
35.         // setters - getters
36. ...
37. }
```

- ligne 6 : le n° du créneau
- ligne 7 : le n° du médecin auquel appartient le créneau
- ligne 8 : l'heure de début
- ligne 9 : les minutes de début de créneau
- ligne 10 : l'heure de fin

- ligne 11 : les minutes de fin de créneau
- lignes 19-22 : le constructeur permettant d'initialiser une instance [RawCreneau] avec les six informations nécessaires
- lignes 25-27 : le constructeur permettant d'initialiser une instance [RawCreneau] avec les informations d'un autre objet [RawCreneau]
- lignes 30-33 : la méthode [toString] de la classe.
- lignes 35 et au-delà : les méthodes get / set associées à chacun des champs privés de la classe.

2.2.5 Classe [RawRv]

La classe [RawRv] représente une ligne de la table [RV]. Son code est le suivant :

```
1.  package istia.st.rdvmedecins.dao.entites;
2.
3.  import java.text.SimpleDateFormat;
4.  import java.util.Date;
5.
6.  public class RawRv {
7.          // caractéristiques
8.          private int id;
9.          private Date jour;
10.         private int idClient;
11.         private int idCreneau;
12.
13.         // constructeur par défaut
14.         public RawRv() {
15.
16.         }
17.
18.         // constructeur avec paramètres
19.         public RawRv(int id, Date jour, int idClient, int idCreneau) {
20. ...
21.         }
22.
23.         // constructeur par recopie
24.         public RawRv(RawRv rv) {
25. ...
26.         }
27.
28.         // toString
29.         public String toString() {
30.                 return "[" + id + ","
31.                                 + new SimpleDateFormat("dd/MM/yyyy").format(jour)
32.                                 + "," + idClient + "," + idCreneau + "]";
33.         }
34.
35.         // getters - settters
36. ...
37. }
```

- ligne 8 : le n° du RV
- ligne 9 : le jour du RV
- ligne 10 : le n° du client
- ligne 11 : le n° du créneau
- lignes 16-23 : le constructeur permettant d'initialiser une instance [RawRv] avec les quatre informations nécessaires
- lignes 24-26 : le constructeur permettant d'initialiser une instance [RawRv] avec les quatre informations d'une autre instance de type [RawRv].
- lignes 29-33 : la méthode [toString] de la classe. L'instruction :

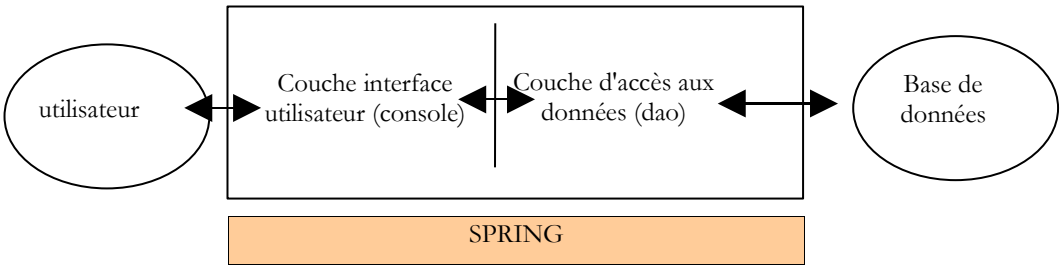
```
new SimpleDateFormat("dd/MM/yyyy").format(jour)
```

fait la conversion d'un type [java.util.Date] vers un type String au format " dd/MM/yyyy " où dd (day) désigne les deux chiffres du jour dans le mois, MM (month) les deux chiffres du mois dans l'année, yyyy (year) les quatre chiffres de l'année. La classe [SimpleDateFormat] nécessite l'import de la ligne 3.

- lignes 35 et au-delà : les méthodes get / set associées à chacun des champs privés de la classe.

2.3 L'interface de la couche [dao]

Revenons à l'architecture de l'application :



L'interface Java de la couche [dao] sera la suivante :

```
1.  package istia.st.rdvmedecins.dao.service;
2.
3.  import istia.st.rdvmedecins.dao.entites.RawClient;
4.  import istia.st.rdvmedecins.dao.entites.RawCreneau;
5.  import istia.st.rdvmedecins.dao.entites.RawMedecin;
6.  import istia.st.rdvmedecins.dao.entites.RawRv;
7.
8.  import java.util.List;
9.
10. public interface IDao {
11.
12.         // liste des clients
13.         public List<RawClient> getAllClients();
14.         // liste des Médecins
15.         public List<RawMedecin> getAllMedecins();
16.         // liste des créneaux horaires d'un médecin
17.         public List<RawCreneau> getAllCreneaux(int idMedecin);
18.         // liste des Rv d'un médecin, un jour donné
19.         public List<RawRv> getRvMedecinJour(int idMedecin, String jour);
20.         // ajouter un RV
21.         public int ajouterRv(String jour, int idCreneau, int idClient);
22.         // supprimer un RV
23.         public void supprimerRv(int idRv);
24.
25. }
```

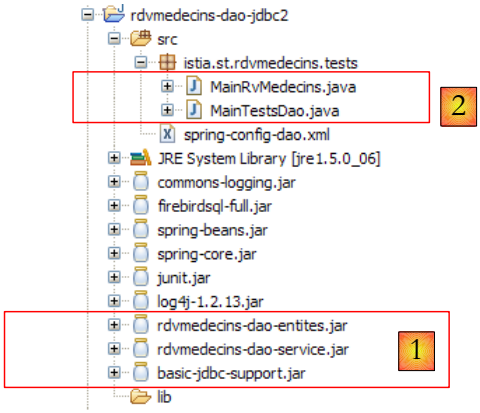
- ligne 13 : la méthode [getAllClients] met les lignes de la table [CLIENTS] dans une liste d'objets de type [RawClient] et rend cette liste.
- ligne 15 : la méthode [getAllMedecins] met les lignes de la table [MEDECINS] dans une liste d'objets de type [RawMedecin] et rend cette liste.
- ligne 17 : la méthode [getAllCreneaux] met les lignes de la table [CRENEAUX] pour lesquelles la colonne ID_MEDECIN a la valeur du paramètre **idMedecin** dans une liste d'objets de type [RawCreneau] et rend cette liste.
- ligne 19 : la méthode [getRvMedecinJour] met dans une liste d'objets de type [RawRv] les lignes de la table [RV] pour lesquelles :
    - la colonne ID_CRENEAU correspond au médecin dont le n° a pour valeur le paramètre **idMedecin**
    - la colonne JOUR (de type Date) correspond au paramètre **jour** (de type String).

  et rend cette liste.
- ligne 21 : la méthode [ajouterRv] ajoute dans la table [RV] un RV pour :
    - le client **idClient**,
    - le créneau horaire **idCreneau** (donc pour un médecin précis) du jour **jour**.

  et rend comme résultat **la clé primaire** (champ ID) de la ligne insérée dans la table RV.
- ligne 23 : la méthode [supprimerRv] supprime la ligne de la table [RV] identifiée par le n° **idRv**.

2.4 Exemple d'utilisation de la couche [dao]

Un projet Eclipse de test basique de la couche [dao] pourrait être le suivant :



- l'implémentation de la couche [dao] a été encapsulée dans les archives [1].

Le programme de test **MainTestsDao** [2] est le suivant :

```
1.  package istia.st.rdvmedecins.tests;
2.
3.  import java.util.List;
4.
5.  import istia.st.rdvmedecins.dao.service.IDao;
6.
7.  import org.springframework.beans.factory.xml.XmlBeanFactory;
8.  import org.springframework.core.io.ClassPathResource;
9.
10. public class MainTestsDao {
11.
12.         public static void main(String[] args) {
13.                 // instanciation couche [dao]
14.                 IDao dao = (IDao) (new XmlBeanFactory(new ClassPathResource(
15.                                 "spring-config-dao.xml"))).getBean("dao");
16.                 // affichage médecins
17.                 try {
18.                         display("Liste des médecins :", dao.getAllMedecins());
19.                 } catch (JdbcException ex) {
20.                         System.out.println(ex);
21.                 }
22.                 // affichage créneaux d'un médecin
23.                 try {
24.                         display("Liste des créneaux du médecin n° 1 :", dao
25.                                         .getAllCreneaux(1));
26.                 } catch ( JdbcException ex) {
27.                         System.out.println(ex);
28.                 }
29.                 // liste des Rv d'un médecin, un jour donné
30.                 try {
31.                         display("Liste des Rv du médecin n° 1, le 23/08/2006 :", dao
32.                                         .getRvMedecinJour(1, "2006:08:23"));
33.                 } catch ( JdbcException ex) {
34.                         System.out.println(ex);
35.                 }
36.                 // ajouter un RV
37.                 System.out.println("Ajout d'un Rv au médecin n° 1");
38.                 int idRv=0;
39.                 try {
40.                         idRv=dao.ajouterRv("2006:08:23", 2, 2);
41.                         display("Liste des Rv du médecin n° 1, le 23/08/2006 :", dao
42.                                         .getRvMedecinJour(1, "2006:08:23"));
43.                 } catch ( JdbcException ex) {
44.                         System.out.println(ex);
45.                 }
46.                 // ajouter un RV dans le même créneau du même jour
47.                 System.out.println("Ajouter un RV dans le même créneau du même jour");
48.                 try {
49.                         idRv=dao.ajouterRv("2006:08:23", 2, 4);
50.                         display("Liste des Rv du médecin n° 1, le 23/08/2006 :", dao
51.                                         .getRvMedecinJour(1, "2006:08:23"));
52.                 } catch ( JdbcException ex) {
53.                         System.out.println(ex);
54.                 }
55.                 // supprimer un RV
56.                 System.out.println("Suppression du Rv ajouté");
57.                 try {
58.                         dao.supprimerRv(idRv);
59.                         display("Liste des Rv du médecin n° 1, le 23/08/2006 :", dao
```

```
60.                                     .getRvMedecinJour(1, "2006:08:23"));
61.                 } catch ( JdbcException ex) {
62.                         System.out.println(ex);
63.                 }
64.         }
65.
66.         // méthode utilitaire - affiche les éléments d'un objet List
67.         private static void display(String message, List elements) {
68.                 System.out.println(message);
69.                 for (Object element : elements) {
70.                         System.out.println(element);
71.                 }
72.         }
73.
74. }
```

On notera les points suivants :

- lignes 14-15 : instanciation d'une classe implémentant l'interface [IDao]. Le fichier de configuration [spring-config-dao.xml] fournit à cette classe les informations sur le SGBD (pilote JDBC, url de la base, utilisateur de la connexion et son mot de passe). C'est pourquoi ces informations n'apparaissent nulle part dans le programme de test. Il n'a pas à les connaître.
- les exceptions lancées par la couche [dao] sont de type [JdbcException], un type dérivé de [RuntimeException]. On sait que les exceptions dérivées de [RuntimeException] sont dites **non contrôlées** parce que le compilateur ne nous oblige pas à les gérer avec un try / catch. Ici chaque appel de méthode à la couche [dao] a été entouré d'un try / catch, mais ce n'était pas obligatoire.

A l'exécution, l'affichage écran obtenu est le suivant :

```
1.  Liste des médecins :
2.  [1,Mme,Marie,PELISSIER]
3.  [2,Mr,Jacques,BROMARD]
4.  [3,Mr,Philippe,JANDOT]
5.  [4,Melle,Justine,JACQUEMOT]
6.  Liste des créneaux du médecin n° 1 :
7.  [1,1,8:0,8:20]
8.  [2,1,8:20,8:40]
9.  [3,1,8:40,9:0]
10. [4,1,9:0,9:20]
11. [5,1,9:20,9:40]
12. [6,1,9:40,10:0]
13. [7,1,10:0,10:20]
14. [8,1,10:20,10:40]
15. [9,1,10:40,11:0]
16. [10,1,11:0,11:20]
17. [11,1,11:20,11:40]
18. [12,1,11:40,12:0]
19. [13,1,14:0,14:20]
20. [14,1,14:20,14:40]
21. [15,1,14:40,15:0]
22. [16,1,15:0,15:20]
23. [17,1,15:20,15:40]
24. [18,1,15:40,16:0]
25. [19,1,16:0,16:20]
26. [20,1,16:20,16:40]
27. [21,1,16:40,17:0]
28. [22,1,17:0,17:20]
29. [23,1,17:20,17:40]
30. [24,1,17:40,18:0]
31. Liste des Rv du médecin n° 1, le 23/08/2006 :
32. [4,23/08/2006,3,4]
33. [8,23/08/2006,1,6]
34. [9,23/08/2006,3,7]
35. [6,23/08/2006,1,12]
36. [5,23/08/2006,4,20]
37. Ajout d'un Rv au médecin n° 1
38. Liste des Rv du médecin n° 1, le 23/08/2006 :
39. [74,23/08/2006,2,2]
40. [4,23/08/2006,3,4]
41. [8,23/08/2006,1,6]
42. [9,23/08/2006,3,7]
43. [6,23/08/2006,1,12]
44. [5,23/08/2006,4,20]
45. Ajouter un RV dans le même créneau du même jour
46. istia.st.utils.jdbc.JdbcException: Erreur d'accès aux données :
    org.firebirdsql.jdbc.FBSQLException: GDS Exception. 335544665. violation of PRIMARY or UNIQUE KEY
    constraint "UNQ1_RV" on table "RV"
47. Suppression du Rv ajouté
48. Liste des Rv du médecin n° 1, le 23/08/2006 :
49. [4,23/08/2006,3,4]
50. [8,23/08/2006,1,6]
51. [9,23/08/2006,3,7]
52. [6,23/08/2006,1,12]
53. [5,23/08/2006,4,20]
```

Nous laissons au lecteur le soin de faire le lien entre le code exécuté et les résultats obtenus.

2.5 L'application console de gestion des Rv

Nous souhaitons écrire une application console basique de gestion des Rv. C'est l'application [MainRvMedecins] présentée en [2] dans le projet Eclipse précédent. Nous présentons ci-dessous des copies d'écran de son fonctionnement.

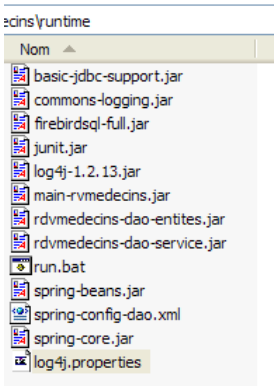
Le programme console est interactif : il accepte d'exécuter des commandes tapées au clavier. Celles-ci sont présentées au démarrage et après chaque exécution d'une commande :

```
Commandes (* pour arrêter) :
1 : (getMedecins)
2 : (getCreneauxMedecin) idMedecin
3 : (getClients)
4 : (getRvMedecinJour) idMedecin aaaa:mm:jj
5 : (ajouterRvJourCreneauClient) aaaa:mm:jj idCreneau idClient
6 : (supprimerRv) idRv
```

Pour taper une commande, l'utilisateur tape son n° et éventuellement les informations complémentaires dont elle a besoin. L'argument entre parenthèses n'est pas à donner.

Dans la démonstration qui suit, nous supposons :

- que tous les exécutables nécessaires à l'application ont été placés dans des archives
- que celles-ci ont été rassemblées dans un dossier avec les fichiers de configuration nécessaires



La classe [istia.st.rdvmedecins.tests.MainRvMedecins] testée a été placée ci-dessus dans l'archive [main-rvmedecins.jar]. L'application est lancée dans une fenêtre Dos, avec le fichier [run.bat] suivant :

```
1.  @ECHO OFF
2.  SET JAVA_HOME="C:\Program Files\Java\jre1.5.0_10"
3.  %JAVA_HOME%\bin\java.exe -classpath .;rdvmedecins-dao-entites.jar;rdvmedecins-dao-
    service.jar;basic-jdbc-support.jar;spring-core.jar;spring-beans.jar;log4j-
    1.2.13.jar;junit.jar;firebirdsql-full.jar;commons-logging.jar;main-rvmedecins.jar
    istia.st.rdvmedecins.tests.MainRvMedecins
```

- ligne 2 : fixe l'emplacement de la machine virtuelle Java (JVM)
- ligne 3 : lance cette JVM pour exécuter la classe [istia.st.rdvmedecins.tests.MainRvMedecins].

On lance le programme :

```
dos>run

Commandes (* pour arrêter) :
1 : (getMedecins)
2 : (getCreneauxMedecin) idMedecin
3 : (getClients)
4 : (getRvMedecinJour) idMedecin aaaa:mm:jj
5 : (ajouterRvJourCreneauClient) aaaa:mm:jj idCreneau idClient
6 : (supprimerRv) idRv
```

Dans les copies d'écran ci-dessous, **D** désigne la **demande** de l'utilisateur, **R** la **réponse** que lui fait l'application. Celle-ci comprend toujours la liste des commandes ci-dessus. Par facilité, celle-ci n'est pas reproduite dans les copies d'écran ci-dessous.

On demande la liste des médecins :

D
```
1
```
R
```
Liste des médecins :
[1,Mme,Marie,PELISSIER]
[2,Mr,Jacques,BROMARD]
[3,Mr,Philippe,JANDOT]
[4,Melle,Justine,JACQUEMOT]
```

On demande la liste des clients :

D
```
3
```
R
```
Liste des clients :
[1,Mr,Jules,MARTIN]
[2,Mme,Christine,GERMAN]
[3,Mr,Jules,JACQUARD]
[4,Melle,Brigitte,BISTROU]
```

On demande la liste des créneaux du médecin n° 1 (Mme PELISSIER) :

D
```
2 1
```
R
```
Liste des créneaux du médecin n° 1 :
[1,1,8:0,8:20]
[2,1,8:20,8:40]
[3,1,8:40,9:0]
[4,1,9:0,9:20]
[5,1,9:20,9:40]
[6,1,9:40,10:0]
[7,1,10:0,10:20]
[8,1,10:20,10:40]
[9,1,10:40,11:0]
[10,1,11:0,11:20]
[11,1,11:20,11:40]
[12,1,11:40,12:0]
[13,1,14:0,14:20]
[14,1,14:20,14:40]
[15,1,14:40,15:0]
[16,1,15:0,15:20]
[17,1,15:20,15:40]
[18,1,15:40,16:0]
[19,1,16:0,16:20]
[20,1,16:20,16:40]
[21,1,16:40,17:0]
[22,1,17:0,17:20]
[23,1,17:20,17:40]
[24,1,17:40,18:0]
```

Mme PELISSIER a 24 créneaux.

Mr BROMARD (client n° 2) téléphone pour avoir un RV avec Mme PELISSIER le 23/08/2006. On regarde les RV de Mme PELISSIER pour ce jour :

D
```
4 1 2006:08:23
```
R
```
Liste des Rv du médecin n° [1], le 2006:08:23
[4,23/08/2006,3,4]
[8,23/08/2006,1,6]
[9,23/08/2006,3,7]
[6,23/08/2006,1,12]
[5,23/08/2006,4,20]
```

Après discussion, Mr BROMARD prend le créneau n° 10 (11 h – 11 h 20). On le lui réserve :

D
```
5 2006:08:23 10 2
```
R
```
Rv ajouté
```

On vérifie :

D
```
4 1 2006:08:23
```
R
```
Liste des Rv du médecin n° [1], le 2006:08:23
[4,23/08/2006,3,4]
[8,23/08/2006,1,6]
[9,23/08/2006,3,7]
[77,23/08/2006,2,10]
[6,23/08/2006,1,12]
[5,23/08/2006,4,20]
```

La réservation a bien été faite (n° 77 ci-dessus).

Mr MARTIN (client n° 1) qui, parce qu'il hésitait, avait fait 2 réservations pour le 23/08/2006, téléphone pour annuler celle de 11 h 40 (créneau n° 12). On supprime le RV n° 6 :

D
```
6 6
```
R
```
Rv n° 6 supprimé.
```

On vérifie :

D
```
4 1 2006:08:23
```
R
```
Liste des Rv du médecin n° [1], le 2006:08:23
[4,23/08/2006,3,4]
[8,23/08/2006,1,6]
[9,23/08/2006,3,7]
[77,23/08/2006,2,10]
[5,23/08/2006,4,20]
```

On arrête l'application :

D
```
*
```

Le programme teste la validité des commandes tapées au clavier, comme le montre la nouvelle exécution suivante :

```
dos>run

Commandes (* pour arrêter) :
1 : (getMedecins)
...
6 : (supprimerRv) idRv
```

On tape une commande vide :

D
```

```
R
```
Tapez une commande...

Commandes (* pour arrêter) :
1 : (getMedecins)
...
6 : (supprimerRv) idRv
```

On tape une commande inexistante :

D
```
7
```
R
```
Commande non reconnue ...

Commandes (* pour arrêter) :
1 : (getMedecins)
...
6 : (supprimerRv) idRv
```

On tape la commande 1 avec des arguments incorrects :
D
```
1 xx
```
R

```
Syntaxe incorrecte

Commandes (* pour arrêter) :
1 : (getMedecins)
...
6 : (supprimerRv) idRv
```

etc...

Voici les cas à considérer selon la valeur de la commande :

1. la commande 1 ne doit pas avoir de paramètres
2. la commande 2 doit avoir un paramètre de type entier
3. la commande 3 ne doit pas avoir de paramètres
4. la commande 4 doit avoir un paramètre de type entier et un paramètre de type chaîne.
5. la commande 5 doit avoir un paramètre de type chaîne et deux paramètres de type entier.
6. la commande 6 doit avoir un paramètre de type entier

Note : il n'y a pas lieu de vérifier que les dates saisies sont valides ni que les nombres entiers saisis sont dans un intervalle donné. En effet, dans ces deux cas d'erreur, le SGBD réagit correctement :

- dans le cas d'un SELECT, il ne rend aucune ligne
- dans le cas d'un INSERT dans la table RV, il ne fait pas l'insertion et lance une exception à cause des contraintes de clés étrangères que possède la table RV
- dans le cas d'un DELETE sur la table RV, il ne fait pas la suppression parce qu'aucune ligne n'a la clé primaire recherchée

---

Question 1 : Ecrire l'application console [MainRvMedecins].

- la syntaxe des différentes commandes tapées par l'utilisateur sera vérifiée et les erreurs éventuelles signalées.
- les exceptions seront interceptées et donneront lieu à des messages d'erreur à l'écran
- on s'inspirera de l'exemple présenté au paragraphe 2.4, page 8. Notamment, l'instanciation de la couche [dao] sera faite de la même façon.

Conseils : Pour récupérer les éléments de la commande tapée au clavier par l'utilisateur, on pourra utiliser la méthode **[String].split("expression régulière")**. Voici un exemple d'utilisation de cette méthode :

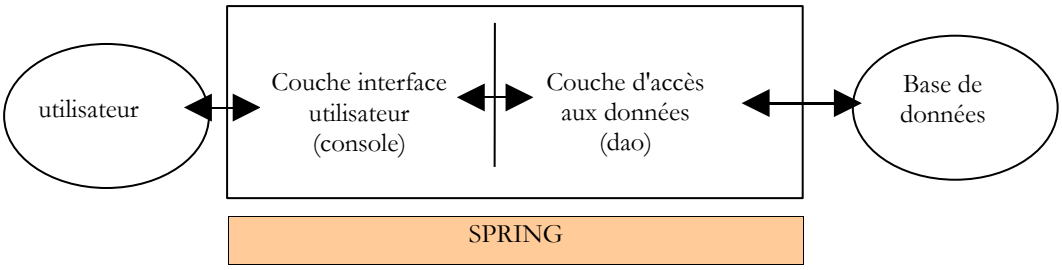
```
String ligne="champ0 champ1 champ2";
String[] champs=ligne.split("\\s+");
```

La chaîne "**\\s+**" indique que les champs de la chaîne **ligne** sont séparés par un ou plusieurs " espaces ". Après les instructions précédentes, **champs** sera un tableau de 3 objets de type String avec :

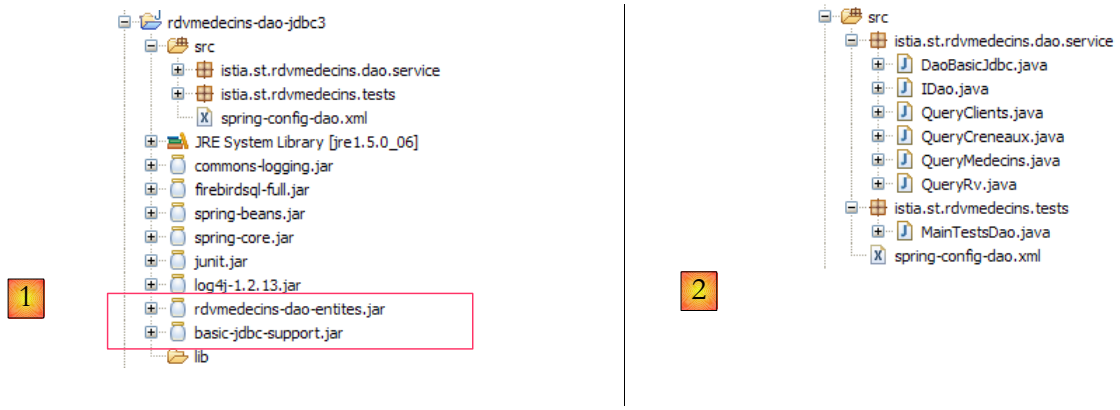
```
champ[0]="champ0"
champ[1]="champ1"
champ[2]="champ2"
```

3 Implémentation de la couche [dao]

Nous nous proposons d'étudier maintenant l'implémentation de l'interface [IDao] de la couche [dao] :



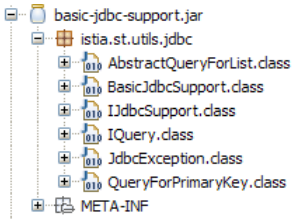
L'interface Java [IDao] de la couche [dao] a été présentée au paragraphe 2.3, page 7. Le projet Eclipse de son implémentation pourrait être le suivant :



- les objets de l'application, décrits au paragraphe 2.2, page 3, sont encapsulés dans l'archive [rdvmedecins-dao-entites.jar] (cf [1])
- l'archive [basic-jdbc-support] [1], contient des classes utilitaires facilitant la gestion des bases de données.
- [2] montre le code source de l'implémentation particulière de l'interface [IDao] que nous allons écrire.

3.1 Classes utilitaires

Pour implémenter la couche [dao], nous allons nous appuyer sur un certain nombre de classes utilitaires encapsulées dans l'archive [basic-jdbc-support]. Ces classes sont les suivantes :



[**IJdbcSupport**] est une interface :

```
1.  package istia.st.utils.jdbc;
2.
3.  import java.util.List;
4.
5.  public interface IJdbcSupport {
6.
7.          // liste de lignes d'une table
8.          public abstract List queryForList(IQuery query);
9.
10.         // obtention d'une valeur de clé primaire à partir d'un générateur de valeurs entières
11.         public abstract int queryForPrimaryKey(String selectText);
12.
```

```
13.         // mise à jour (UPDATE, DELETE, INSERT) d'une table
14.         public abstract int update(String updateText);
15. }
```

- ligne 8 : la méthode **queryForList** permet d'exécuter une commande SQL Select et d'en obtenir le résultat dans une liste d'objets. Le paramètre [IQuery query] permet d'encapsuler les lignes, résultat du SELECT, dans le type d'objet approprié. Ce paramètre encapsule un certain nombre d'informations :
  - la requête SQL SELECT à exécuter
  - la connexion sur laquelle l'exécuter
  - le type d'objet à utiliser pour encapsuler chaque ligne résultat du SELECT
- ligne 14 : la méthode **update** permet d'exécuter un ordre SQL de mise à jour (UPDATE, INSERT, DELETE). Elle rend le nombre de lignes mises à jour.
- ligne 11 : chaque insertion d'une nouvelle ligne dans une table ayant une clé primaire, nécessite qu'on donne à cette ligne une valeur de clé primaire qui n'existe pas dans les lignes déjà existantes dans la table. Lorsque la clé primaire de la table est de type entier, et que la base offre un générateur de valeurs entières, la méthode **queryForPrimaryKey** permet d'obtenir auprès de ce générateur une valeur de clé primaire pour la ligne à insérer. Le paramètre **selectText** est l'ordre SQL SELECT permettant d'interroger le générateur de valeurs entières.
- les méthodes d'accès aux données d'une base de données relationnelle lance l'exception de type **SQLException**. On voit ici, qu'aucune des trois méthodes de l'interface [IJdbcSupport] ne lance d'exception. Cela signifie que les classes implémentant l'interface [IJdbcSupport] devront lancer une exception non contrôlée, c.a.d. dérivée de la classe [RuntimeException].

[**BasicJdbcSupport**] est une classe implémentant l'interface [**IJdbcSupport**] qui lance des exceptions de type [**JdbcException**]. Cette classe d'exceptions est dérivée de [**RuntimeException**].

[**IQuery**] est une interface :

```
1.  package istia.st.utils.jdbc;
2.
3.  import java.sql.Connection;
4.  import java.sql.SQLException;
5.  import java.util.List;
6.
7.  public interface IQuery {
8.          // connexion
9.          void setConnection(Connection connection);
10.
11.         // ordre SQL select
12.         void setSelectText(String selectText);
13.
14.         // exécution de la requête Select
15.         List execute() throws SQLException;
16. }
```

- ligne 9 : la méthode **setConnection** permet de passer à l'interface [IQuery] un objet de type **Connection** qui représente une connexion Jdbc **ouverte**.
- ligne12 : la méthode **setSelectText** permet de passer à l'interface [IQuery] le texte de l'ordre SQL SELECT à exécuter sur la connexion ouverte de l'interface.
- ligne 15 : la méthode **execute** permet d'exécuter l'ordre SQL **selectText** sur la connexion ouverte de l'interface [IQuery]. Elle rend une **liste d'objets**, chaque objet encapsulant l'une des lignes rendues par l'ordre SQL Select exécuté. Elle est susceptible de lancer une exception de type **SQLException**.

La classe **AbstractQueryForList** est une classe implémentant l'interface [IQuery]. Son code est le suivant :

```
1.  package istia.st.utils.jdbc;
2.
3.  import java.sql.Connection;
4.  import java.sql.ResultSet;
5.  import java.sql.SQLException;
6.  import java.util.List;
7.
8.  public abstract class AbstractQueryForList implements IQuery {
9.
10.         // champs privés
11.         private Connection connection;
12.         private String selectText;
13.
14.         // constructeurs
15.         public AbstractQueryForList() {
16.
17.         }
18.
19.         public AbstractQueryForList(Connection connection, String selectText) {
20.                 setConnection(connection);
```

```
21.                 setSelectText(selectText);
22.         }
23.
24.         // méthode execute
25.         public List execute() throws SQLException {
26.                 return getRecords(connection.createStatement().executeQuery(selectText));
27.         }
28.
29.
30.         // méthode implémentée par les classes dérivées
31.         protected abstract List getRecords(ResultSet resultSet) throws SQLException;
32.
33.         // getters - setters
34. ...
35.         public void setConnection(Connection connection) {
36.                 this.connection = connection;
37.         }
38.
39. }
```

- ligne 8 : la classe [AbstractQueryForList] implémente l'interface **IQuery**. Elle est déclarée abstraite (mot clé abstract).
- la classe [AbstractQueryForList] exécute un ordre SQL SELECT sur une connexion ouverte. La connexion est mémorisée ligne 11 et le texte de l'ordre SQL SELECT, ligne 12.
- lignes 14-22 : les constructeurs de la classe [AbstractQueryForList]
- lignes 34-37 : les getters / setters liés aux champs privés. On y trouve la méthode [setConnection] lignes 35-37, qui était l'une des deux méthodes de l'interface [IQuery] à implémenter.
- lignes 25-27 : implémentation de la méthode [execute], la seconde méthode de l'interface [IQuery]. Cette méthode doit exécuter l'ordre SQL **selectText**. La ligne 26 l'exécute et récupère un objet *ResultSet* qu'elle demande à la méthode interne [getRecords] d'exploiter.
- ligne 31 : la méthode [getRecords] reçoit un paramètre de type *ResultSet* à partir duquel elle doit rendre une liste d'objets, encapsulant les lignes du *ResultSet*. Comme la méthode [getRecords] n'a pas connaissance du texte de la requête SELECT qui a donné naissance au *ResultSet* qu'on lui a passé, ni du type d'objet qui doit être créé à partir de chaque ligne du *ResultSet*, elle ne peut être implémentée. Elle le sera par les classes dérivées de [AbstractQueryForList]. Pour cette raison, elle est déclarée abstraite (mot clé abstract) faisant de la classe elle-même une classe abstraite. Une classe abstraite ne peut être instanciée et doit être obligatoirement dérivée pour l'être.

En conclusion, les classes implémentant l'interface [IQuery] pourront être construites en :
- héritant de la classe [AbstractQueryForList]
- implémentant la méthode [getRecords]

Nous en savons assez pour écrire un programme utilisant l'interface [**IJdbcSupport**]. Ce programme de test réalisera les quatre opérations élémentaires sur la table [MEDECINS] :
- liste des lignes
- insertion d'une nouvelle ligne
- modification d'une ligne existante
- suppression d'une ligne

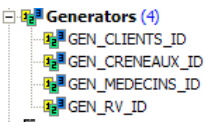
La table [MEDECINS] contient des informations sur les médecins gérés par l'application [RdvMedecins].

PK	Field Name	Field Type	Domain	Size
1	ID	INTEGER		
	NOM	VARCHAR		30
	PRENOM	VARCHAR		30
	TITRE	VARCHAR		5

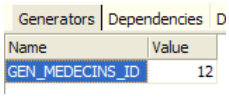
ID	NOM	PRENOM	TITRE
1	PELISSIER	Marie	Mme
2	BROMARD	Jacques	Mr
3	JANDOT	Philippe	Mr
4	JACQUEMOT	Justine	Melle

- ID : n° identifiant le médecin - clé primaire de la table
- NOM : le nom du médecin
- PRENOM : son prénom
- TITRE : son titre (Melle, Mme, Mr)

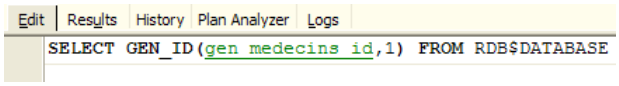
Les valeurs de la clé primaire ID peuvent être obtenues auprès d'un générateur de valeurs, comme le montre l'exemple qui suit :



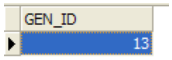
La base a 4 générateurs de valeurs entières, 1 par table.



Celui de la table [MEDECINS] s'appelle GEN_MEDECINS_ID et a pour l'instant la valeur 12.

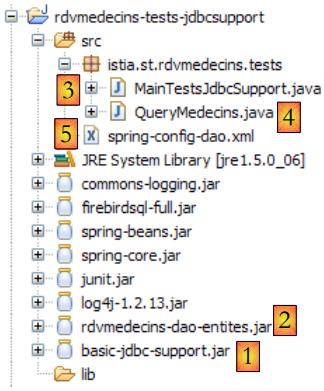


On demande la valeur suivante du générateur



On obtient 13.

Le projet Eclipse de test de l'interface **IJdbcSupport** est le suivant :



- [1] : le jar qui encapsule les classes utilitaires décrites au début de ce paragraphe, page 14.
- [2] : le jar qui encapsule les objets de l'application [RdvMedecins], décrits page 3.
- [3] : le programme de test
- [4] : l'objet [IQuery] permettant d'interroger la table [MEDECINS].
- [5] : le fichier de configuration de l'application

La classe [QueryMedecins] est la suivante :

```
1.  package istia.st.rdvmedecins.tests;
2.
3.  import istia.st.rdvmedecins.dao.entites.RawMedecin;
4.  import istia.st.utils.jdbc.AbstractQueryForList;
5.
6.  import java.sql.Connection;
7.  import java.sql.ResultSet;
8.  import java.sql.SQLException;
9.  import java.util.ArrayList;
10. import java.util.List;
11.
12. public class QueryMedecins extends AbstractQueryForList {
13.
14.  // constructeurs
15.  public QueryMedecins() {
16.
17.  }
18.
19.  public QueryMedecins(Connection connection, String selectText) {
20.     super(connection, selectText);
21.  }
22.
23.  // getRecords
24.  protected List getRecords(ResultSet resultSet) throws SQLException {
25.     // exploitation résultat
26.     List<RawMedecin> listMedecins = new ArrayList<RawMedecin>();
27.     while (resultSet.next()) {
28.        listMedecins.add(new RawMedecin(resultSet.getInt(1), resultSet
29.              .getString(2), resultSet.getString(3), resultSet
30.              .getString(4)));
31.     }
32.     return listMedecins;
```

```
33.   }
34.
35. }
```

- ligne 12 : pour implémenter l'interface [IQuery], la classe [QueryMedecins] hérite de la classe [AbstractQueryForList]
- ligne 24 : implémentation de la méthode [getRecords] qui était déclarée abstraite dans la classe [AbstractQueryForList]
- lignes 26-32 : la classe [QueryMedecins] est destinée à exploiter le résultat d'ordres SQL Select suivants " **SELECT ID, TITRE, NOM, PRENOM FROM MEDECINS ....** ". Le signe ... peut représenter des clauses WHERE, ORDER BY. On sait donc exactement quelles sont les colonnes demandées. On peut donc encapsuler les lignes, résultat du SELECT, dans un objet approprié, ici le type [RawMedecin]. Ces informations n'étaients pas connues de la méthode [AbstractQueryForList].getRecords, ce qui nous avait obligés à déclarer **abstraite** cette méthode.

Le programme de test [MainTestsJdbcSupport] est le suivant :

```
1.  package istia.st.rdvmedecins.tests;
2.
3.  import java.util.List;
4.
5.  import istia.st.rdvmedecins.dao.entites.RawMedecin;
6.  import istia.st.utils.jdbc.IJdbcSupport;
7.  import istia.st.utils.jdbc.IQuery;
8.  import istia.st.utils.jdbc.JdbcException;
9.
10. import org.springframework.beans.factory.xml.XmlBeanFactory;
11. import org.springframework.core.io.ClassPathResource;
12.
13. public class MainTestsJdbcSupport {
14.
15.  // requête SELECT sur la table MEDECINS
16.  static private IQuery queryMedecins = null;
17.
18.  // support JDBC
19.  static private IJdbcSupport jdbcSupport;
20.
21.  public static void main(String[] args) {
22.     // instanciation interface [IJdbcSupport]
23.     jdbcSupport = (IJdbcSupport) (new XmlBeanFactory(new ClassPathResource(
24.           "spring-config-dao.xml"))).getBean("jdbcSupport");
25.     // l'interface IQuery de requête
26.     queryMedecins = new QueryMedecins(null,
27.           "SELECT ID, TITRE, NOM, PRENOM FROM MEDECINS");
28.     // affichage médecins
29.     displayMedecins();
30.     // insertion d'une ligne
31.     int idMedecin = jdbcSupport
32.           .queryForPrimaryKey("SELECT GEN_ID(GEN_MEDECINS_ID,1) FROM RDB$DATABASE");
33.     jdbcSupport.update("INSERT INTO MEDECINS(ID,TITRE,NOM,PRENOM) VALUES("
34.           + idMedecin + ",'Mr','Delfou','Jean-Claude')");
35.     // affichage médecins
36.     displayMedecins();
37.     // modification d'une ligne
38.     jdbcSupport.update("UPDATE MEDECINS SET NOM='DELFOU' WHERE ID="
39.           + idMedecin);
40.     // affichage médecins
41.     displayMedecins();
42.     // insertion d'une ligne en doublon (même clé primaire)
43.     try {
44.        jdbcSupport
45.              .update("INSERT INTO MEDECINS(ID,TITRE,NOM,PRENOM) VALUES("
46.                    + idMedecin + ",'Mr','Delfou','Jean-Claude')");
47.     } catch (JdbcException ex) {
48.        System.out.println(ex);
49.     }
50.     // suppression ligne
51.     jdbcSupport.update("DELETE FROM MEDECINS WHERE ID=" + idMedecin);
52.     // affichage médecins
53.     displayMedecins();
54.  }
55.
56.  // liste des médecins
57.  @SuppressWarnings("unchecked")
58.  private static void displayMedecins() {
59.     System.out.println("Liste des médecins :");
60.     for (RawMedecin element : (List<RawMedecin>) jdbcSupport
61.           .queryForList(queryMedecins)) {
62.        System.out.println(element);
63.     }
64.  }
65.
66. }
```

On notera les points suivants :

- lignes 23-24 : instanciation d'une classe implémentant l'interface [IJdbcSupport]. Le fichier de configuration fournit à cette classe les informations sur le SGBD (pilote JDBC, url de la base, utilisateur de la connexion et son mot de passe). C'est pourquoi ces informations n'apparaissent nulle part dans le programme de test. Il n'a pas à les connaître.
- lignes 26-27 : instanciation d'un objet implémentant l'interface [IQuery]. On lui fournit l'ordre SQL Select à exécuter sur la table [MEDECINS]. On remarquera qu'on ne lui fournit pas la connexion nécessaire à l'exécution de l'ordre SQL. Ce sera l'objet [IJdbcSupport] qui le lui fournira.
- lors des appels de méthodes de l'interface [IJdbcSupport], on ne gère pas les éventuelles exceptions. On n'y est pas obligés, car ces méthodes lancent des exceptions de type [JdbcException] dérivé de [RuntimeException]. Lignes 43-49, on gère l'exception car on sait qu'elle va se produire et on veut la gérer.

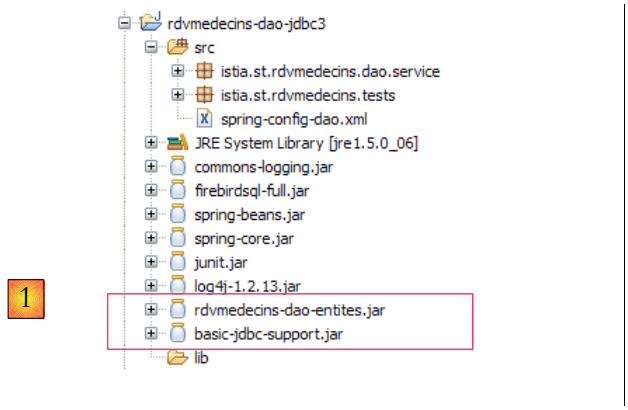
L'affichage obtenu dans la console lors de l'exécution de ce code est le suivant :

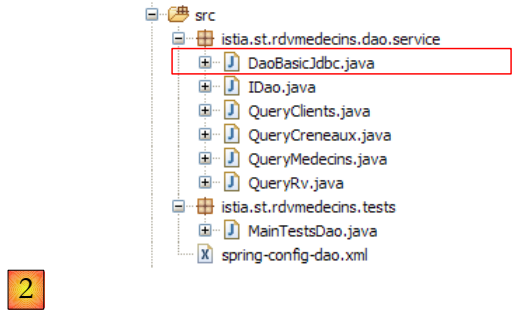
```
1.  Liste des médecins :
2.  [1,Mme,Marie,PELISSIER]
3.  [2,Mr,Jacques,BROMARD]
4.  [3,Mr,Philippe,JANDOT]
5.  [4,Melle,Justine,JACQUEMOT]
6.
7.  Liste des médecins :
8.  [1,Mme,Marie,PELISSIER]
9.  [2,Mr,Jacques,BROMARD]
10. [3,Mr,Philippe,JANDOT]
11. [4,Melle,Justine,JACQUEMOT]
12. [14,Mr,Jean-Claude,Delfou]
13.
14. Liste des médecins :
15. [1,Mme,Marie,PELISSIER]
16. [2,Mr,Jacques,BROMARD]
17. [3,Mr,Philippe,JANDOT]
18. [4,Melle,Justine,JACQUEMOT]
19. [14,Mr,Jean-Claude,DELFOU]
20.
21. istia.st.utils.jdbc.JdbcException: Erreur d'accès aux données :
    org.firebirdsql.jdbc.FBSQLException: GDS Exception. 335544665. violation of PRIMARY or UNIQUE KEY
    constraint "PK_MEDECINS" on table "MEDECINS"
22.
23. Liste des médecins :
24. [1,Mme,Marie,PELISSIER]
25. [2,Mr,Jacques,BROMARD]
26. [3,Mr,Philippe,JANDOT]
27. [4,Melle,Justine,JACQUEMOT]
```

Nous laissons le lecteur vérifier que les résultats obtenus sont ceux qui étaient attendus.

3.2 Implémentation de l'interface [IDao]

Redonnons une vue d'ensemble du projet Eclipse de l'implémentation de l'interface [IDao] :





- les objets de l'application, décrits au paragraphe 2.2, page 3, sont encapsulés dans l'archive [rdvmedecins-dao-entites.jar] (cf [1])
- l'archive [basic-jdbc-support] [1], contient les classes utilitaires décrites au paragraphe précédent.
- [2] montre les classes et interfaces de l'implémentation particulière de l'interface [IDao] qu'il faut écrire.

Un programme utilisant la couche [dao] a été décrit au paragraphe 2.4, page 8. Celui utilisait (lignes 14-15) le fichier de configuration [spring-config-dao.xml] suivant :

```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
```

```
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <!-- l'interface utilitaire IJdbcSupport -->
5.    <bean id="jdbcSupport" class="istia.st.utils.jdbc.BasicJdbcSupport" init-method="init">
6.       <property name="driverClassName"
7.          value="org.firebirdsql.jdbc.FBDriver" />
8.       <property name="urlDataBase"
9.          value="jdbc:firebirdsql:localhost/3050:C:\data\2006-
    2007\databases\rdvMedecins\rdvmedecins.gdb" />
10.      <property name="userName" value="SYSDBA" />
11.      <property name="password" value="masterkey" />
12.   </bean>
13.   <!-- la couche [rdvmedecins-dao] -->
14.   <bean id="dao" class="istia.st.rdvmedecins.dao.service.DaoBasicJdbc">
15.      <property name="jdbcSupport" ref="jdbcSupport" />
16.   </bean>
17. </beans>
```

- lignes 14-16 : instanciation de la classe [DaoBasicJdbc]
- ligne 15 : initialisation du champ [jdbcSupport] de l'objet [DaoBasicJdbc] instancié, avec une référence à l'objet de type [BasicJdbcSupport] créé lignes 4-12

On suivra ce schéma :

- la classe [DaoBasicJdbc] à écrire aura un champ privé appelé [jdbcSupport] qui sera supposé initialisé dès la construction de l'objet de type [DaoBasicJdbc].
- les méthodes implémentant l'interface [IDao] utiliseront les méthodes du champ privé [jdbcSupport].

```
1.  package istia.st.rdvmedecins.dao.service;
2.
3.  ...
4.  public class DaoBasicJdbc implements IDao {
5.
6.        // champs privés
7.        private IJdbcSupport jdbcSupport;
8.
9.        // méthodes implémentées
10.       @SuppressWarnings("unchecked")
11.       public List<RawCreneau> getAllCreneaux(int idMedecin) {
12. ...
13.       }
14.
15.       @SuppressWarnings("unchecked")
16.       public List<RawClient> getAllClients() {
17. ...
18.       }
19.
20.       @SuppressWarnings("unchecked")
21.       public List<RawMedecin> getAllMedecins() {
22. ...
23.       }
24.
25.       @SuppressWarnings("unchecked")
26.       public List<RawRv> getRvMedecinJour(int idMedecin, String jour) {
27. ...
28.       }
29.
30.       public int ajouterRv(String jour, int idCreneau, int idClient) {
31. ...
32.       }
33.
34.       public void supprimerRv(int idRv) {
35. ...
36.       }
37.
38.       // getters - setters
39.       public IJdbcSupport getJdbcSupport() {
40.              return jdbcSupport;
41.       }
42.
43.       public void setJdbcSupport(IJdbcSupport jdbcSupport) {
44.              this.jdbcSupport = jdbcSupport;
45.       }
46. }
```

Question 2 : en vous aidant de ce qui a précédé, écrire une classe d'implémentation de l'interface [IDao], appelée [DaoBasicJdbc], qui s'appuierait sur les classes utilitaires de l'archive [basic-jdbc-support.jar] décrites précédemment. Les classes implémentant l'interface [IQuery] pour interroger les tables [MEDECINS, CLIENTS, CRENEAUX, RV] s'appelleront respectivement [QueryMedecins, QueryClients, QueryCreneaux, QueryRv]. Vous devrez définir le contenu de ces classes.

Conseils : tout ce que vous devez savoir pour cette question est contenu dans l'exemple des pages 17 (QueryMedecins) et 18 (MainTestsJdbcSupport). Comprenez et suivez cet exemple pour écrire la classe [DaoBasicJdbc].

4 Implémentation de l'interface [IJdbcSupport]

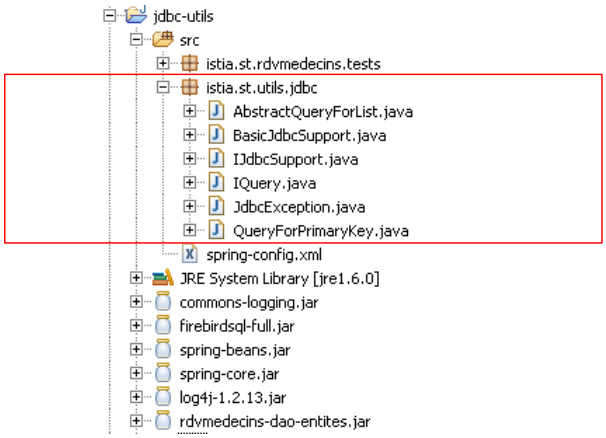
L'implémentation de l'interface [IDao] faite précédemment utilise un champ privé de type [IJdbcSupport] :

```
1.  public class DaoBasicJdbc implements IDao {
2.
3.          // champs privés
4.          private IJdbcSupport jdbcSupport;
5.
6.  ...
7.
8.          // getters - setters
9.          public IJdbcSupport getJdbcSupport() {
10.                 return jdbcSupport;
11.         }
12.
13.         public void setJdbcSupport(IJdbcSupport jdbcSupport) {
14.                 this.jdbcSupport = jdbcSupport;
15.         }
16. }
```

Nous nous proposons maintenant d'implémenter l'interface [IJdbcSupport] :

```
1.  package istia.st.utils.jdbc;
2.
3.  import java.util.List;
4.
5.  public interface IJdbcSupport {
6.
7.          // liste de lignes d'une table
8.          public abstract List queryForList(IQuery query);
9.
10.         // obtention d'une valeur de clé primaire à partir d'un générateur de valeurs entières
11.         public abstract int queryForPrimaryKey(String selectText);
12.
13.         // mise à jour (UPDATE, DELETE, INSERT) d'une table
14.         public abstract int update(String updateText);
15. }
```

Le projet Eclipse sera le suivant :



L'implémentation qui va être faite de l'interface [IJdbcSupport] nécessite diverses classes :

- [BasicJdbcSupport] est la classe implémentant l'interface [IJdbcSupport]. C'est elle que nous nous proposons d'écrire.
- [IQuery] est l'interface décrite au paragraphe 3.1, page 15.
- [AbstractQueryForList] est une classe abstraite implémentant l'interface [IQuery]. Elle a été décrite au paragraphe 3.1, page 15.
- [QueryForPrimaryKey] est une classe implémentant l'interface [IQuery] et dérivée de [AbstractQueryForList]. Elle permet d'obtenir une valeur de clé primaire tel qu'il a été montré dans l'exemple [MainTestsJdbcSupport] du paragraphe 3.1, page 18.
- [JdbcException] est une classe dérivée de [RuntimeException].

Le code de la classe [QueryForPrimaryKey] pourrait être le suivant :

```
1.  package istia.st.utils.jdbc;
2.
3.  import java.sql.Connection;
4.  import java.sql.ResultSet;
5.  import java.sql.SQLException;
6.  import java.util.ArrayList;
7.  import java.util.List;
8.
9.  public class QueryForPrimaryKey extends AbstractQueryForList{
10.
11.   // constructeurs
12.   public QueryForPrimaryKey(){
13.
14.   }
15.   public QueryForPrimaryKey(Connection connection, String selectText){
16.      super(connection,selectText);
17.   }
18.
19.   // getRecords
20.   public List getRecords(ResultSet resultSet) throws SQLException {
21.      // exploitation resultSet qui contient une seule ligne avec une unique colonne de type entier
22.      // on pointe sur la ligne
23.      resultSet.next();
24.      // on met la valeur entière de cette ligne dans un objet List
25.      List<Integer> liste = new ArrayList<Integer>();
26.      liste.add(new Integer(resultSet.getInt(1)));
27.      // on rend l'objet List
28.      return liste;
29.   }
30.
31. }
```

- ligne 9 : la classe dérive de la classe [AbstractQueryForList]. Elle doit donc implémenter la méthode [getRecords].
- lignes 20-29 : la méthode [getRecords] doit exploiter un objet *ResultSet* issu de l'exécution d'un ordre SQL SELECT demandant une nouvelle valeur pour une clé primaire de type entier. L'objet *ResultSet* n'a donc qu'une ligne et cette ligne n'a qu'une valeur : un nombre entier.
- ligne 28 : la méthode [getRecords] doit rendre un objet *List*, aussi le nombre entier récupéré dans objet *ResultSet* est-il placé dans un objet *List*. Un tel objet ne peut contenir lui-même que des objets. Or le type *int* n'est pas un objet. Aussi encapsule -t-on la valeur de clé primaire dans un objet de type *Integer* (ligne 26).

La classe [JdbcException] est la suivante :

```
1.  package istia.st.utils.jdbc;
2.
3.  import java.sql.SQLException;
4.
5.  public class JdbcException extends RuntimeException {
6.
7.          private static final long serialVersionUID = 1L;
8.
9.          // champs privés
10.         private SQLException sqlException=null;
11.         private int code=0;
12.
13.         // constructeurs
14.         public JdbcException() {
15.                 super();
16.         }
17.
18.         public JdbcException(String message) {
19.                 super(message);
20.         }
21.
22.         public JdbcException(String message, Throwable cause) {
23.                 super(message, cause);
24.         }
25.
26.         public JdbcException(Throwable cause) {
27.                 super(cause);
28.         }
29.
30.         public JdbcException(String message, int code, SQLException sqlException) {
31.                 super(message);
32.                 setCode(code);
33.                 setSqlException(sqlException);
34.         }
35.
36.         // getters - setters
37.         public int getCode() {
38.                 return code;
39.         }
40.
```

```
41.         public void setCode(int code) {
42.                 this.code = code;
43.         }
44.
45.         public SQLException getSqlException() {
46.                 return sqlException;
47.         }
48.
49.         public void setSqlException(SQLException sqlException) {
50.                 this.sqlException = sqlException;
51.         }
52.
53. }
```

- ligne 5 : la classe dérive de [RuntimeException] et est donc un type d'exception non contrôlée.
- la classe [JdbcException] va servir à encapsuler une exception contrôlée de type [SQLException] dans un type d'exception non contrôlée. L'exception de type [SQLException] ainsi encapsulée sera mémorisée dans le champ privé de la ligne 10. Un code Java interceptant une exception de type [JdbcException] aura accès à l'exception [SQLException] encapsulée grâce à la méthode [getSQLException] (lignes 45-47).
- pour différencier les exceptions de type [JdbcException] entre-elles, on pourra leur affecter un code qui sera mémorisé par le champ privé de la ligne 11. Un code Java interceptant une exception de type [JdbcException] aura accès à ce code d'erreur grâce à la méthode [getCode] (lignes 37-39).
- lignes 14-34 : les différentes façons de construire un objet de type [JdbcException].

Un exemple d'utilisation de la classe [JdbcException] pourrait être le suivant :

```
1.  try {
2.          // on charge le pilote du driver JDBC en mémoire
3.          Class.forName(driverClassName);
4.      } catch (ClassNotFoundException ex) {
5.          throw new JdbcException(
6.                "Le pilote JDBC ["
7.                        + driverClassName
8.                        + "] est introuvable. Vérifiez le fichier de configuration. Exception : "
9.                        + ex.toString(), 1, null);
10.     }
```

- l'exemple ci-dessus utilise le constructeur [JdbcException] des lignes 30-34. Le troisième paramètre de ce constructeur est une instance de type SQLException. Ci-dessus, celle-ci n'existe pas puisque l'exception est de type [ClassNotFoundException]. Aussi passe-t-on le pointeur *null* en 3ième paramètre.

La classe [BasicJdbcSupport] implémentant l'interface [IJdbcSupport] aura le squelette suivant :

```
1.  package istia.st.utils.jdbc;
2.
3.  import java.sql.Connection;
4.  import java.sql.DriverManager;
5.  import java.sql.SQLException;
6.  import java.util.List;
7.
8.  public class BasicJdbcSupport implements IJdbcSupport {
9.
10.         // champs privés
11.
12.         private String driverClassName;
13.         private String urlDataBase;
14.         private String userName;
15.         private String password;
16.
17.         // constructeur
18.         public void init() {
19. ...
20.         }
21.
22.         // liste de lignes d'une table
23.         public List queryForList(IQuery query) {
24. ...
25.         }
26.
27.         // obtention d'une valeur de clé primaire à partir d'un générateur de
28.         // valeurs entières
29.         @SuppressWarnings("unchecked")
30.         public int queryForPrimaryKey(String selectText) {
31. ...
32.         }
33.
34.         // mise à jour (UPDATE, DELETE) d'une table
35.         public int update(String updateText) {
36. ...
37.         }
```

```
38.
39.         // getters - setters
40. ...
41. }
```

- lignes 12-15 : les champs privés de la classe qui permettent de créer une connexion JDBC au SGBD. Ils pourraient être initialisés par le fichier de configuration Spring suivant :

```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <!-- la source de données -->
5.    <bean id="jdbcSupport" class="istia.st.utils.jdbc.BasicJdbcSupport" init-method="init">
6.       <property name="driverClassName"
7.          value="org.firebirdsql.jdbc.FBDriver" />
8.       <property name="urlDataBase"
9.          value="jdbc:firebirdsql:localhost/3050:D:\data\travail\...\rdvmedecins.gdb" />
10.      <property name="userName" value="SYSDBA" />
11.      <property name="password" value="masterkey" />
12.   </bean>
13. </beans>
```

Question 3 : écrire le code de la classe [BasicJdbcSupport]

Conseils :

- les accès au SGBD sont faits avec les méthodes classiques JDBC
- lorsqu'une exception de type SQLException se produit, elle doit être encapsulée dans une [JdbcException] comme il a été montré.
- au début d'un accès au SGBD, une connexion est ouverte. Elle doit être obligatoirement fermée à la sortie de la méthode qu'il y ait eu exception ou non.

Table des matières