**الفِهْرِسْت**

المقدّمة---------------------------------------------------------------------------------------------------------4

مميّزات قاعدة بيانات صفوف------------------------------------------------------------------------------------5

**الفصل الأوّل: أساسيّات قاعدة البيانات----------------------------------------------------------------------------**6

دورة حياة قاعدة البيانات ----------------------------------------------------------------------------7

أوّلًا: إنشاء قاعدة البيانات--------------------------------------------------------------------7

ثانيًا: خادم صفوف--------------------------------------------------------------------------7

نظام المستخدمين----------------------------------------------------------------------------9

ثالثًا: الجلسات-----------------------------------------------------------------------------10

رابعًا: تنفيذ الأوامر------------------------------------------------------------------------11

الأمر ربط------------------------------------------------------------------------11

الأمر تحديد----------------------------------------------------------------------12

الأمر إلغاء الرّبط-----------------------------------------------------------------12

الأمر تحديث---------------------------------------------------------------------13

الأمر التقاط----------------------------------------------------------------------14

النّص التّنفيذيّ--------------------------------------------------------------------------------------15

الشّروط-------------------------------------------------------------------------------------------16

الشّرط BooleanConditionا-----------------------------------------------------------16

الشّرط LimitConditionا---------------------------------------------------------------17

الشّرط ObjectConditionا-------------------------------------------------------------17

المرتّبات-------------------------------------------------------------------------------------------18

الانتقالات Transactionsا-----------------------------------------------------------------------18

**الفصل الثّاني: العلاقات بين الكائنات----------------------------------------------------------------------------19**

**الفصل الثّالث: أفضل استخدام لقاعدة البيانات--------------------------------------------------------------------23**

قواعد حفظ البيانات---------------------------------------------------------------------------------24

خدعٌ وتحايلٌ---------------------------------------------------------------------------------------27

حفظ حقولٍ لا تستخدم النّافذة *Serializable*ا-----------------------------------------------27

إضافة وحذف حقولٍ من الصّفّ------------------------------------------------------------28

**الفصل الرّابع: أدوات صفوف----------------------------------------------------------------------------------30**

ملفّات XMLا--------------------------------------------------------------------------------------31

مشغّل صفوف-------------------------------------------------------------------------------------32

**الفصل الخامس: تخصيص صفوف-----------------------------------------------------------------------------30**

أوامرك الخاصّة------------------------------------------------------------------------------------31

تخصيص الشّروط----------------------------------------------------------------------------------33

تخصيص المرتّبات---------------------------------------------------------------------------------33

**الخاتمة**-------------------------------------------------------------------------------------------------------34

صفوف هو مشروعٌ مفتوح المصدر, ويمكن لأيّ جهةٍ التعديل عليه, فهو مرخّص ب[رخصة وقف العامة](https://ojuba.org/waqf-2.0:%D8%B1%D8%AE%D8%B5%D8%A9_%D9%88%D9%82%D9%81_%D8%A7%D9%84%D8%B9%D8%A7%D9%85%D8%A9), والهدف منه هو تسهيل تخزين البيانات.

26/9/2018 رامي مناف أحمد عبد الله

بسم الله الرحمن الرحيم

المقدّمة:

في عالم جافا وتحديدًا عالم الويب كلّما أراد المبرمجون برمجة موقع احتاجوا إلى طريقة لربط قاعدة البيانات بالموقع, حتى أصدرت Sun أوّل إصدار لEJB أو Enterprise JavaBeans عام 1999م, والذي مكّن المبرمجين من وصف البناء الدّاخلي للمواقع بسهولة أكبر وفصل عناصر الموقع بحيث يكون الموقع متعدد الإطارات Multitiered, ومكّنهم من التعامل مع قاعدة البيانات باستخدام الكائنات من خلال Entity Bean, فمن المعروف أن فصل قاعدة البيانات عن منطق البرنامج Application Logic هو مطلب أساسي في المواقع, والذي يساعد على تسهيل التّعديل على منطق البرنامج أو على قاعدة البيانات دون أن يؤثّر التغيير كثيرًا في الإطار الآخر, وفي عام 2001 أصدر Gavin King بيئة عمل Hibernate والتي وفّرت إمكانيات أفضل لل Persistence, وبالطّبع سمحت بربط قاعدة البيانات بكائنات جافا POJO أو Plain Old Java Object عن طريق ما يسمى بORM أو Object Relational Mapping, وطريقة عمل الORM هي ربط صفّ جافا بجدولٍ في قاعدة البيانات العلائقية Relational Database بحيث يرتبط كل حقل Field في الصفّ بعمودٍ في الجّدول, فعندما يتمّ إضافة كائنٍ جديدٍ وحفظه سيتمّ إضافة سطر جديد في الجّدول المرتبط بالصفّ, وهكذا لباقي العمليات الأساسية من حذف وتعديل واستعلام أيضًا, ويمكن استخدام Hibernate في برامج الحاسوب أيضًا, لكن ما ينقص ذلك هو إمكانية تخزين الصفوف المعقّدة, ففي Hibernate لا تتوفر إلّا إمكانية ربط البيانات البدائية Primitives مع قاعدة البيانات (نص, عدد, تاريخ, إلخ), وهذا ما جاءت صفوف لتحلّه, وتضيف عليه الكثير بالطّبع, فصفوف لا توفر إمكانية تخزين البيانات وحسب بل وتطبيق مختلف الأوامر التي توفّرها SQL بطريقة مختلفة وأسهل وباستخدام جافا فقط.

مميّزات قاعدة بيانات صفوف:

تتميّز صفوف بالمميزات التالية:

* إمكانيّة تخزين البيانات بصيغتها الكائنيّة مباشرةً, وإمكانية استرجاعها والقيام بمختلف العمليات عليها.
* إمكانيّة عمل قاعدة البيانات لتخدم أكثر من برنامج أو موقع (بعيد أو على نفس الجهاز) في نفس الوقت.
* توفّر نظام مستخدمين لحماية قاعدة البيانات مع إمكانية تخصيص الصّلاحيات لكلّ مستخدمٍ.
* إمكانيّة تقسيم البيانات وتوزيعها.
* إمكانيّة استخدام بعض الميزات لتسهيل استخدام قاعدة البيانات, مثل المرتبات والشروط.
* إمكانيّة استخدام ملفّ XML لتسهيل إعداد قاعدة البيانات وتشغيلها.
* توفّر دعم مباشر لJSP وتوفير وسوم جاهزة للاستخدام.
* إمكانيّة إنشاء الأوامر المختلفة وتنفيذها على قاعدة البيانات.

الفصل الأوّل: أساسيات قاعدة البيانات

سنتعلّم في هذا الفصل أساسيات استخدام قاعدة البيانات, وستكون بعد قراءة هذا الفصل قادرًا على استخدام قاعدة البيانات استخدامًا بسيطًا, ويفضّل أن يكون مستواك متوسطًا في لغة جافا.

**دورة حياة قاعدة البيانات:**

بعد تنزيل قاعدة بيانات صفوف يجب عليك إضافتها إلى مسار الصّفوف ClassPath, بعدها يمكنك استخدامها لإنشاء, تشغيل, تنفيذ الأوامر على قواعد البيانات.

تتبع صفوف نهجًا كما في باقي قواعد البيانات لاستخدامها, وهذا ما يوضحه المخطّط في الأسفل:











**أوّلًا: إنشاء قاعدة البيانات**

يتمّ تمثيل قاعدة البيانات بالصّفّ *Database* في الحزمة *org.sofof,* ويمكن إنشاء قاعدة البيانات من خلال الدّالّة الإستاتيكيّة createDatabase والّتي تقبل ملفّ قاعدة البيانات غير المُنشأ كمعامل, وإن كان الملفّ الممرّر موجودًا بالفعل ستعيد الدّالّة القيمة خاطئ False ولن تقوم بعمل شيءٍ, أما في غير تلك الحالة فستعيد القيمة صحيح True وستقوم بإنشاء قاعدة البيانات, وفي حال حدوث أي خطأ إدخالٍ وإخراجٍ IO ستطلق الّدالّة الاستثناء SofofException. وهذا مثالٌ يبيّن ذلك:

import java.io.File;

import org.sofof.SofofException;

import org.sofof.Database;

public class CreateDatabase{

public static void main(String[] args){

try{

Database.createDatabase(new File("[C:/Users/LENOVO](about:blank) PC/Desktop/sofofDB"));

}catch(SofofException ex){

System.err.println(ex.getMessage());

}

}

}

وعند فتح المجلد sofofDB ستجد ما في الصورة التالية:

يحتوي الملفّ binds على إعدادات قاعدة البيانات ومعلوماتٍ خاصةٍ بها.

**ثانيًا: خادم صفوف**

يقوم خادم صفوف بتوفير خدمات قاعدة البيانات, ويُوجد في الحزمة *org.sofof* واسمه *Server,* وهو عبارةٌ عن *Thread,* وله بانيان, الأوّل بلا معاملاتٍ, والثّاني يطلب معاملين, أوّلهما هو مجلد قاعدة البيانات, والثاني هو رقم المنفذ الّذي سيستمع الخادم له, ويمكن تشغيل الخادم من خلال الدّالّة *startUp* والّتي قد تطلق الاستثناء *SofofException* إذا حدثت مشكلة إدخالٍ وإخراجٍ أثناء قراءة إعدادات قاعدة البيانات, وهذا مثالٌ يبين طريقة تشغيل الخادم:

import java.io.File;

import org.sofof.Server;

import org.sofof.SofofException;

public class ServerStartUp{

public static void main(String[] args) throws SofofException{

Server server = new Server(new File("[C:/Users/LENOVO](about:blank) PC/Desktop/sofofDB"), 6969);

server.startUp();

}

}

عند تشغيل الخادم سيقوم بقراءة إعدادات قاعدة البيانات من مجلد قاعدة البيانات, ثم سيستمع إلى المنفذ الممرّر وهو في حالتنا هذه 6969, وبما أنّ الخادم هو *Thread* ويتمّ وضعه تلقائيًا في حالة الخفيّة Daemon فإنّ الخادم سيُغلق تلقائيًا عند إغلاق جميع الدّوالّ الّتي لا تحمل الصّفة Daemon, ولمنع ذلك يمكننا استخدام الدّالّة *setDaemon* وتمرير قيمة خاطئٍ لها, وعندها لن يتمّ إغلاق الخادم تلقائيًا, لذلك يجب عليك عند الانتهاء من استخدام قاعدة البيانات إغلاق الخادم عبر الدالة *shutdown* وإلّا لن يُغلق البرنامج وسيعمل الخادم في خلفية.

**نظام المستخدمين:**

لدى صفوف نظام إدارةٍ للمستخدمين يتمّ تعيينه في كلّ مرةٍ يتمّ تشغيل الخادم, ويتألف هذا النّظام من مستخدمين يُعبّر عنهم بالصّفّ *User* الموجود في الحزمة *org.sofof.permission.* يتعرّف نظام إدارة المستخدمين على المستخدم من خلال اسم المستخدم وكلمة المرور, ويتمّ تحديدهما من خلال باني الصّفّ *User* الوحيد الّذي يستقبل اسم المستخدم كمعاملٍ أوّلٍ وكلمة المرور كمعاملٍ ثانيٍ ثمّ يتمّ إضافته للخادم عبر الدالة getUsers().add()*,* ويكن تحديد صلاحيّات المستخدم عبر النّافذة *SofofSecurityManager* الّتي تُستخدم مع *SecurityManager,* وسيتمّ شرحها في فصلٍ كاملٍ لاحقًا , لكن يكفي حتى الآن معرفة أنّه سيُسمح للمستخدم بالقيام بجميع العمليات مادام لم يتمّ تحديد صلاحياته عبر *SofofSecurityManager*, وهذا مثالٌ يوضّح ما سبق:

import org.sofof.Server;

import org.sofof.SofofException;

import org.sofof.permission.User;

import java.io.File;

public class AdminUser{

public static void main(String[] args) throws SofofException{

Server server = new Server(new File("[C:/Users/LENOVO](about:blank) PC/Desktop/sofofDB"), 6969);

User rami = new User("rami", "manaf");

server.getUsers().add(rami);

server.startUp();

}

}

**ثالثًا: الجلسات**

بعد إعداد الخادم وإضافة المستخدمين إليه ثمّ تشغيله لم يتبقّى لنا إلّا خطوةٌ واحدةٌ لاستخدام قاعدة البيانات, وتتمثّل هذه الخطوة بفتح اتصالٍ مع الخادم بحيث يتمّ إرسال الأوامر إلى الخادم لتنفيذها, ويسمى هذا الاتصال بالجلسة *Session* والّتي توجد في الحزمة *org.sofof*, ولبدء جلسةٍ علينا إنشاء كائن قاعدة بياناتٍ *Database* ثمّ تنفيذ الدّالّة *startSession*, ولإنشاء كائن قاعدة بياناتٍ جديدٍ علينا تمرير معاملين, أوّلهما عنوان المضيف, وثانيهما رقم المنفذ, ويمثّل عنوان مضيف قاعدة البيانات عنوان الشّبكة الخاصّ به, فمثلًا يمكن للعنوان أن يكون نطاق الخادم site.com, أو عنوان بروتوكول الشّبكة IP كما في مواقع الويب أو الشّبكات المحلّيّة, أو قد يكون localhost إذا كان الخادم يعمل على نفس الجهاز, أمّا رقم المنفذ فيجب أن يكون نَفسه رقم المنفذ الّذي يعمل عليه الخادم, وبعد إنشاء كائن قاعدة البيانات يجب تنفيذ الدّالّة *startSession* الّتي تطلب معاملين, أوّلهما هو المستخدم الذي سيتمّ تسجيل الدخول عبره, فإن كان المستخدم غير مسجلٍ عند الخادم ستُطلق الدّالّة الاستثناء *SofofException,* أمّا الثّاني فهو عن هل سيكون الاتصال بقاعدة البيانات من طبقة الاتصالات المحميّة SSL أم لا, وهناك دالّةٌ أخرى تطلب المستخدم فقط ولن يكون الاتّصال في تلك الحالة من طبقة الاتصالات المحميّة *,* وبما أنّ الجلسة هي عبارةٌ عن اتّصالٍ مفتوحٍ فيجب إغلاقها بعد الانتهاء من استخدامها عبر الدّالة close, ويمكن فتح الجلسة داخل try-with-resource, وهذا مثالٌ يشرح ذلك:

import org.sofof.Server;

import org.sofof.SofofException;

import org.sofof.Database;

import org.sofof.Session;

import org.sofof.permission.User;

import org.sofof.permission.AdminRule;

import java.io.File;

public class Login{

public static void main(String[] args) thorws SofofException{

Server server = new Server(new File("[C:/Users/LENOVO](about:blank) PC/Desktop/sofofDB"), 6969);

User rami = new User("rami", "manaf");

server.getUsers().add(rami);

server.startUp();

Database db = new Database("localhost", 6969);

Session session = db.startSession(new User("rami", "manaf"), false);

session.close();

}

}

**رابعًا: تنفيذ الأوامر**

تقسّم الأوامر في صفوف إلى تنفيذيّةٍ *Executable* واستعلاميّةٍ *Query*, أمّا التنفيذيّة فمُهمّتها القيام بالعمليّات على الكائنات في قاعدة البيانات, والاستعلاميّة تقوم بالاستعلام عن الكائنات في قاعدة البيانات, ويتمّ تنفيذ الأوامر على قاعدة البيانات من خلال الجلسة, فلتنفيذ أمرٍ تنفيذيٍّ نقوم بتمريره إلى الدالة *execute* الموجودة في كائن الجلسة والّتي تعيد عدد الكائنات المُتأثّرة بالأمر, أما للاستعلام عن البيانات فنقوم بتمرير الاستعلام إلى الدّالّة *query* والّتي تعيد قائمةً بالكائنات المُستعلم عنها, وتوجد الأوامر في الحزمة *org.sofof.command.*

تعتمد صفوف على أسماء الرّبط في تخزين البيانات, واسم الرّبط هو اسم تربط الكائنات فيه, ويمكن تخزين كائنات أكثر من صفٍّ في نفس اسم الرّبط, ولكي أقرّبها أكثر يمكنني أن أشبّه اسم الرّبط بالجدول في قواعد بيانات SQL, أمّا الأعمدة فتتمثّل في الصّفوف Classes المختلفة المخزّنة في نفس اسم الرّبط.





**الأمر ربط:**

يستخدم الأمر التّنفيذيّ *Bind* لربط الكائنات باسم ربطٍ معينٍ, ويمكن لاسم الرّبط أن يكون باللغة العربيّة أو بأيّ لغةٍ أخرى, ويشترط ألّا يحوي أيًا من هذه الرّموز المحصورة بين القوسين ( \ | / \* : > <), ويشترط أيضًا أن تكون جميع الكائنات الّتي تريد حفظها تستخدم النّافذة *Serializable* هي وجميع حقولها*,* وأن يكون لها حقلٌ من النّوع الطّويل long وبالاسم serialVersionUID,.

للأمر *Bind* بانيان, أحدهما يستقبل المعامل المتسلسل Varargs *Object*, والباني الثّاني يستقبل قائمةً كمعامل *List*, وهذه الطّرق المختلفة مهمّتها الأساسية ربط الكائنات الممرّرة, ولربط الكائنات باسم ربطٍ معيّنٍ يجب استخدام الدّالّة to والّتي تستقبل *String* الّتي تمثّل اسم الرّبط, وهذا مثالٌ يبيّن ربط نصٍّ باسم الرّبط المنشورات:

import org.sofof.command.Bind;

import org.sofof.permission.User;

import java.io.File;

public class Binding {

public static void main(String[] args) throws SofofException {

Database.createDatabase(new File("sofof"));

Server s = new Server(new File("sofof"), 6969, false);

s.getUsers().add(new User("rami", "secret"));

s.startUp();

try (Session sess = new Database("localhost", 6969).startSession(new User("rami", "secret"), false)) {

sess.execute(new Bind("قاعدة بيانات صفوف رائعة").to("المنشورات"));

}

}

}

تسمح قاعدة بيانات صفوف بعدم تعيين اسم ربطٍ, وعندها سيكون اسم الرّبط SofofNoName ويعتبر هذا الاسم من أسماء الرّبط المحجوزة, ويكون اسم الرّبط هو بلا اسم عندما تكون قيمته هي ألّا قيمة أو عندما يتكوّن من المسافات فقط.

**الأمر تحديد:**

تسمح قاعدة بيانات صفوف بالاستعلام عن البيانات المرتبطة, ويعتبر الأمر تحديد *Select* من الأوامر سهلة الاستخدام والّتي توفّر الكثير من الخدمات بسهولةٍ وسرعةٍ, وبما أنّه يستخدم النّافذة *Query* فيتمّ تنفيذه على الدّالّة query والّتي تعيد قائمةً بالكائنات المستعلم عنها, وللصفّ بانيان, أحدهما يستقبل صفًّا *Class*, ويستعلم هذا الباني عن الكائنات من الصّفّ الممرّر والمُرتبطة باسم الرّبط الممرّر عبر الدّالّة from, أمّا الباني الثّاني فيستقبل صفًّا ونصًّا تنفيذيًا, وسنتحدّث عن النصّ التّنفيذيّ بالتّفصيل في الفصول التّالية. هذا مثالٌ على الباني الأوّل للاستعلام عن المنشورات المسجّلة سابقًا:

import org.sofof.command.Select;

import org.sofof.permission.User;

import java.io.File;

public class Selection {

public static void main(String[] args) throws SofofException {

Database.createDatabase(new File("sofof"));

Server s = new Server(new File("sofof"), 6969, false);

s.getUsers().add(new User("rami", "secret"));

s.startUp();

Session sess = new Database("localhost", 6969).startSession(new User("rami", "secret"), false);

List<String> posts = sess.query(new Select(String.class).from("المنشورات"));

for(String post : posts){

System.out.println(post);

}

}

}

**الأمر إلغاء الربط:**

يستخدم الأمر *Unbind* لإلغاء ربط الكائنات (مسحها), ولهذا الصفّ ثلاثة بناةٍ, أحدهما يستقبل مصفوفة صفوفٍ كمعاملٍ Varargs *Class*, وعند استعمال هذا الباني سيقوم الأمر بإلغاء ربط جميع الكائنات الّتي تنتمي للصفوف الممرّرة والمرتبطة باسم الرّبط المحدد عن طريق الدّالّة from, أما الباني الثّاني فيستقبل مصفوفة كائناتٍ Varargs *Object* كمعامل, ويجب أن تكون الكائنات الممرّرة جميعها تنتمي لنفس الصفّ, وسيقوم الأمر حينها بإلغاء ربط جميع الكائنات الّتي تساوي Equal الكائنات الممرّرة, والثّالث يساوي الثاني في مبدأ عمله إلا أنّه يستقبل الكائنات على شكل قائمةٍ, وهذا مثالٌ يبيّن هذا الأمر:

import org.sofof.command.Unbind;

import org.sofof.permission.User;

import java.io.File;

import java.util.Arrays;

public class Unbinding {

public static void main(String[] args) throws SofofException {

Database.createDatabase(new File("sofof"));

Server s = new Server(new File("sofof"), 6969, false);

s.getUsers().add(new User("rami", "secret"));

s.startUp();

Session sess = new Database("localhost", 6969).startSession(new User("rami", "secret"), false);

sess.execute(new Unbind(String.class));

sess.execute(new Unbind("خالد").from("الأعضاء"));

sess.execute(new Unbind(Arrays.asList("تفاح", "برتقال")).from("الطعام الممنوع"));

}

}

**الأمر تحديث:**

عند قيام المستخدم بتغيير اسمه في موقعك ستحتاج لتغيير اسمه المسجّل في قاعدة البيانات, وهذه مهمّة الأمر تحديث *Update,* ويسمح لك هذا الأمر بتحديث كائنٍ أو أكثر في قاعدة البيانات, وبتحديد شروطٍ على الكائن المحدّث (سيتمّ شرح ذلك في فصولٍ لاحقةٍ) ,ولتحديد التّحديث الجديد يمكننا تمريره إلى الدالة set التي تستقبل كائنًا (يجب الانتباه إلى أنّ التّحديث يجب أن يكون من نفس صف المُحدّث ) , وللأمر بانيان, أحدهما يطلب صفًّا كمعاملٍ, ويقوم هذا الباني بتحديث كائنات ذلك الصفّ جميعًا المرتبطة باسم الرّبط المحدد إذا لم يتمّ وضع شروطٍ, أمّا الثّاني فيستقبل كائنًا كمعاملٍ, وسيقوم الأمر حينها بالبحث بين جميع الكائنات المرتبطة باسم الرّبط المحدد عن الكائنات الّتي تساوي هذا الكائن Equals, فإذا كان هناك مجموعةٌ من الكائنات الّتي تساوي هذا الكائن قام بتحديثها, ويجدر الانتباه إلى أنّه في هذه الحالة من الضّروريّ التعديل على الدالة equals حتى تتناسب مع الصفّ الذي تريد حفظ كائناته, فمثلًا يتميّز المستخدم في موقع ما برمزٍ تعريفيٍّ أو بعنوان بريده الإلكترونيّ. ,وهذا المثال يشرح الأمر:

public class Updating {

public static void main(String[] args) throws SofofException {

Database.createDatabase(new File("sofof"));

Server s = new Server(new File("sofof"), 6969, false);

s.getUsers().add(new User("rami", "secret"));

s.startUp();

Session sess = new Database("localhost", 6969).startSession(new User("rami", "secret"), false);

Student rami = new Student(0, "Rami");

sess.execute(new Bind(rami).to("students"));

rami.setName("Hani");

sess.execute(new Update(rami).from("students").set(rami));

}

public static class Student{

private int id;

private String name;

public Student(int id, String name) {

this.id = id;

this.name = name;

}

@Override

public boolean equals(Object obj) {

if(obj == null)return false;

if(!(obj instanceof Student))return false;

Student s2 = (Student)obj;

return s2.id == id;

}

public void setName(String name) {

this.name = name;

}

}

}

**الأمر التقاط:**

في بعض الحالات قد لا تحتاج لحفظ الصّفّ بالكامل لأنّ مهمّته الأساسيّة لا تكون تخزين البيانات, بالإضافة لكونك تحتاج لحفظ بيانات لن تتكرر كإعدادات برنامج مثلًا, ولنأخذ هذا الصّفّ كتوضيح:

public class Game {

private int score;

private Node world;

private Node player;

private String playerName;

public void start(){

//some code that load the world and the player character and start the game

}

public void killEnemy(){

score++;

}

}

يقوم هذا الصف عند إنشاء كائنٍ منه ثمّ تشغيله بتحميل شخصية اللاعب والعالم وبإعداد اللعبة بشكلٍ عامٍّ, ومهمة هذا الصّفّ ليست تخزين البيانات لكن سيكون من غير المنطقيّ إنشاء صفٍّ جديدٍ لنخزن فيه اسم اللاعب ونقاطه خاصّةً أننا لن نحتاج لتخزين الصّفّ أكثر من مرّةٍ, وهنا يأتي دور الأمر التقاط *Capture*, فمهمّته التقاط البيانات من حقولٍ معيّنةٍ في الصّفّ وحفظها في قاعدة البيانات, وعند تكرار عملية الحفظ سيقوم هذا الأمر تلقائيًّا بمسح البيانات القديمة وحفظ الجديدة, وهكذا لن يسمح لأكثر من لقطةٍ واحدةٍ للصفّ الواحد, ويكفي تحديد البيانات الّتي تريد حفظها بالعلامة *Annotation* المسمى *Particle,* وتوجد هذه العلامة بالحزمة com.sefr.sofof.annotation, وهكذا سيكون الصف السابق بعد تحديد ما نريد حفظه هكذا:

public class Game {

@Particle

private int score;

private Node world;

private Node player;  
 @Particle

private String playerName;

public void start(){

//some code that load the world and the player character and start the game

}

public void killEnemy(){

score++;

}

}

بعدما ينتهي اللاعب من اللعبة ويضغط على زر الخروج يمكننا تنفيذ الكود التالي لالتقاط لقطةٍ للصفّ:

Capture.capture(session, gameObject);

ستعيد هذه الدّالّة واحدًا إن كان هناك لقطةٌ قديمةٌ لنفس الصّفّ وصفرًا في غير ذلك, وما يحدث بالضّبط أنّ هذه الدّالّة تبحث عن جميع الحقول المحدّدة بالعلامة *Particle* سواءً بنفس الصّفّ أو بآبائه, وتقوم بتخزينها في قاعدة البيانات وتحديدًا يتمّ ربطها باسم الربط SofofCapture, ولتحميل البيانات مرّةً أخرى يجب عليك تنفيذ الدّالّة الإستاتيكيّة load والّتي تستقبل الجلسة كمعاملٍ أوّلٍ والصّفّ الّذي تريد تحميل لقطته كمعاملٍ ثانٍ, وستعيد الدّالّة ألّا قيمة Null في حالة عدم وجود لقطةٍ سابقةٍ أو كائن *Capture* في حال وجود لقطةٍ, ويوفّر هذا الكائن بعض البيانات عن اللقطة كزمن التقاط اللقطة بالملّي ثانية عبر الدّالّة captureTime, لكنّ الاستخدام الأساسيّ لهذا الصّفّ يتمّ من خلال استخدام الدّالّة copyTo, فبعد إنشاء الكائن Game سنحتاج لتحميل البيانات الّتي تمّ التقاطها, وذلك يتمّ بتمرير كائن الصّفّ Game للدالّة copyTo فيصبح الكود كالتّاليّ:

Game game = new Game();  
Capture capture = Capture.load(session, Game.class);  
capture.coptTo(game);

**النصّ التّنفيذيّ:**

تتلخّص مهمة النصّ التّنفيذيّ في أنّه يقوم بالعمليّات على الكائنات, إذ يمكنك تنفيذ الدّوالّ واستدعاء الحقول, ويتمّ استخدامه مثلما يتمّ استخدام لغة جافا, وهذه قواعد استخدامه:

* يجب أن يبدأ أيّ نصٍّ تنفيذيٍّ بالرمز #.
* يمكن كتابة أسماء الدّوالّ والحقول مباشرة.
* إذا أردت أن تعيد نفس الكائن بدون تنفيذ أي دالة أو استدعاء أي حقل عليه فلا تكتب بعد رمز # أيّ شيءٍ.
* يشترط كتابة القوسين في حال كتابة اسم دالّةٍ سواءً أكان للدالّة معاملاتٌ أم لا.
* إذا أردت تنفيذ دالّةٍ أو استدعاء حقلٍ من الكائن الناتج عن تنفيذ دالّةٍ أو من الحقل المستدعى سابقًا فيكفي كتابة نقطةٍ يتبعها اسم الحقل أو الدّالّة (كما في لغة جافا).
* عند استدعاء دالّةٍ بمعاملاتٍ فيجب وضع لواحق بحسب معاملات الدّالّة, فقد تحتاج لتمرير نصٍّ كنصٍّ, وقد تطلب الدّالّة كائنًا كمعاملٍ فتمرّره على شكل نص أو رقم, لذلك يجب الالتزام بما تطلبه الدّالّة من معاملاتٍ أثناء كتابة اللواحق, وفي حال لم تتم كتابة لاحقة فسيتمّ استخدام أنواع البيانات الافتراضيّة int وfloat للأعداد الّتي تحوي فاصلةً, وهذه هي كلّ لاحقةٍ ونوع البيانات الذي يقابلها.

|  |  |  |
| --- | --- | --- |
| الصفّ | الرمز | الشرح |
| String | S | تحاط بإشارتي اقتباس مزدوجتين |
| Character | C | تحاط بإشارتي اقتباس |
| Integer | I |  |
| Float | F | يجب أن تتواجد فاصلة عشرية (.) |
| Long | L |  |
| Short | H |  |
| Double | D |  |
| Byte | B |  |
| Object | O | يمكن كتابتها مع النص أو الحرف أو الرقم أو الرقم بفاصلة عشرية فقط |

إذا كنت تريد تمرير البيانات كبيانات بدائية فيجب كتابة اللاحقة بالأحرف الصغيرة, مثلًا لتمرير int فيجب عليك كتابة اللاحقة I وهكذا.

تستخدم بعض الأوامر النّصّ التّنفيذيّ, ولكلٍّ طريقته الخاصّة في استخدامه, وهذا ما سأبيّنه في السّطور القادمة:

الأمر Select:

يسمح هذا الأمر بتمرير النّصّ التّنفيذيّ عن طريق الباني الثاني, والذي يستقبل صفًّا في المعامل الأوّل والنّصّ في المعامل الثّاني, ويقوم الأمر بتنفيذ النّصّ على جميع الكائنات التي تم اختيارها وسيعيد الناتج من تنفيذ النّصّ عليها, فمثلًا إذا قمنا بتخزين كائنات المدارس جميعها, وأردنا استدعاء مدراء جميع المدارس, حينها سنقوم بكتابة السطر التّالي:

List directors = sess.query(new Select(School.class, "#getDirector()"));

**الشّروط:**

عند استخدام الأوامر لتنفيذ العمليات على كائناتك ستحتاج إلى وضع بعض الشّروط حتى لا تتم العمليات على كائنات التي لا تريدها, فمثلًا قد تحتاج للاستعلام عن المستخدمين الذين يسكنون في حماة, أو مثلًا قد تحتاج لتحديث معلومات الأشخاص الذين يملكون مبالغ تزيد عن مئة ألف ليرة, أو حتى تحديد خمسةٍ من المتسابقين الذين فازوا في إحدى المسابقات, ولفعل ذلك يجب أن نستخدم الشروط الموجودة في الحزمة org.sofof.command.condition, والتي تستخدم جميعها النافذة *Condition*, وتحوي تلك الحزمة العديد من الشروط الجاهزة للاستخدام.

الشّرط **BooleanCondition**:

هذا الشّرط من أبسط أنواع الشروط, ويعتمد على قيمة منطقية تحدد ما إذا كان الشرط ينطبق على جميع الكائنات أم لا, ولهذا الشرط بانيٌ واحدٌ يستقبل القيمة المنطقية تلك, فإذا كانت صحيح صحّ الشرط على جميع الكائنات, أما إن كانت خاطئ فالشرط لن ينطبق على أيٍّ من الكائنات, فمثلًا لو أردنا استخدام الشرط على الأمر *Select* من خلال تمرير الشرط عبر الدّالة where, فإذا كان الشرط يحمل القيمة خاطئ على سبيل المثال فلن يعيد الاستعلام أي كائن, والعكس بالعكس, وينطبق الأمر نفسه على الأمرين *Update* و *Unbind*, فإذا مررنا الشرط بالقيمة خاطئ إلى الأمر Update فلن يتمّ تحديث أيّ كائنٍ, وإذا مررنا نفس الشرط إلى الأمر *Unbind* فلن يتم إلغاء ربط أي كائنٍ, والعكس بالعكس أيضًا, وهذا مثالٌ يوضح ذلك:

import org.sofof.command.Update;

import org.sofof.command.condition.BooleanCondition;

import org.sofof.permission.User;

import java.io.File;

public class BooleanConditionTesting {

public static void main(String[] args) throws SofofException {

Database.createDatabase(new File("sofof"));

Server s = new Server(new File("sofof"), 6969, false);

s.getUsers().add(new User("rami", "secret"));

s.startUp();

Session sess = new Database("localhost", 6969).startSession(new User("rami", "secret"), false);

int affected = sess.execute(new Update(String.class).from("comments").set("unavilabel").where(new BooleanCondition(false)));

System.out.println(affected);

}

}

//result

0

الشّرط LimitCondition:

مهمّة هذا الشرط تحديد عدد الكائنات الّتي ستتأثّر, فباستخدام الأمر Unbind مثلًا بعد تمرير هذا الشرط إليه وتحديد العدد المطلوب من الكائنات -خمسة على سبيل المثال- سيتمّ إلغاء ربط خمسة كائنات, أما إذا كان عدد الكائنات أقل من العدد الممرّر فسيتمّ إلغاء ربط تلك الكائنات فقط, وهذا مثالٌ باستخدام الأمر *Select:*

import org.sofof.command.Select;

import org.sofof.command.condition.LimitCondition;

import org.sofof.permission.User;

import java.io.File;

public class LimitedProjects {

public static void main(String[] args) throws SofofException {

Database.createDatabase(new File("sofof"));

Server s = new Server(new File("sofof"), 6969, false);

s.getUsers().add(new User("rami", "secret"));

s.startUp();

Session sess = new Database("localhost", 6969).startSession(new User("rami", "secret"));

sess.query(new Select(Project.class).from("website").where(new LimitCondition(5)));

}

}

الشّرط ObjectCondition:

يعد هذا الشرط من أهم الشروط المستخدمة في قاعدة البيانات, إذ أنه يعتمد على النصوص التنفيذية لكتابة الشروط, مما يسمح بتنفيذ الدّوالّ أو استدعاء الحقول للتحقق من الشروط, ويمتلك هذا الشرط بانيًا واحدًا يستقبل ثلاث معاملاتٍ, الأوّل والثّالث هما *Object*, ويسمح لهذين المعاملين أن يكونا نصّا تنفيذيًا أو أي كائن آخر, أما الثّاني فهو العملية *Operation* الّتي سيتمّ إجراؤها على هذين الكائنين, والعملية هي تعداد Enum يحوي على التعدادات الآتية:

* Equal: وتستخدم للتأكد من تساوي قيمتين عبر الدّالّة equals أو عبر مساواة عددين.
* NotEqual: عكس تنفيذ Equal.
* Greater: تستخدم للتأكد من أنّ عددًا أكبر من عدد, وهي مخصّصةٌ للأعداد فقط.
* GreaterOrEqual: تستخدم للتأكد من أنّ عددًا أكبر أو يساوي عددًا آخر, وهي مخصّصةٌ للأعداد فقط.
* Less: : تستخدم للتأكد من أنّ عددًا أقل من عدد, وهي مخصّصةٌ للأعداد فقط.
* LessOrEqual: تستخدم للتأكد من أنّ عددًا أقلّ أو يساوي عددًا آخر, وهي مخصّصةٌ للأعداد فقط.

ولتوضيح ذلك يمكننا تخيّل أننا نقوم ببناء موقعٍ ونحتاج لاختيار المقالات الّتي تمّت قراءتها أكثر من مئة مرّةٍ, وبغضّ النظر عن طريقة عمل الصّفّ *Essay* الّذي سيمثّل المقالة في حالتنا هذه فإنّ الكود سيكون كالتّاليّ:

List<Essay> essays = session.query(new Select(Essay.class).from("essays").where(new ObjectCondition("#getViews()", Operation.Greater, 100)));

التعامل مع الشروط:

للنافذة *Condition* عدة دوالٍّ تسمح بربط الشروط ببعضها بعلاقات معينة, وهذه العلاقات لا تؤثر على الشرط نفسه إنّما تنتج شرطًا جديدًا يمثّل تلك العلاقة, وتلك العلاقات تتمثل بالعلاقات المنطقيّة المعروفة, وهي (and, or, xor, not), ويتمّ استخدامها في بعض الحالات المعقّدة, فمثلًا إذا أردنا اختيار اللاعبين الذين تجاوزوا المرحلة الخامسة ويمتلكون أقلّ من عشرة آلاف قطعةٍ نقديّةٍ لتضاعف نقودهم كهديّة فسيكون كود الاستعلام عنهم هكذا:

List<Player> players = session.query(new Select(Player.class).from("players").where(new ObjectCondition("#getLevel()", Operation.Greater, 5).and(new ObjectCondition("#getResources().getGold()", Operation.Less, 10000))));

**المرتّبات:**

عندما تريد الاستعلام عن أسماء الطلاب مرتبين حسب الأبجدية فيمكنك استخدام الدّالة sort من الصّفّ *Collections* المعادة من الاستعلام إذا كان الصف الذي استعلمت عنه يستخدم النافذة *Comparable,* لكن ماذا إن أردت ترتيب الطلاب حسب درجات كل طالب في مادة الرياضيات أو العلوم؟ لا يمكنك استخدام النافذة *Comparable* عدّة مرات, وسيكون بناء *Comparator* في كلّ مرّة تحتاج لترتيب كائناتك أمرًا مملًّا ويؤدّي إلى كودٍ غير نظيفٍ, لذلك توفّر صفوف إمكانيّة إضافة المرتبات لبعض الأوامر الّتي تدعم المرتّبات, والمرتّبات هي صفوفٌ تستخدم النافذة *Sorter* والّتي توجد وجميع المرتّبات في الحزمة org.sofof.command.sorter*,* ومن أهمّ المرتّبات المرتبّ *ObjectSorter*, ويسمح لك بترتيب الكائنات بالاعتماد على نصٍّ تنفيذيٍّ يشير إلى الكائن الذي سيتمّ ترتيب الكائنات وفقه, فمثلًا إن أردت ترتيب الطلاب وفق أسمائهم فما عليك إلا الإشارة لاسم الطالب بالنّصّ التّنفيذيّ, وهذا مثالٌ:

import org.sofof.bean.Student;

import org.sofof.command.Select;

import org.sofof.command.sorter.ObjectSorter;

import org.sofof.command.sorter.Order;

import org.sofof.permission.User;

import java.io.File;

public class Documentation {

public static void main(String[] args) throws SofofException {

Database.createDatabase(new File("sofof"));

Server s = new Server(new File("sofof"), 6969, false);

s.getUsers().add(new User("rami", "secret"));

s.startUp();

Session session = new Database("localhost", 6969).startSession(new User("rami", "secret"));

session.query(new Select(Student.class).from("students").sort(new ObjectSorter("#getName()", Order.Aescending)));

}

}

لاحظ عزيزي القارئ وجود معامل آخر بعد النّصّ التّنفيذيّ في باني المرتّب *ObjectSorter,* وهذا المعامل هو التعداد *Order* الموجود في نفس حزمة المرتّبات, ومهمة هذا التعداد تحديد الترتيب تصاعديًّا Aescending كان أم تنازليًّا Descending, ويجدر بالذكر أيضًا أنّ المرتّب *ObjectSorter* يستطيع ترتيب البيانات البدائية بالإضافة لكل الكائنات التي تستخدم النافذة *Comparable* والكائن LocalDate أيضًا. في بعض الحالات ستحتاج لترتيب الكائنات عدّة ترتيبات حسب أولويّة كلٍّ منها, ويكفي لفعل ذلك كتابة المُرتّب الأكثر الأَوْلَى متبوعًا بالدّالّة after الّتي تستقبِل المُرتّب الأقلّ أولويّةً.

**الانتقالات Transaction:**

عند قيامك بعدّة عملياتٍ على قاعدة البيانات مرّةً واحدةً فأنت تقوم بنقل قاعدة البيانات من حالةٍ لأخرى, لكن عند تنفيذ العمليّات على حداً ستكون قاعدة البيانات في وضعٍ غير صحيحٍ, فمثلًا إن أراد عميلٌ بحسابٍ به 5000ل شراء حاسوبٍ بمبلغ 5000ل فستقوم في البداية بالتحقق من كونه يملك المال الكافي ثمّ ستقوم بسحب المبلغ من حسابه كخطوةٍ ثانيةٍ ثمّ ستودع المبلغ في حساب البائع, لكن إن قمت بتنفيذ هذه العمليات على قاعدة البيانات كلًّا على حدًا قد يحاول العميل مثلًا شراء منتجٍ آخر أثناء تنفيذ تلك العمليّة سترى العمليّة الثّانية أن حسابه ما زال يملك 5000ل, وإن حدث أن تمّ شراءه الثاني أثناء شراءه الأول وقبل خصم سعر الحاسوب سيكون رصيده النهائيّ عددًا سالبًا, أي أنّه تمكّن من الشّراء بأكثر مما يملك, ولتفادي حصول مثل هذه الأمور يتم تنفيذ الخطوات العمليّة مرّةً واحدةً لضمان انتقال قاعدة البيانات من وضعٍ صحيحٍ إلى وضعٍ صحيحٍ آخر, وتوفّر صفوف الأمر AtomicProcess الّذي يسمح لك بتنفيذ عدّة أوامر مرّةً واحدةً من خلال تمريرها له بالتّرتيب إلى بانيه.

الفصل الثّاني: العلاقات بين الكائنات

ستتعلّم في هذا الفصل وضع العلاقات بين الكائنات المختلفة في قاعدة البيانات.

**أساس العلاقات:**

إن أساس أيّ علاقةٍ بين كائنين في صفوف هو المُعرّف ID الموجود في الحزمة org.sofof, وهذا الصف يصف موقع الكائن في قاعدة البيانات من حيث اسم الرّبط والصّف ورقم التّعريف المميّز الخاص به, وتعتمد العلاقات على كائن هذا الصف في الإشارة للكائنات الأخرى الّتي ترتبط معها, وتقوم صفوف بالكشف عن هذا الصف من خلال البحث في حقول Fields الكائن الممرّر لها, فإذا لم يكن لذلك الحقل أي قيمةٍ null ستقوم صفوف بتوليد معرفٍ خاصٍ به, أمّا إن وجدت أنّ له قيمةً فإن لم تكن تلك القيمة مستخدمةَ بالفعل ستتركها وإلا ستطلق استثناءً بأنّه لا يمكن وجود معرفين متطابقين.

public class Student{  
private ID id;

private String name;

//code...  
}

**تكوين علاقة بين كائنين:**

بعد تمرير كائنٍ وتوليد المُعرّف الخاص به سيتم حفظ ذلك الكائن مع مُعرّفه في قاعدة البيانات, لذلك فعلى المستخدم قراءة الكائن مرّةً أخرى للحصول على الكائن مع مُعرّفه, وهنا يجدر الانتباه إلى أنّ مقارنة التّطابق equal بين كائنين لا يجب أن تتمّ على أساس تطابق المُعرّف إلّا في حالة وجود معرّفٍ للكائنين. يوفّر الأمر Bind إمكانية استرجاع الكائنات بسهولة من خلال تنفيذه كاستعلام بعد تنفيذه كأمر, وسيعيد في تلك الحالة نفس الكائنات الّتي تم ربطها بناءً على الدّالّة equals, وبعد إرجاع الكائن مع مُعرّفه يمكن ربطه بأي كائنٍ آخر كوضعه في أحد الحقول لكائنٍ آخر, ولا يشترط للكائن الآخر أن يكون له مُعرّفً, وحينها إن تمّ حفظ ذلك الكائن في صفوف واستُرجع مرّةً أخرى ستقوم قاعدة البيانات بالبحث في جميع حقول ذلك الكائن وحقول حقوله إلى آخر نقطةٍ بحثًا عن كائناتٍ تحمل مُعرّفًا, وعندما تجد أحدها ستقوم بقراءة آخر قيمةٍ موجودةٍ في قاعدة البيانات لذلك الكائن بناءً على مُعرّفه, وستضع القيمة الجديدة مكان القديمة, وهذا مثالٌ يوضّح ذلك:

public class Teacher implements Serializable{

private static final long serialVersionUID = 124530;

private ID id;

private String name;

public Teacher(String name) {

this.name = name;

}

@Override

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if (obj == null) {

return false;

}

if (getClass() != obj.getClass()) {

return false;

}

final Teacher other = (Teacher) obj;

if (id != null && other.id != null && !id.equals(other.id)) {

return false;

}

if (!Objects.equals(this.name, other.name)) {

return false;

}

return true;

}

}

public class Course implements Serializable {

private static final long serialVersionUID = 5465235;

private ID id;

private String name;

private Teacher teacher;

public Course(String name, Teacher teacher) {

this.name = name;

this.teacher = teacher;

}

//getters and setters

@Override

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if (obj == null) {

return false;

}

if (getClass() != obj.getClass()) {

return false;

}

final Course other = (Course) obj;

if (id != null && other.id != null && !id.equals(other.id)) {

return false;

}

if (!Objects.equals(this.name, other.name)) {

return false;

}

if (!Objects.equals(this.teacher, other.teacher)) {

return false;

}

return true;

}

}

//code after all preparations for database and connections

Teacher samer = new Teacher("Samer");

Bind teacherBind = new Bind(samer);

session.execute(teacherBind);

samer = (Teacher)session.query(teacherBind).get(0);

Course math = new Course("Math", samer);

session.execute(new Bind(math));

samer.setName("Majed");

session.execute(new Bind(samer));

System.out.println(session.query(new Select(Course.class).where(new ObjectCondition("getName()", Operation.Equal, "Math"))).get(0).getTeacher().getName());

//output: Majed

الفصل الثّالث: أفضل استخدام لقاعدة البيانات

لا يمكنك الغوص في المياه الضّحلة كما لا يمكنك استعمال صفوف دون بعض الإرشادات التي ستمدّك بالهواء تحت الماء. في هذا الفصل ستتعلّم كيف تستخدم صفوف أحسن استخدام والطريقة المثلى لتخزين البيانات فلا تفوت إلقاء نظرة .

**قواعد حفظ البيانات:**

قواعد حفظ البيانات هي مجموعة من القوانين الّتي يسمح لك الالتزام بها بالتعامل بسهولة وسلاسة مع قاعدة البيانات, وعند حدوث أيّ خطأ راجع هذه القواعد.

قواعد كتابة الصّفوف:

* دائمًا ابدأ باستعمال النافذة *Serializable.*
* لا تنسى أبدًا كتابة المتغيّر serialVersionUID.
* يجب أن تكون جميع الحقول الّتي سيتمّ كتابتها تستخدم النافذة *Serializabl.*
* يفضّل أن يتّبع الصّفّ قواعد حبوب قهوة جافا Java Beans حتى يستطيع المستخدم استخدام نفس الصفوف في بيئات عمل مثل Spring.
* إن كان لديك حقولٌ لا تريد حفظها فلا تكتبها من الأساس, لكن يمكنك استخدام الكلمة المفتاحية transient قبل كتابة نوع المتغيّر إن كنت لم تحدد طريقة كتابة الصّفّ بنفسك.
* حدد بينة الصّفوف الخاصة بك واحرص على عدم تكرار كتابة السطور باستخدام OOP.
* اكتب جميع العمليات التي تحتاجها على شكل دوالٍّ في الصّفّ.
* يفضّل دائما تحديد طريقة كتابة الصّفّ بنفسك, وسيأتي شرح هذه النّقطة لاحقًا.
* عدّل دائمًا دائمًا على الدّالّة equals بحيث تعتمد على حقل معين يميّز الكائن عن باقي الكائنات, لأن بعض أوامر صفوف تعتمد على هذه الدّالّة, ولن يعمل الأمر بالطريقة الصحيحة إن لم يتمّ التعديل عليها.

تعدّ الخطوات السابقة نتيجة الفهم المتكامل لعمليّة كتابة الكائنات, لذلك فهمها مرتبط بفهم تلك العمليّة, والهدف الأساسيّ من تلك العمليّة كتابة بيانات الصّفّ لتتحوّل لبياناتٍ ثنائيّةٍ, ويكون ذلك عن طريق عدّة خطوات تبدأ بالتأكّد من أنّ الصف يستخدم النافذة Serializable أو أنّ أحدًا من آبائه يستخدمها, ثم يتمّ كتابة جميع حقول التي لم تُسبق بالكلمة transient أو final أو static, ويفضل دائمًا تحديد طريقة الكتابة وذلك عن طريق خطوات بسيطة ومهمّة في نفس الوقت, ويتمّ ذلك عن طريق كتابة دالّتين مسؤولتين عن كتابة وقراءة كائنات الصّفّ وهما readObject الّتي تستقبل ObjectInputStream كمعامل, وتقوم بقراءة بيانات الكائن, و writeObject الّتي تستقبل ObjectOutputStream كمعامل, وتقوم بكتابة بيانات الكائن, فعند تحديدهما ستتمكّن من إضافة وحذف أيّ حقلٍ من الصّفّ بسهولةٍ دون الحاجة لمسح البيانات المخزّنة باستخدام الصّفّ قبل التعديل [[1]](#footnote-0) , أمّا عن كيفيّة كتابة أو قراءة بيانات الكائن فذلك يتمّ عن طريق المعاملات الممررة, فالمعامل ObjectOutputStream في الدّالّة writeObject يحتوي على دوالّ لكتابة شتى أنواع البيانات البدائية والّتي يبدأ اسمها بالكلمة write يتبعها نوع البيانات مثلًا: writeFloat وwriteInt إلخ, وتختلف String عن البقيّة بكونها تكتب بالدالّة writeUTF, وفي حال كون الصّف يحتوي على حقولٍ لكائنات فيمكن كتابة تلك الكائنات عبر الدّالّة writeObject, ونفس الشيء تمامًا في حالة قراءة الكائن, فيتمّ استخدام ObjectInputStream والدّوالّ الّتي تبدأ بالكلمة read.

وهذا المثال يوضح تلك القواعد جميعها.

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.Serializable;

import java.util.Objects;

public class Student implements Serializable {

private static final long serialVersionUID = 4702347432L;

private String name;

private int age;

private String nationality;

public Student(String name, int age, String nationality){

this.nationality = nationality;

this.age = age;

this.name = name;

}

public int getAge() {

return age;

}

public String getName() {

return name;

}

public String getNationality() {

return nationality;

}

public void setAge(int age) {

this.age = age;

}

public void setName(String name) {

this.name = name;

}

public void setNationality(String nationality) {

this.nationality = nationality;

}

private void readObject(ObjectInputStream in) throws IOException {

name = in.readUTF();

age = in.readInt();

nationality = in.readUTF();

}

private void writeObject(ObjectOutputStream out) throws IOException {

out.writeUTF(name);

out.writeInt(age);

out.writeUTF(nationality);

}

@Override

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if (obj == null) {

return false;

}

if (getClass() != obj.getClass()) {

return false;

}

final Student other = (Student) obj;

if (!Objects.equals(this.name, other.name)) {

return false;

}

if (!Objects.equals(this.age, other.age)) {

return false;

}

if (!Objects.equals(this.nationality, other.nationality)) {

return false;

}

return true;

}

}

إن لاحظت, فإن تسلسل كتابة عمليات الكتابة هو نفسه تسلسل عمليات القراءة في الدّالّتين readObject وwriteObject, وهذا الأمر ضروريٌ جدًا.

**خدعٌ وتحايلٌ:**

يمكنك التّحايل على بعض المشاكل الّتي قد تواجهك في المستقبل أثناء كتابة الصّفوف الّتي سيتمّ حفظها في قاعدة البيانات, وهذا شرحٌ لأهمّ تلك المشاكل:

**حفظ حقولٍ لا تستخدم النّافذة *Serializable:***

عند بناء الصفوف المعقّدة ستجد أنّ أغلب حقولك لا تستخدم النّافذة *Serializable*, ويمكن التحايل على تلك المشكلة بتحديد طريقة كتابة وقراءة الكائن ثمّ استخراج البيانات الأساسيّة منه وحفظها, وعند القراءة فتقوم بقراءة البيانات الأساسيّة ثم تحويلها للكائن, فمثلًا خصائص JavaFX أو JavaFX Properties مثل *SimpleStringProperty* لا تستخدم النّافذة *Serializable*, ويمكن التحايل على ذلك الأمر في دالّتي قراءة وكتابة الكائن هكذا:

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.Serializable;

import javafx.beans.property.SimpleStringProperty;

public class Person implements Serializable {

private static final long serialVersionUID = 8036435743L;

private SimpleStringProperty name;

public Person(){  
 name = new SimpleStringProperty(this, "name", "");  
 }

private void writeObject(ObjectOutputStream out) throws IOException {

out.writeUTF(name.getValue());

}

private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {

this.name = new SimpleStringProperty(this, "name", in.readUTF());

}

}

**إضافة وحذف حقولٍ من الصّفّ:**

بعد إطلاق مشروعك ورغبتك بتحديث المشروع قد تحتاج لحذف أو إضافة حقل أو أكثر, مما سيتسبب في إطلاق استثناءات أثناء قراءة البيانات القديمة, لكن يمكن التحايل على ذلك إذا كنت قد حددت طرق القراءة والكتابة من البداية, ويتطلب فهم ذلك فهم سبب انطلاق تلك الاستثناءات, فعند حذف حقل ما مثلًا لن يظهر أيّ استثناء (طبعًا في حالة قد تمّ تحديد طرق القراءة والكتابة), لكنّ ذلك الأمر قد يحدث بعض الأخطاء مستقبلًا إن أُضيف مكانه أو بعده حقولٌ أخرى في دالتي القراءة والكتابة, لذلك لتفادي هذا الأمر يمكنك كتابة القيمة الافتراضية لذلك الحقل في دالة الكتابة مثلا: 0, نص فارغ, إلخ, وعند القراءة اقرأ قيمته لكن لا تضعها في أيّ متغيّر (أهملها), أمّا عند إضافة حقلٍ آخر فيجب وضعه في نهاية عمليات الكتابة في دالّة الكتابة, وفي نهاية عمليات القراءة في دالّة القراءة, لكن عندما يقرأ البرنامج البيانات القديمة سيضطر لقراءة الحقل الجديد الغير مسجلٍ, وعندها سيُطلق الاستثناء *EOFException,* وهذا الاستثناء يطلق عند محاولة القراءة بعدما انتهت البيانات, ولتفادي هذا الاستثناء يمكننا إحاطة جميع الحقول الجديدة بالتركيب try-catch بحيث تلتقط ذلك الاستثناء, ويمكنك في تركيب catch وضع قيم افتراضيّة للحقول الجديدة, وهكذا عندما تقوم دالّة القراءة بقراءة البيانات القديمة سيُطلق الاستثناء وسيتمّ إمساكه بحيث لا يتسبب بإيقاف البرنامج, وهذا مثالٌ على صفٍّ والصّفّ بعد تحديثه:

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.Serializable;

public class User implements Serializable {

private static final long serialVersionUID = 555646L;

private String username;

private String password;

private String image;

private void readObject(ObjectInputStream in) throws IOException {

username = in.readUTF();

password = in.readUTF();

image = in.readUTF();

}

private void writeObject(ObjectOutputStream out) throws IOException {

out.writeUTF(username);

out.writeUTF(password);

out.writeUTF(image);

}

}

وهذا الصفّ بعد حذف الحقل image وإضافة الحقل nickname:

import java.io.EOFException;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.Serializable;

public class User implements Serializable {

private static final long serialVersionUID = 555646L;

private String username;

private String nickname;

private String password;

private void readObject(ObjectInputStream in) throws IOException {

username = in.readUTF();

password = in.readUTF();

in.readUTF();

try{

nickname = in.readUTF();

}catch(EOFException ex){}

}

private void writeObject(ObjectOutputStream out) throws IOException {

out.writeUTF(username);

out.writeUTF(password);

out.writeUTF("");

out.writeUTF(nickname);

}

}

الفصل الرّابع: أدوات صفوف

بعد تعلّم كل ما سبق بقي الآن تعلّم كيفيّة استخدام بعض الأدوات الّتي ستساعدك أثناء عملك باستخدام صفوف والّتي توفرها قاعدة البيانات نفسها, وستتمكن بعد قراءة هذا الفصل من استخدامها.

**ملفّات XML:**

عند إعداد قاعدة البيانات يدويًّا ستحتاج لإنشاء قاعدة البيانات ثم إعداد الخادم وتشغيله ثم بدء جلسة لاستخدامها, وكلّ ذلك بالكود, ممّا يصعب كتابة كل ذلك في كلّ مرّة نحتاج فيها لاستخدام صفوف في مشروعٍ ما, لذلك توفّر صفوف إمكانيّة إعدادها في ملفّ XML, وتتضمّن إضافة المستخدمين للخادم وإنشاء قاعدة البيانات و إعداد وإنشاء اتصالٍ بقاعدة البيانات, وسنشرح ذلك على ملف XML يستخدم جميع تلك الميّزات.

<**sofof**

**xmlns:xsi**='http://www.w3.org/2001/XMLSchema-instance'

**xmlns**='http://sofof.org/xsd'>

<**server** **port**="6969" **ssl**="true">

<**database** **path**="data" **name**="sofof-database"/>

<**users**>

<**user** **name**="Rami" **password**="password"/>

<**user** **name**="user2" **password**="password"/>

</**users**>

</**server**>

<**sessions**>

<**session** **name**="web" **host**="website.org" **port**="7647">

<**user** **name**="supervisor5" **password**="\*%55554@"/>

</**session**>  
 <**session** **name**="local" **host**=”localhost” **port**="6969">

<**user** **name**="Rami" **password**="password"/>  
 </**session**>

</**sessions**>

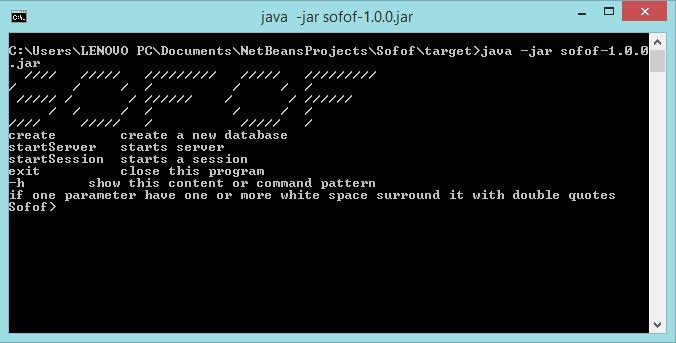
</**sofof**>

يتكوّن ملفّ XML من العقدة الرئيسيّة sofof التي تحمل بعض إعدادات XML المهمّة وقد تتسبب بخطأٍ عند عدم كتابتها, يأتي بعدها العقدتين الرئيسيّتين server وsession. تصف كلتا العقدتين الخادم والجلسات المُعدّة, ويمكن تحميل إعدادات الخادم من ملفّ XML عبر الدّالّة configure, أمّا الجلسات فيتم إعدادها عبر الدّالّة الإستاتيكيّة configure الموجودة في الصّفّ *Database*, وكما هو واضحٌ من ملفّ XML فإنّ لكلّ جلسةٍ اسمًا يميّزها, ويمكن استدعاء الجلسة بعد إعداد الجلسات بتمرير اسمها إلى الدّالّة الإستاتيكيّة getSession الموجودة في الصّفّ *Database* أيضًا, وستعيد الدّالّة اللا قيمة في حال كون الجلسة المرتبطة بذلك الاسم غير موجودةً, وهذا مثالٌ يوضّح ذلك:

new Server().configure().startUp();  
Database.configure();  
Session local = Database.getSession("local");

**مشغّل صفوف:**

كما في قواعد البيانات الأخرى فلصفوف مشغّلٌ أيضًا, ويمكن تشغيله من ملف jar الخاص بصفوف مباشرة, وذلك عبر تشغيله من موجّه الأوامر( CMD في Windows مثلًا), وستكون النتيجة كالتّالي:



كما هو واضحٌ من الصورة يمكنك تنفيذ أوامر قاعدة البيانات الأساسيّة, وذلك عن طريق استخدام الأوامر التي طُبعت على شاشة موجّه الأوامر, لكن بالإضافة للأوامر الأساسيّة فللأمرين startServer وstartSession خاصيّة مختلفة, فيمكنك بعد كتابة المعاملات الضروريّة لأحدهما (والتي يمكن عرضها من خلال كتابة اسم الأمر متبوعا بأمر إظهار التعليمات -h) تنفيذ الأوامر باستخدام لغة JavaScript, فعند تنفيذ الأمر startServer سيتم تعيين متغيّرٍ بالاسم server يحوي كائن *Server* الّذي تمّ إنشاؤه من الأمر startServer لكي تستطيع إضافة المستخدمين له ثمّ تشغيله, وكذلك الأمر بالنّسبة للأمر startSession الّذي يعيّن الكائن session.

الفصل الخامس: تخصيص صفوف

كلّ ما تعلّمته حتى الآن كان الاستخدام النّموذجي لقاعدة البيانات, أمّا ما ستتعلّمه في هذا الفصل فحدوده السّماء. ستستطيع بعد قراءة هذا الفصل فعل الآتي:

* إنشاء أوامرك التّنفيذيّة واستعلاماتك الخاصّة.
* استخدام النّصّ التّنفيذيّ في أوامرك.
* إنشاء شروطك الخاصّة.
* إنشاء مرتّباتك الخاصّة.

**أوامرك الخاصّة:**

إن أردت إنشاء أوامرك الخاصّة فعليك فهم طبيعة عمل الخادم وكيف ينفّذ الأوامر, وذلك يعتمد بالضرورة على رؤيتك لكيفيّة عمل قاعدة البيانات, فهي ليست مجموعة أوامر جاهزةٍ يتبنّاها الخادم, بل هي بيئة لتنفيذ الأوامر مهما كان مصدرها وطبيعة عملها, مما يسمح لأيّ شخصٍ بكتابة أوامره الّتي ستخدمه في أيّ مجالٍ يريده (تجميع البيانات الضّخمة, كتابة السّجلات, مشاركة البيانات, إلخ), ويعتمد الخادم بالأساس على الصّفّين *ListInputStream* و *ListOutputStream,* ووظيفة هذين الصّفّين تشبه إلى حدٍّ بعيدٍ وظيفة الأمرين *Bind* و *Select*, ويقوم الخادم بتمرير كليهما في حالة تنفيذ أمرٍ تنفيذيٍّ إلى الدّالّة execute و *ListInputStream* إلى الدّالّة query في حالة تنفيذ استعلام, مما يعطي الأمر إمكانيّة قراءة وكتابة البيانات من وإلى قاعدة البيانات, فالصّفّ *ListOutputStream* له دالّة لكتابة القوائم وهي write والّتي تستقبل القائمة واسم الربط والصّفّ على التّرتيب, وعند تنفيذها ستقوم بمسح جميع الكائنات الصّفّ الممرر المرتبطة باسم الرّبط ثم تكتب الكائنات الجديدة, أمّا *ListInputStream* فيسمح بالقراءة من خلال الدالّة read والّتي تستقبل اسم الرّبط والصّفّ الذي ستقرأ كائناته جميعها المرتبطة باسم الربط.

إنّ ما يحدث في حال استقبال الخادم أكثر من أمرٍ سواء أكان تنفيذيًّا أم استعلامًا يستوجب الاهتمام في حال أردت بناء أوامرك الخاصّة, فعندما يستقبل الخادم أكثر من طلب سيبدأ بتنفيذها مباشرةً, فإن كان هناك أمران يقومان بالقراءة من نفس الصفّ المرتبط باسم ربطٍ معيّنٍ (وهو أصغر وحدة في قاعدة البيانات) سيسمح لهما بالقراءة, أمّا في حالة تنفيذ أمرٍ تنفيذيٍّ سينتظر الجميع حتى ينتهي ذلك الأمر, وذلك يعود لكون الأمر التّنفيذيّ يستطيع القيام بالعديد من الأوامر والّتي تتطلب ثبات قاعدة البيانات أثناء تنفيذها, وذلك يتطلب منك ككاتب أوامر مراعاة سرعة تنفيذ الأمر التّنفيذيّ. نأتي الآن للمثال الذي سيوضح لك كل ما سبق:

import org.sofof.command.Executable;  
import org.sofof.ListInputStream;

import org.sofof.ListOutputStream;

public class Format implements Executable {

private String bind;

private Class[] classes;

public Example(String bind, Class... classes) {

this.bind = bind;

this.classes = classes;

}

@Override

public int execute(ListInputStream in, ListOutputStream out) throws SofofException {

int affected = 0;

for (Class clazz : classes) {

affected +=in.read(bind, clazz).size();

out.write(new LinkedList(), bind, clazz);

}

return affected;

}

}

هذا مثالٌ بسيطٌ لأمرٍ تنفيذيٍّ يقوم بمسح جميع الكائنات من عدّة صفوفٍ مرتبطةٍ بنفس اسم الربط دفعةً واحدةً, ويجب عند انتهاء الأمر إعادة عدد الكائنات المُتأثّرة. أمّا بالنّسبة للاستعلام فسنقوم بكتابة استعلامٍ يعيد خرائط بأسماء الطّلاب ومعدّل علامات كلّ واحدٍ:

import org.sofof.command.Operation;

import example.Student;

import example.Marks;

import org.sofof.command.Query;

import org.sofof.command.Select;

import org.sofof.command.condition.ObjectCondition;

import java.util.HashMap;

import java.util.LinkedList;

import java.util.List;  
  
class AVG implements Query{

@Override

public List query(ListInputStream in) throws SofofException {

List students = in.read("students", Student.class);

LinkedList maps = new LinkedList();

for(Student student : students){

maps.add(new HashMap(){{

put("name", student.getName());

List<Integer> marks = new Select(Mark.class, "#getMark()").from("marks").where(new ObjectCondition("#getName()", Operation.Equal, student.getName())).query(in);

float sum = 0;

for(int mark : marks){

sum += mark;

}

put("avg", sum/marks.size());

}});

}

return maps;

}

}

ما كان غير مألوفٍ لديك هو أنني نفّذت استعلامًا آخر داخل استعلامي بتمرير قارئ القوائم للدالّة query, وذلك طلبًا للسهولة والسّرعة, أمّا ما تبقى الآن فهو إضافة النّصّ التّنفيذيّ لأمرك, وهو بمنتهى السهولة, ويتمّ ذلك عبر الصّفّ *ExpressionExecuter* عبر الدّالّة الإستاتيكيّة execute, ومبدأ عمل النّصّ التّنفيذيّ هو تنفيذ النّصّ على كائن معيّنٍ يمرر للدالّة execute مع النّصّ التّنفيذيّ الذي سيُنفّذ على الكائن الممرر, وستعيد الدّالّة القيمة التي يعيدها تنفيذ النّصّ التّنفيذيّ على الكائن.

**تخصيص الشّروط:**

تعدّ عمليّة تخصيص الشّروط من أسهل العمليات وأمتعها, فصحيحٌ أنّ الشروط المتوافرة تكفي للاستخدام العادي لكن إنشاء الشّرط الخاص بك مطلبٌ مهمٌّ في حال بناء البرامج المعقّدة, ويعتمد الشّرط على دالّة واحدة تستقبل الكائن الّذي ستنفّذ عليه معايير الشّرط, وذلك الكائن هو الكائن المُستعلم عنه من قاعدة البيانات, والدالة هي check الّتي تعرّفها النافذة *Condition*, وتعيد تلك الدّالّة صحيحًا في حال انطبق الشّرط على الكائن وخاطئًا في حال لم ينطبق, وإن كنت تقوم ببناء أمرٍ فيمكنك إضافة خاصيّة إضافة الشروط لأوامرك, فبعد كتابة كود الأمر قم بتمرير الكائنات الّتي تناسب أمرك (قد يكونك أمرك استعلامًا كما في *Select* بحيث تكون الكائنات الّتي تجتاز الشّرط هي المُستعلم عنها, وقد يكون أمرك أمرًا تنفيذيًّا كما في *Unbind* بحيث تكون الكائنات الّتي تجتاز الشّرط هي الّتي سيتمّ إلغاء ربطها, أو غير ذلك من الحالات) واحدًا واحدًا إلى الدّالّة check التي ستنفّذها على الأمر الّذي مرّره المستخدم إلى أمرك, وبناءً على ما تعيده الدّالّة check قرر ماذا ستفعل بالكائن.

**تخصيص المُرتّبات:**

تخصيص المُرتبات هو أبسط العمليّات, فما هو إلا بضعة خطوات وستحظى بمُرتّبك الخاص, أوّل تلك الخطوات هي استخدام النافذة Sorter والّتي لها دالّةٌ وحيدةٌ وهي sort, وتستقبل هذه الدّالّة القائمة الّتي سيتمّ ترتيبها, فما عليك إلا ترتيب تلك القائمة بالشّكل الّذي تريد.

الخاتمة

إنّ ما درسته في هذا الكتاب هو خاصٌ بمن يستخدمون النسخة الأساسيّة من جافا Java SE, وسيتمّ إصدار كتابٍ آخر مبنيٍّ على هذا الكتاب ومخصصٍ لمبرمجي Java EE, وسيحتوي ذلك الكتاب على استخدام صفوف في بيئة Java EE واستخدام JSP Tags.

**تمّ بحمد الله تعالى**

1. صحيح, في حالة عدم تحديد طرق الكتابة والقراءة بنفسك فلن تتمكّن من استرجاع البيانات المخزّنة في الصّفّ قبل التعديل فذلك سيطلق استثناءً, لذلك فإن عمليّة تحديد طريقة القراءة والكتابة مهمّةٌ للمدى الطويل. [↑](#footnote-ref-0)