**Lab 9**

**Saving and restoring the Shop (and Sku and Cart) objects to and from a database**

This lab uses the H2 database [1]. The benefit of this database is that is can be used as an “in memory” database [2], i.e. be “pulled in” on the fly. This is beneficial for the lab setup, where we don’t have a fixed database installed.

**Add the Maven dependency**

In order to reference the H2 library in your code you need to import the relevant classes. Maven can load the H2 library dynamically (as we have done with other libraries in our application).

For Maven to load the libraries you need to declare a dependency in the pom.xml file (in the application root folder). The values are provided by the distributor, see references.

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<version>1.4.194</version>

</dependency>

Firstly, think about the data that you would like to persist to the database. At this point this is likely to come from the objects that you already have, such as Shop and Sku. Create the SQL statements for the tables, see Appendix A. These statements can later be loaded by the application in order to create the tables on the fly.

Then write a class that implements the methods for loading and saving these objects, e.g. H2Shop.

**Import the libraries**

In Java classes and methods for accessing and processing data stored in a data source (usually a relational database) are in the java.sql package. You need to import the relevant libraries:

import java.sql.\*;

Other libraries used in this example are:

import Lombok.NonNull; //Lombok NonNull annotation

import org.slf4j.Logger; //SLF4J Logger

import org.slf4j.LoggerFactory; //LoggerFactory

import java.io.IOException; //Exception handling

import java.util.Map; //

import java.util.Scanner; //

Details about these classes can be looked up in the relevant API documentation.

**Working with databases in Java**

The basic process for working with databases in Java is as follows [4]:

1. [Establish a connection to the database.](https://docs.oracle.com/javase/tutorial/jdbc/basics/processingsqlstatements.html#establishing_connections)
2. [Create a statement object.](https://docs.oracle.com/javase/tutorial/jdbc/basics/processingsqlstatements.html#creating_statements)
3. Using the statement object e[xecute the SQL query.](https://docs.oracle.com/javase/tutorial/jdbc/basics/processingsqlstatements.html#executing_queries)
4. Executing an SQL statement may produce a ResultSet object. [Process the ResultSet object.](https://docs.oracle.com/javase/tutorial/jdbc/basics/processingsqlstatements.html#processing_resultset_objects)
5. [Close the connection.](https://docs.oracle.com/javase/tutorial/jdbc/basics/processingsqlstatements.html#closing_connections)

In order to increase application security, we should use prepared statements when we pass values, i.e. input parameters, into the SQL statements. Together with filtering user input, which is often difficult to achieve due to the large number of ways of implementing an injection attack using prepared statements is the recommended way of preventing SQL Injection Attacks. Prepared statements are sent to the DBMS right away, where it is compiled. Therefore, the PreparedStatement object contains not just a SQL statement, but a SQL statement that has been precompiled. This means that when the PreparedStatement is executed, the DBMS can just run the PreparedStatement SQL statement without having to compile it first.

Precompiled SQL statements usually contain input parameters, that weren’t known at creation time. In addition, using prepared statements often improves the execution time of statement objects, especially when they are used repeatedly.

Create the prepared statement from the string representing the prepared statement. The java.sql.PreparedStatement API provides the following example of creating a prepared statement and setting the parameter values:

In the following example of setting a parameter, con represents an active connection:

PreparedStatement pstmt = con.prepareStatement("UPDATE EMPLOYEES

SET SALARY = ? WHERE ID = ?");

pstmt.setBigDecimal(1, 153833.00);

pstmt.setInt(2, 110592);

Many of the methods used for processing database statements require you to catch an SQLException, e.g. by putting the relevant code into a try-catch block and catching an SQLException.

**Connect to the database**

In your class set up a connection String, similar to the one below:

**public static final** String ***MEM\_DB*** = **"jdbc:h2:mem:shop"**;  
**public static final** String ***FILE\_DB*** = **"jdbc:h2:~/shop"**;

Create an instance variable of type Connection.  
  
**private** Connection **connection**;

Write a method that connects to the database.

**static** Connection getConnection(String dbFile) **throws** SQLException, ClassNotFoundException {  
 Class.*forName*(**"org.h2.Driver"**); *// this ensures the driver class is loaded whrn the DriverManager looks for an installed class. Weird but Java* **return** DriverManager.*getConnection*(dbFile, **"sa"**, **""**); *// default password, ok for embedded.*}

Write a method that closes the connection to the database.

*/\*\*  
 \* Close the connection to the database  
 \** ***@throws*** *H2ShopException if there is a database problem.  
 \*/*@Override  
**public void** close() {  
 **try** {  
 **if** (**connection** != **null**) {  
 **connection**.close();  
 **connection** = **null**;  
 }  
 } **catch** (SQLException e) {  
 **throw new** H2ShopException(e);  
 }  
}

**Create the tables**

Write a constructor that creates the tables.

*/\*\*  
 \* Create a new link to the shop database  
 \** ***@param dbFile*** *The type of backing to the database.  
 \** ***@throws*** *H2ShopException if there is a database problem.  
 \*/*

*In the constructor of this class*  
**public** H2Shop(String dbFile) {  
 **try** {  
 **connection** = *getConnection*(dbFile);  
 loadResource(**"/db/shop-spec.sql"**);  
 } **catch** (Exception e) {  
 ***LOG***.error(**"Can't find database driver: "** + e.getMessage());  
 **throw new** H2ShopException(e);  
 }  
}

**Write a method to store a Shop**

1. Firstly create the SQL statement that retrieves the data based on your tables and objects.
2. Secondly, execute the statement.
3. Thirdly, with the results, populate an object of the required type and
4. (fourthly) return the object.
5. Ensure to catch SQLException.

*/\*\*  
 \* Given a shop, store it in the database. THe previous shop with this name (if any) will be deleted  
 \** ***@param shop*** *The shop to store  
 \** ***@throws*** *H2ShopException if there is a problem with the database  
 \*/***public void** storeShop(@NonNull Shop shop) {  
 String shopName = shop.getShopName();  
 **try** (PreparedStatement ps = **connection**.prepareStatement(

**"DELETE FROM shop WHERE shopName = ?"**

);  
 PreparedStatement ps2 =

**connection**.prepareStatement(

**"INSERT INTO shop (shopName, skuId) VALUES (?,?)"**

))

{  
 ps.setString(1, shopName);  
 ps.execute();  
  
 **connection**.setAutoCommit(**false**);  
 **for** (Sku sku : shop.getSkus()) {  
 saveSku(sku);  
 ps2.setString(1, shopName);  
 ps2.setString(2, sku.getId());  
 ps2.execute();  
 }  
 **connection**.commit();  
 **connection**.setAutoCommit(**true**);  
 } **catch** (SQLException e) {  
 **throw new** H2ShopException(e);  
 }  
}

**Write a method to load a Shop**

1. Firstly create the SQL statement that retrieves the data based on your tables and objects.
2. Secondly, execute the statement.
3. Thirdly, with the results, populate an object of the required type and
4. (fourthly) return the object.
5. Ensure to catch SQLException.

*/\*\*  
 \* Given a shop name, load the shop into memory.  
 \** ***@param shopName*** *The name of the shop  
 \** ***@return*** *The stored shop. An empty shop if nothing was stored  
 \** ***@throws*** *H2ShopException if there is a database problem  
 \*/***public** Shop loadShop(String shopName) {  
 Shop shop = **new** Shop(shopName);  
 **try** (PreparedStatement ps = **connection**.prepareStatement(**"SELECT skuId FROM shop WHERE shopName = ?"**)) {  
 ps.setString(1, shopName);  
 ResultSet rs = ps.executeQuery();  
 **while** (rs.next()) {  
 String skuId = rs.getString(1);  
 Sku sku = loadSku(skuId);  
 shop.addSku(sku);  
 }  
 **return** shop;  
 } **catch** (SQLException e) {  
 **throw new** H2ShopException(e);  
 }  
  
}

**Write a method to save a Sku**

1. Firstly create the SQL statement that retrieves the data based on your tables and objects.
2. Secondly, execute the statement.
3. Thirdly, with the results, populate an object of the required type and
4. (fourthly) return the object.
5. Ensure to catch SQLException.

**public void** saveSku(Sku sku) {  
 String pss = **"INSERT INTO sku (skuId, title, description, thumbURL, thumbWidth, thumbHeight, imageURL, imageWidth, imageHeight, unitPrice) "** +  
 **"VALUES (?,?,?,?,?,?,?,?,?,?)"**;  
 **try** (PreparedStatement ps = **connection**.prepareStatement(pss)) {  
 ps.setString(1, sku.getId());  
 ps.setString(2, sku.getTitle());  
 ps.setString(3, sku.getDescription());  
 ps.setString(4, sku.getThumb().getUrl());  
 ps.setInt(5, sku.getThumb().getWidth());  
 ps.setInt(6, sku.getThumb().getHeight());  
 ps.setString(7, sku.getImage().getUrl());  
 ps.setInt(8, sku.getImage().getWidth());  
 ps.setInt(9, sku.getImage().getHeight());  
 ps.setString(10, sku.getUnitPrice().toString());  
 ps.execute();  
 } **catch** (SQLException e) {  
 **throw new** H2ShopException(e);  
 }  
}

**Write a method to load a Sku**

1. Firstly create the SQL statement that retrieves the data based on your tables and objects.
2. Secondly, execute the statement.
3. Thirdly, with the results, populate an object of the required type and
4. (fourthly) return the object.
5. Ensure to catch SQLException.

**public** Sku loadSku(String skuId) {  
 **final** String s = **"SELECT title, description, thumbURL, thumbWidth, thumbHeight, imageURL, imageWidth, imageHeight, unitPrice FROM sku WHERE skuId = ?"**;  
 **try** (PreparedStatement ps = **connection**.prepareStatement(s)) {  
 ps.setString(1, skuId);  
 ResultSet rs = ps.executeQuery();  
 **if** (rs.next()) {  
 **return new** Sku(skuId)  
 .setTitle(rs.getString(1))  
 .setDescription(rs.getString(2))  
 .setThumb(**new** SkuImage(rs.getString(3)).setWidth(rs.getInt(4)).setHeight(rs.getInt(5)))  
 .setImage(**new** SkuImage(rs.getString(6)).setWidth(rs.getInt(7)).setHeight(rs.getInt(8)))  
 .setUnitPrice(**new** BigDecimal(rs.getString(9)));  
 }  
 **return null**;  
 } **catch** (SQLException e) {  
 **throw new** H2ShopException(e);  
 }  
}

**Utility method**

**private void** loadResource(String name) {  
 **try** {  
 String cmd = **new** Scanner(getClass().getResource(name).openStream()).useDelimiter(**"\\Z"**).next();  
 **try** (PreparedStatement ps = **connection**.prepareStatement(cmd)) {  
 ps.execute();  
 } **catch** (Exception e) {  
 **throw new** H2ShopException(e);  
 }  
 } **catch** (IOException e) {  
 **throw new** H2ShopException(**"Can't open "** + name + **" to load db commands: "** + e.getMessage());  
 }  
}

**Storing a Cart**

Similarly you can write a method to store a Cart:

*/\*\*  
 \* Save a cart under a given key  
 \** ***@param cookieValue*** *The key under which to save the cart.  
 \** ***@param cart*** *The cart being saved.  
 \** ***@throws*** *H2ShopException is there is an SQL problem.  
 \*/***public void** saveCart(@NonNull **final** String cookieValue, @NonNull Cart cart) {  
 **try** (PreparedStatement ps = **connection**.prepareStatement(**"DELETE FROM cart WHERE userId = ?"**);  
 PreparedStatement ps2 = **connection**.prepareStatement(**"INSERT INTO cart (userId, skuId, quantity) VALUES (?,?,?)"**)) {  
 ps.setString(1, cookieValue);  
 ps.execute();  
  
 **connection**.setAutoCommit(**false**);  
 **for** (Map.Entry<String,Integer> entry : cart.getEntries()) {  
 ps2.setString(1, cookieValue);  
 ps2.setString(2, entry.getKey());  
 ps2.setInt(3, entry.getValue());  
 ps2.execute();  
 }  
 **connection**.commit();  
 **connection**.setAutoCommit(**true**);  
 } **catch** (SQLException e) {  
 **throw new** H2ShopException(e);  
 }  
}

**Write a method to load a stored Cart**

*/\*\*  
 \* Load a stored cart, given the key under which it is stored  
 \*  
 \** ***@param cookieValue*** *This can be the value of the session cookie (JSESSIONID value) or a user id if this is login-based.  
 \** ***@return*** *The cart stored under this key. An empty cart if there is none..  
 \** ***@throws*** *H2ShopException if there is a problem with the database.  
 \*/***public** Cart loadCart(@NonNull **final** String cookieValue) {  
 Cart cart = **new** Cart();  
 **try** (PreparedStatement ps = **connection**.prepareStatement(**"SELECT skuId, quantity FROM cart WHERE userId = ?"**)) {  
 ps.setString(1, cookieValue);  
 ResultSet rs = ps.executeQuery();  
 **while** (rs.next()) {  
 cart.add(rs.getString(1), rs.getInt(2));  
 }  
 **return** cart;  
 } **catch** (SQLException e) {  
 **throw new** H2ShopException(e);  
 }  
}

**Exceptions**

You can write your own exception for this class, otherwise you need to catch RuntimeExeception.

**public class** H2ShopException **extends** RuntimeException {  
 H2ShopException(Exception e) {  
 **super**(e);  
 }  
 H2ShopException(String s) {  
 **super**(s);  
 }  
 }  
}

**References**

**H2 tutorials**

[1] http://www.h2database.com/html/tutorial.html (also see the menu on the right hand side to navigate to other sections)

[2] http://www.h2database.com/html/features.html#in\_memory\_databases

[3] http://www.javatips.net/blog/h2-database-example

**Java Working with Databases**

[4] https://docs.oracle.com/javase/tutorial/jdbc/basics/processingsqlstatements.html

**Java Working with Prepared Statements**

[5] http://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html

**H2 Maven Dependency reference**

http://www.h2database.com/html/cheatSheet.html

**Open Web Application Security Project – Prevention of SQL Injection Attacks**

https://www.owasp.org/index.php/SQL\_Injection\_Prevention\_Cheat\_Sheet

**Appendix A – SQL Sample Statements for Creating the Tables**

**CREATE TABLE** IF **NOT EXISTS** sku (  
 skuId **VARCHAR**(255) **PRIMARY KEY**,  
 title **VARCHAR**(255),  
 description **VARCHAR**(4096),  
 thumbURL **VARCHAR**(255),  
 thumbWidth **SMALLINT**,  
 thumbHeight **SMALLINT**,  
 imageURL **VARCHAR**(255),  
 imageWidth **SMALLINT**,  
 imageHeight **SMALLINT**,  
 unitPrice **VARCHAR**(8)  
);  
  
**CREATE TABLE** IF **NOT EXISTS** shop (  
 id **int** AUTO\_INCREMENT **PRIMARY KEY**,  
 shopName **VARCHAR**(255),  
 skuId **VARCHAR**(255),  
 **FOREIGN KEY**(skuId) **REFERENCES** sku(skuId)  
);  
  
**CREATE TABLE** IF **NOT EXISTS** cart (  
 id **int** AUTO\_INCREMENT **PRIMARY KEY**,  
 userId **VARCHAR**(255), // can be JSESSIONID cookie **value for** now  
 skuId **VARCHAR**(255),  
 quantity **INT**);  
  
**CREATE** INDEX IF **NOT EXISTS** shop\_index **ON** shop(shopName);  
**CREATE** INDEX IF **NOT EXISTS** cart\_index **ON** cart(userId);

**Appendix B– SQL Sample Statements: the Cart class**

**import** lombok.Data;  
**import** lombok.NonNull;  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** java.io.Serializable;  
**import** java.math.BigDecimal;  
**import** java.text.NumberFormat;  
**import** java.util.LinkedHashMap;  
**import** java.util.Map;  
**import** java.util.Set;  
  
@Data  
**public class** Cart **implements** Serializable {  
 @SuppressWarnings(**"unused"**)  
 **static final** Logger ***LOG*** = LoggerFactory.*getLogger*(Cart.**class**);  
 **private static final long *serialVersionUID*** = -746135577060516953L;  
  
 **private final** Map<String,Integer> **entries**;  
  
 **public** Cart() {  
 **entries** = **new** LinkedHashMap<>();  
 }  
  
 @SuppressWarnings({**"unused"**})  
 **public int** getCount() {  
 **int** count = 0;  
 **for** (Map.Entry<String,Integer> entry : **entries**.entrySet()) {  
 count += entry.getValue();  
 }  
 **return** count;  
 }  
  
 @SuppressWarnings({**"unused"**})  
 **public boolean** hasContents() {  
 **return entries**.size() > 0;  
 }  
  
 **public** Set<Map.Entry<String,Integer>> getEntries() {  
 **return entries**.entrySet();  
 }  
  
 */\*\* Change the number ordered for a SKU. Can be negative. If the new  
 \* total is less or equal 0 then the sku is removed from the cart.  
 \*  
 \** ***@param skuId*** *The id of the sku  
 \** ***@param count*** *The change to be made  
 \*/* **public void** add(@NonNull String skuId, **int** count) {  
 Integer current = **entries**.get(skuId);  
 **if** (current == **null**) {  
 current = 0;  
 }  
 **int** sum = current + count;  
 **if** (sum <= 0) {  
 **entries**.remove(skuId);  
 } **else** {  
 **entries**.put(skuId, sum);  
 }  
 }  
  
  
 **public void** clear(@NonNull String skuId) {  
 **entries**.remove(skuId);  
 }  
  
 **private** BigDecimal cost(Shop shop) {  
 BigDecimal cur = **new** BigDecimal(**"0"**);  
 **for** (Map.Entry<String,Integer> entry : **entries**.entrySet()) {  
 cur = cur.add(cost(entry.getKey(), entry.getValue(), shop));  
 }  
 **return** cur;  
 }  
  
 **private** BigDecimal cost(String skuId, **int** quantity, Shop shop) {  
 Sku sku = shop.getSku(skuId);  
 **return** (sku == **null**) ? **new** BigDecimal(**"0"**) : sku.getUnitPrice().multiply(**new** BigDecimal(quantity));  
 }  
  
 **public** String costString(Shop shop) {  
 **return** NumberFormat.*getCurrencyInstance*().format(cost(shop));  
 }  
  
}

**Appendix C – SQL Sample Statements: the Shop class**

**import** com.fasterxml.jackson.annotation.JsonInclude;  
**import** lombok.Data;  
**import** lombok.NonNull;  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** java.io.IOException;  
**import** java.io.Serializable;  
**import** java.util.LinkedHashSet;  
**import** java.util.Set;  
  
@Data  
@JsonInclude(JsonInclude.Include.***NON\_NULL***)  
@SuppressWarnings(**"unused"**)  
**public class** Shop **implements** Serializable {  
 **static final** Logger ***LOG*** = LoggerFactory.*getLogger*(Shop.**class**);  
 **private static final long *serialVersionUID*** = -4478190721183244953L;  
  
 **private** String **shopName**;  
 **private** Set<Sku> **skus**;  
  
 **public static** Shop fromJSONString(@NonNull String s) {  
 **try** {  
 **return new** ShopMapper().readValue(s, Shop.**class**);  
 } **catch** (IOException e) {  
 **throw new** RuntimeException(e);  
 }  
 }  
  
 **public** Shop() {  
 **skus** = **new** LinkedHashSet<>();  
 }  
  
 **public** Shop(@NonNull String storeName) {  
 **this**();  
 **this**.**shopName** = storeName;  
 }  
  
 **public** Shop(@NonNull Shop shop) {  
 **this**.**shopName** = shop.**shopName**;  
 **this**.**skus** = **new** LinkedHashSet<>(shop.**skus**);  
 }  
  
 **public** Shop addSku(@NonNull Sku sku) {  
 **skus**.add(sku);  
 **return this**;  
 }  
  
  
 **public** Sku getSku(@NonNull String id) {  
 **for** (Sku sku : **skus**) {  
 **if** (id.equals(sku.getId())) {  
 **return** sku;  
 }  
 }  
 **return null**;  
 }  
  
 **public** String toJSONString() {  
 **try** {  
 **return new** ShopMapper().writeValueAsString(**this**);  
 } **catch** (Exception e) {  
 **throw new** RuntimeException(e);  
 }  
 }  
}

**Appendix C – SQL Sample Statements: the Sku class**

**import** com.fasterxml.jackson.annotation.JsonInclude;  
**import** lombok.Data;  
**import** lombok.NonNull;  
**import** lombok.experimental.Accessors;  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** java.io.IOException;  
**import** java.io.Serializable;  
**import** java.math.BigDecimal;  
**import** java.text.NumberFormat;  
  
@Data  
@JsonInclude(JsonInclude.Include.***NON\_NULL***)  
@Accessors(chain=**true**)  
@SuppressWarnings({**"unused"**})  
**public class** Sku **implements** Serializable {  
 @SuppressWarnings(**"unused"**)  
 **static final** Logger ***LOG*** = LoggerFactory.*getLogger*(Sku.**class**);  
 **private static final long *serialVersionUID*** = -4119412393370563146L;  
  
 */\*\* SKUs are unique on their ids. Should check when reading from JSON to make sure this constraint is observed \*/* **private** String **id**;  
 **private** String **title**;  
 **private** String **description**;  
 **private** SkuImage **thumb**;  
 **private** SkuImage **image**;  
 **private** BigDecimal **unitPrice**;  
  
 **private** Sku() {  
 **this**(**""**);  
 }  
  
 **public** Sku(@NonNull String id) {  
 **this**.**id** = id;  
 }  
  
 **public** String unitPriceString() {  
 **return** NumberFormat.*getCurrencyInstance*().format(getUnitPrice());  
 }  
  
 **public** String toJsonString() {  
 **try** {  
 **return new** ShopMapper().writeValueAsString(**this**);  
 } **catch** (IOException e) {  
 **throw new** RuntimeException(e);  
 }  
 }  
  
 @Override  
 **public int** hashCode() {  
 **return id**.hashCode();  
 }  
  
 @Override  
 **public boolean** equals(Object o) {  
 **if** (o **instanceof** Sku) {  
 Sku oSku = (Sku)o;  
 **return id**.equals(oSku.**id**);  
 }  
 **return false**;  
 }  
  
}

**Appendix E – SQL Sample Statements: the SkuImage class**

**import** com.fasterxml.jackson.annotation.JsonInclude;  
**import** lombok.Data;  
**import** lombok.experimental.Accessors;  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** java.io.Serializable;  
  
@Data  
@Accessors(chain=**true**)  
@JsonInclude(JsonInclude.Include.***NON\_DEFAULT***)  
@SuppressWarnings({**"unused"**})  
**public class** SkuImage **implements** Serializable {  
 @SuppressWarnings(**"unused"**)  
 **static final** Logger ***LOG*** = LoggerFactory.*getLogger*(SkuImage.**class**);  
 **private static final long *serialVersionUID*** = 3256793117596298468L;  
  
 **private int width**;  
 **private int height**;  
 **private** String **url**;  
  
 **public** SkuImage() {  
 **this**(**""**);  
 }  
  
 **public** SkuImage(String url) {  
 **this**.**url** = url;  
 }  
  
}

**Appendix F – The ShopMapper class**

**import** com.fasterxml.jackson.annotation.JsonAutoDetect;  
**import** com.fasterxml.jackson.databind.DeserializationFeature;  
**import** com.fasterxml.jackson.databind.ObjectMapper;  
**import** com.fasterxml.jackson.databind.module.SimpleModule;  
**import** com.fasterxml.jackson.databind.ser.std.ToStringSerializer;  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** java.math.BigDecimal;  
  
  
**class** ShopMapper **extends** ObjectMapper {  
 @SuppressWarnings(**"unused"**)  
 **static final** Logger ***LOG*** = LoggerFactory.*getLogger*(ShopMapper.**class**);  
 **private static final long *serialVersionUID*** = -3817083726149967662L;  
  
 ShopMapper() {  
 SimpleModule module = **new** SimpleModule();  
 module.addSerializer(BigDecimal.**class**, **new** ToStringSerializer());  
 registerModule(module);  
 setVisibility(getVisibilityChecker().withFieldVisibility(JsonAutoDetect.Visibility.***ANY***));  
 configure(DeserializationFeature.***FAIL\_ON\_UNKNOWN\_PROPERTIES***, **false**);  
 }  
}