



Simba SDK

Build a Java ODBC Driver for SQL-Capable Data Sources in 5 Days

Version 10.4

July 2026

Copyright

This document was released in July 2026.

Copyright ©2014-2026 insightsoftware. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission from insightsoftware.

The information in this document is subject to change without notice. insightsoftware strives to keep this information accurate but does not warrant that this document is error-free.

Any insightsoftware product described herein is licensed exclusively subject to the conditions set forth in your insightsoftware license agreement.

Simba, the Simba logo, SimbaEngine, and Simba Technologies are registered trademarks of Simba Technologies Inc. in Canada, the United States and/or other countries. All other trademarks and/or servicemarks are the property of their respective owners.

All other companies and product names mentioned herein are used for identification purposes only and may be trademarks or registered trademarks of their respective owners.

Information about the third-party products is contained in a third-party-licenses.txt file that is packaged with the software.

Contact Us

insightsoftware

www.insightsoftware.com

Contents

Copyright	2
Contents	3
About this Guide	5
Purpose	5
Advantages of Using the Product	5
Audience	5
Document Conventions	6
Knowledge Prerequisites	6
Product Overview	7
ODBC Standards	7
The Product Solution	7
About the JavaUltraLight Sample Connector	8
Day One	12
Install the Product	12
Set Environment Variables	13
Build the Sample Java ODBC Connector	13
Update the Windows Registry	17
Examine the Windows Registry	18
Connect to the Data Store	19
Set Up a Custom Project	20
Update the Registry for a Custom Connector	22
Day Two	25
Finding the TODO messages	25

Find or create the Java Virtual Machine	25
Set the Connector Name	25
Set the Connector Properties	25
Set the Logging Details	26
Check the Connection Settings	26
Establish a Connection	27
Update the Configuration File	27
Day Three	28
Create and Return Metadata Sources	28
Day Four	33
Enable Data Retrieval	33
Day Five	36
Productize Your Custom Connector	36
Create a Connector Configuration Dialog	37
Reference	38
Data Retrieval	38
Java Server Configuration	39
32-bit vs 64-bit ODBC Data Source Administrator	40
Bitness and the Windows Registry	40
Contact Us	42
Third-Party Trademarks	43

About this Guide

Purpose

This guide explains how to use the Product to create a custom Java ODBC connector for a data store that is SQL-capable. It explains how to customize the JavaUltraLight sample connector, which is included with the Product.

Using this sample connector is the quickest and easiest way to create a custom Java ODBC connector. At the end of five days, you will have a read-only connector that connects to your data store. This custom connector can be used as the foundation for a commercial DSI implementation.

Advantages of Using the Product

The ODBC specification defines a rich interface that allows any ODBC-enabled application to connect to a data store. In order to implement a connector that supports this specification, developers have to understand all the complexities of error checking, session management, and data conversion, then design their code in a robust and efficient manner. Developers must also understand how to optimize data retrieval in order to get maximum performance when connecting to large and complex data stores.

The Product, developed by experts in the field, is a complete implementation of the ODBC specification. It exposes an easy-to-use SDK that allows you to create a robust and efficient connector for your data store.

Build a Custom ODBC Connector in Five Days

Over the course of five days, this guide explains how to accomplish the following tasks:

1. Set up the development environment and build the sample connector.
2. Use the sample connector as a template to create a custom Java ODBC connector.
3. Make a connection to the data store.
4. Retrieve metadata.
5. Work with columns.
6. Retrieve data.
7. Rename and rebrand the custom Java ODBC connector.

In the JavaUltraLight connector, the areas of code that require modification are marked with “TODO” messages and a short explanation. Some of these changes customize the connector for your specific data store, while other changes rename the connector for your company or product.

Audience

The guide is intended for developers who want to use the Product to build a connector for a data store that is SQL-capable.

Document Conventions

Italics are used when referring to book and document titles.

Bold is used in procedures for graphical user interface elements that a user clicks and text that a user types.

Monospace font indicates commands, source code or contents of text files.

NOTE:

Indicates a short note appended to a paragraph.



Important:

Indicates an important comment related to the preceding paragraph.

Knowledge Prerequisites

To use the Product to build a custom ODBC connector, the following knowledge is helpful:

- Familiarity with the Java programming language.
- Ability to use the data store to which the connector you are developing will connect.
- An understanding of the role of ODBC technologies and driver managers in connecting to a data store.
- Exposure to SQL.

Product Overview

Applications, such as Crystal Reports and Tableau, use connectors to connect to data stores from which they read and write data. Applications support the ODBC protocol to enable connection with any connector that also supports ODBC. A connector exposes the ODBC protocol to the application and another API, such as SQL or a custom API, to the data store.

Note:

This guide explains how to create a Java ODBC connector for a data store that is SQL-capable. For more details, see [Simba SDK Driver Development](#).

ODBC Standards

ODBC is one of the most established and widely-supported APIs for connecting to and working with databases. A main component of this technology is the ODBC connector, which connects an application to the database.

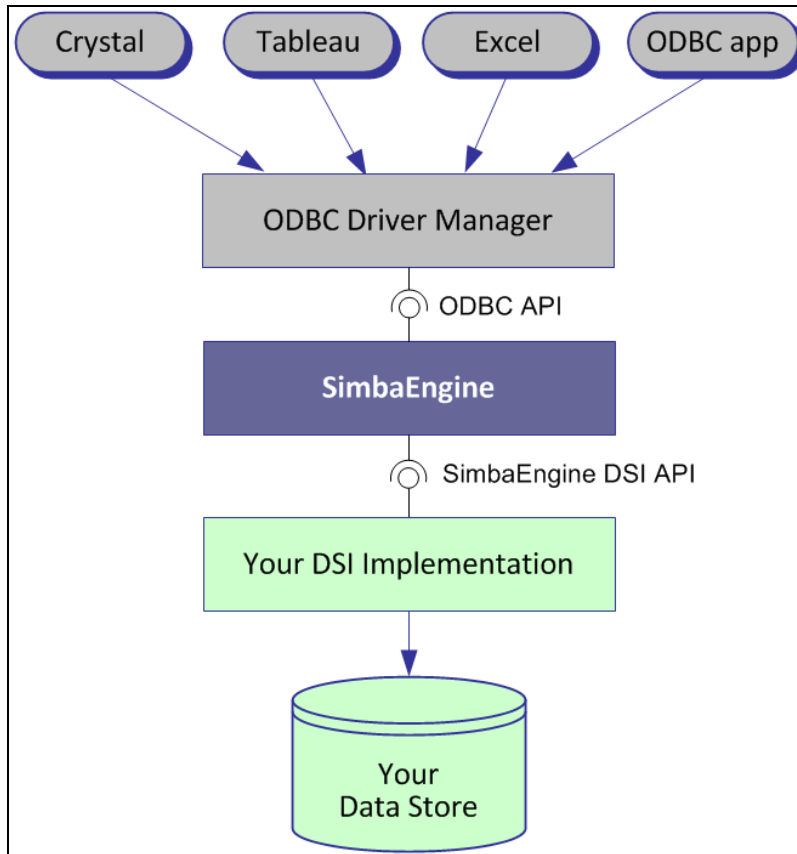
For a brief description of the ODBC standard, see <http://www.simba.com/resources/data-access-standards-library#!odbc>.

For complete information on the ODBC 3.80 specification, see the ODBC Programmer's Reference at [http://msdn.microsoft.com/en-us/library/ms714177\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms714177(v=vs.85).aspx).

The Product Solution

Connectors based on the Product leverage its error checking, session management, data conversion, optimization, and other low-level implementation details. The Product uses ODBC to communicate with the driver manager and a simple API (called the Data Store Interface API or DSI API) to communicate with the data store. The DSI API defines the primitive operations needed to access a data store.

The figure below shows a typical ODBC stack:



SDK developers create an implementation of a DSI (also known as a DSI Implementation or DSII) that applications use to access the particular data store in the process of executing an SQL statement. In the final executable, the components from Product take responsibility for meeting the data access standards while the custom DSI implementation takes responsibility for accessing the data store and translating it to the DSI API.

ODBC applications, such as Tableau or Microsoft Excel, use this executable when connecting to the data store in the process of executing an SQL statement.

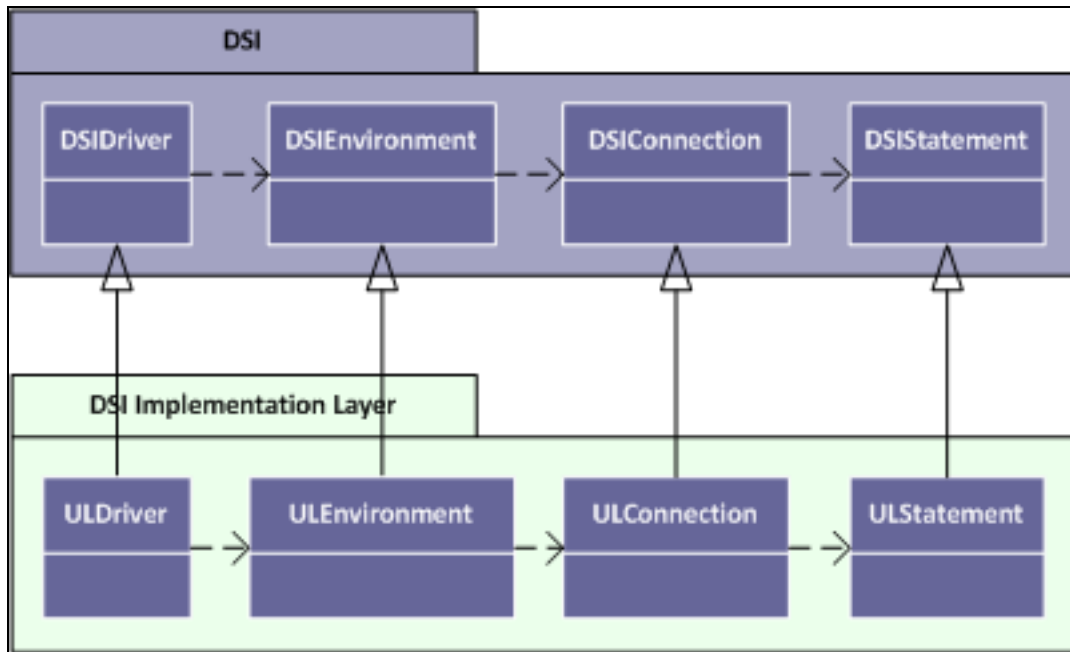
About the JavaUltraLight Sample Connector

The Product includes a sample connector that you can use as a template to create a custom ODBC connector for data stores that are SQL-capable. The JavaUltraLight connector is a sample DSI implementation of an ODBC connector, written in C++, which reads hard-coded data. The sample data is represented by a hard-coded table object, called the Person table. This table is always returned if an executed query contains SELECT. If the query does not contain SELECT, then a row count of 12 rows is returned.

Using the JavaUltraLight sample connector to prototype a DSI implementation for a custom data store helps developers understand how the Product works. By removing the shortcuts and simplifications implemented in the JavaUltraLight connector, you can use it as the foundation for a commercial DSI implementation and create a custom ODBC connector for a data store that is SQL-capable.

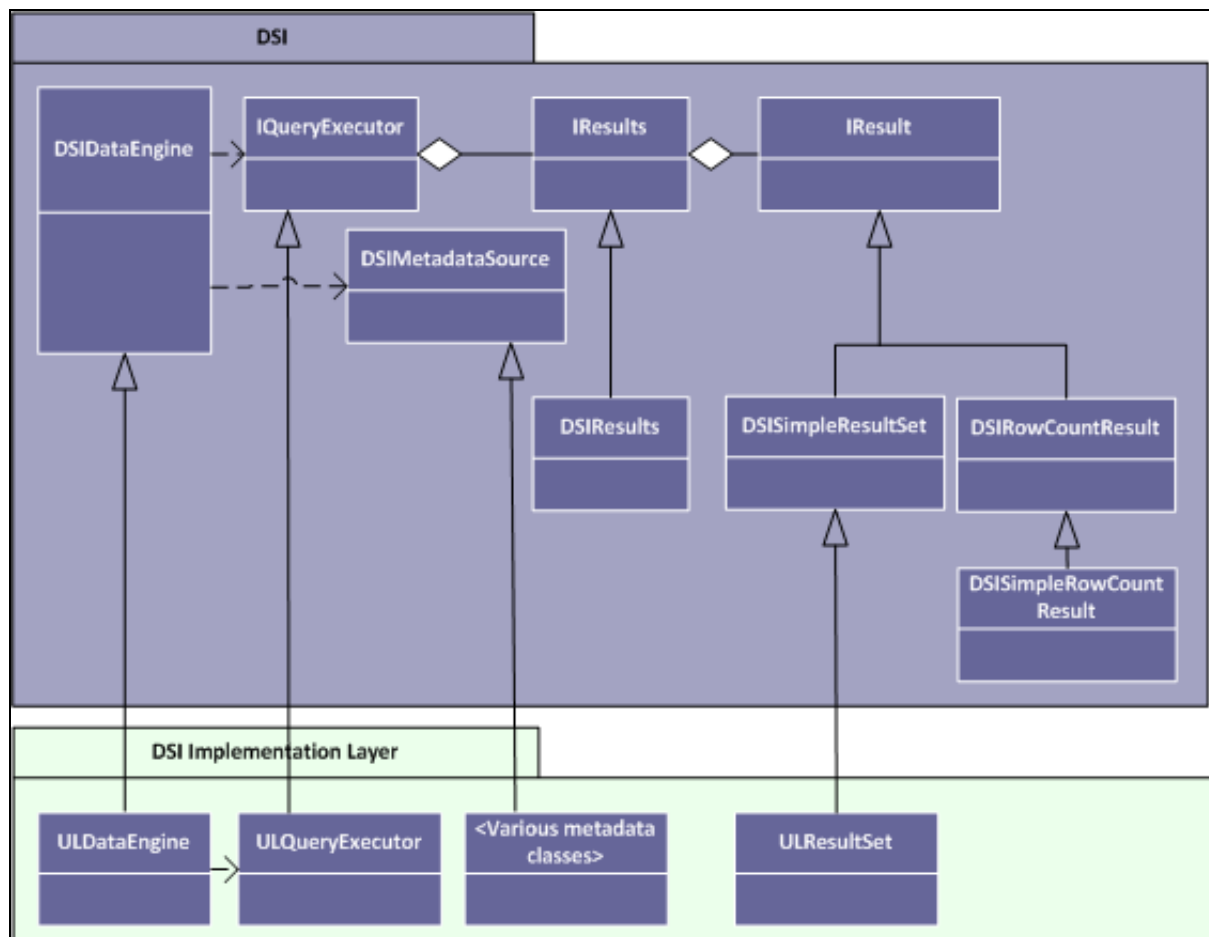
Implementation begins with the creation of a `DSIDriver` class which is responsible for constructing a `DSIEnvironment`. `DSIEnvironment` is used to construct a connection object (`DSIConnection` implementation) which is then used for constructing statements (`DSIStatement` implementations).

This concept is summarized in the figure below:

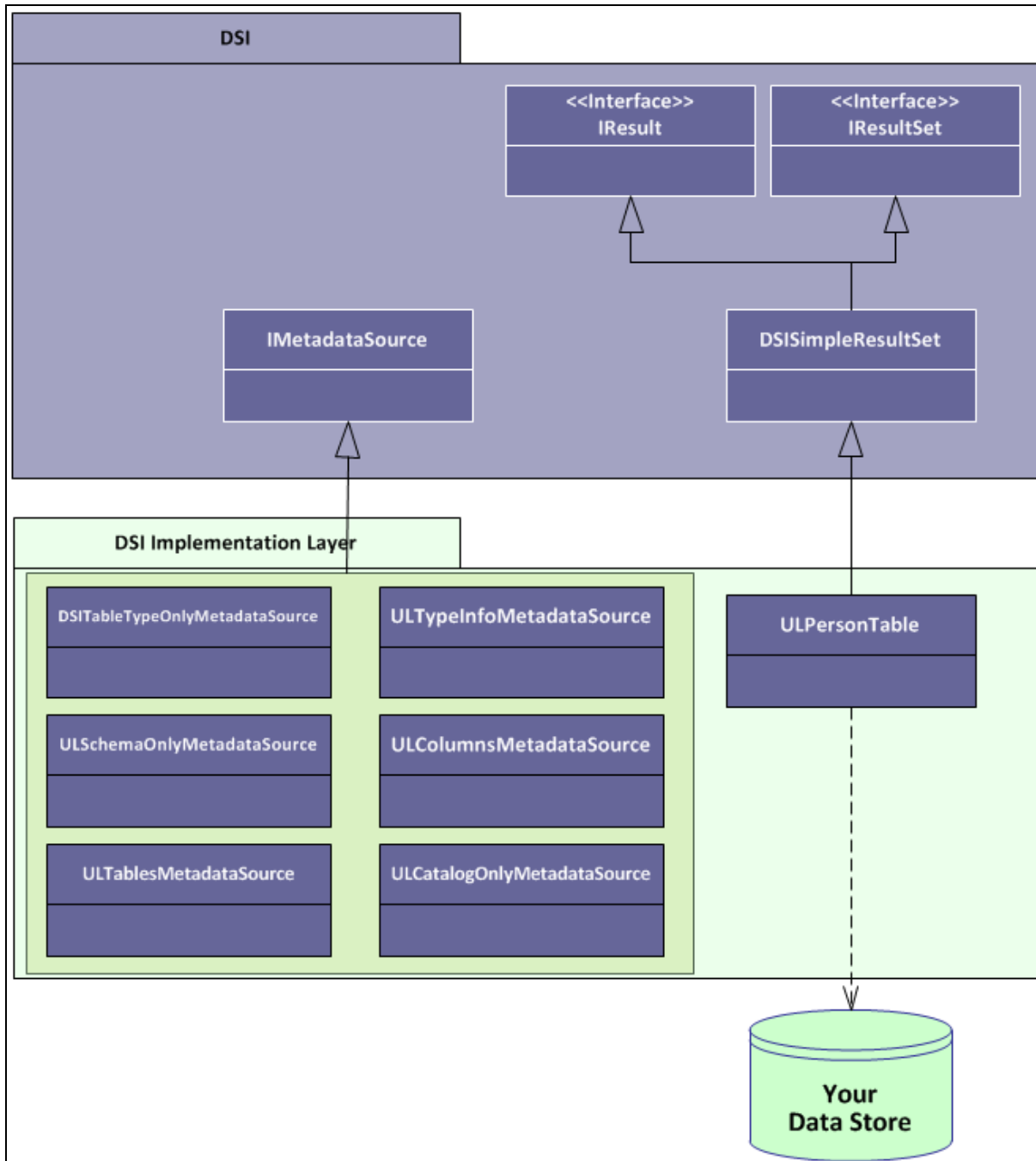


The `DSISStatement` implementation is responsible for creating a `DSIDataEngine` object, which then creates `IQueryExecutor` objects to execute queries and hold results (`IResults`), and `DSIMetadataSource` objects to return metadata information.

This concept is summarized in the figure below:



The final key part of the DSI implementation is to create the framework necessary to retrieve both data and metadata. A summary of this framework and the components implemented by the sample are shown in the figure below:



The `IResult` class is responsible for retrieving column data and maintaining a cursor across result rows.

To implement data retrieval, the custom `IResult` class interacts directly with the data store to retrieve the data and deliver it to the calling framework on demand. The `IResult` class should take care of caching, buffering, paging, and all the other techniques that speed data access.

The various `MetadataSource` classes provide a way for the calling framework to obtain metadata information.

Day One

The Day One instructions explain how to install the Product, compile the sample ODBC connector, and review the configuration information created at compile time.

After the sample ODBC connector is successfully compiled, it is used to retrieve data from the data source that is included with the Product. The sample ODBC connector is then used to create the framework for a custom ODBC connector, which is renamed and used to retrieve sample data.

At the end of the day, you will have compiled, built and tested your custom ODBC connector.

Install the Product

The Product installation package includes:

- The Product implementation in all supported platforms.
- Sample connectors for most supported platforms.
- PDF versions of the documentation.
- An HTML version of the C++ and Java API.

To install the Product:

1. Uninstall any previous versions of the Product.
2. Make sure that Visual Studio is closed.
3. Double-click the Product executable that corresponds to your version of Visual Studio, and then follow the instructions provided in the installation wizard.

The installer sets the following environment variables, where `[INSTALL_DIR]` is the Product installation directory.

Environment Variable	Value
SIMBAENGINE_DIR	<code>[INSTALL_DIR]\SimbaEngineSDK\10.4\DataAccessComponents</code>
SIMBAENGINE_THIRDPARTY_DIR	<code>[INSTALL_DIR]\SimbaEngineSDK\10.4\DataAccessComponents\ThirdParty</code>



Important: Important:

The Product environment variables are defined only for the user who ran the installation. If the SDK is installed as a regular user, Visual Studio must also be run as a regular user and not an administrator.

Components of the JavaUltraLight Sample connector

The Product provides the following components to help you write your custom ODBC connector in Java:

- The native C++ component, JavaUltraLightJNIDSI.

This component finds or creates an instance of the Java Virtual Machine (JVM) and provides the Java connector implementation name to the bridge between C++ and Java (SimbaJNIDSI).

- The Java component, JavaUltraLightDSII.

This component is the DSII.

Set Environment Variables

Set JAVA_HOME to the root of your JDK, and ensure PATH includes the path to `jvm.dll`

To set JAVA_HOME to the root of your JDK:

1. From the Start menu, select **System Properties > Advanced**, then click **Environment Variables**.
2. If JAVA_HOME is not listed in the System variables list:
 - a. Click **New**.
 - b. In the Variable name field, type **JAVA_HOME**.
 - c. In the Variable value field, type the path of the java home directory. For example, `C:\Program Files\Java\jdk1.7.0_09`.

To ensure PATH includes the path to `jvm.dll`

1. Find the location of the `jvm.dll` on your machine. For example, `C:\Program Files\Java\jdk1.8.0_60\jre\bin\server`.
2. From the Start menu, select **System Properties > Advanced**, then click **Environment Variables**.
3. Ensure the PATH environment variables includes the path to the `jvm.dll`.
4. To edit the PATH environment variable, click Edit and then add the path to the `jvm.dll` file to the end of PATH.

Note:

There is a difference between the 32-bit and 64-bit `jvm.dll` files. Ensure you set the path to the correct one.

Build the Sample Java ODBC Connector

The JavaUltraLight sample connector is included with the installation of the Product. It demonstrates one possible implementation of a read-only connector that reads data from tabbed Unicode files.

Build the C++ Component (UltraLightJNIDSI)

1. In Microsoft Visual Studio, click **File > Open > Project/Solution**.
2. In the Open Project dialog, navigate to the following folder:

[INSTALL_DIR]\SimbaEngineSDK\10.4\Examples\Source\JavaUltraLight\Source\JavaUltraLightJNIDSI

Where [INSTALL_DIR] is the installation directory.

3. Select the file `UltraLightJNIDSI_vs2013.sln`, and then click **Open**. This solution file contains the `UltraLightJNIDSI`, which is the connector's native component.
4. Click **Build > Configuration Manager**.
5. Click the drop-down arrow next to the **Active Solution Configuration** field, then select **Debug_MTDLL**, and click **Close**.
6. Click the drop-down arrow next to the **Active Solution Platform** field:
 - a. To build a 32-bit connector, select **Win32**.
 - b. To build a 64-bit connector, select **x64**.
7. Click **Close**.
8. Click **Build > Build Solution**.

The build appears in the following folder:

[INSTALL_DIR]\SimbaEngineSDK\10.4\Examples\Source\JavaUltraLight\Bin\<BUILD>\<RELEASE|DEBUG>\<CONFIGURATION>, where

- <BUILD> is a combination of your operating system, machine bitness, and compiler
- <RELEASE|DEBUG> is release or debug
- <CONFIGURATION> is mt if you select MTDLL as the solution configuration, otherwise md

For example:

C:\Simba Technologies\SimbaEngineSDK\10.4\Examples\Source\JavaUltraLight\Bin\Windows_vs2013\debug32md\UltraLightJNIDSI\ODBC32.dll

Build the Java Component (JavaUltraLight)

You can use ANT or tools that support ANT, such as the Eclipse IDE, to build the Java component. As an example, these instructions describe how to build `JavaUltraLight` using the Eclipse IDE.

1. Ensure the **JAVA_HOME** environment variable is pointing to the root of your JDK, and the **PATH** environment variable contains the path to `jvm.dll`. For more information, see [Set Environment Variables](#).
2. Use Eclipse to import the `JavaUltraLight` project as an existing project into your workspace.

The project files are located under [INSTALL_DIR]\Examples\Source\JavaUltraLight\Source\JavaUltraLightDSII



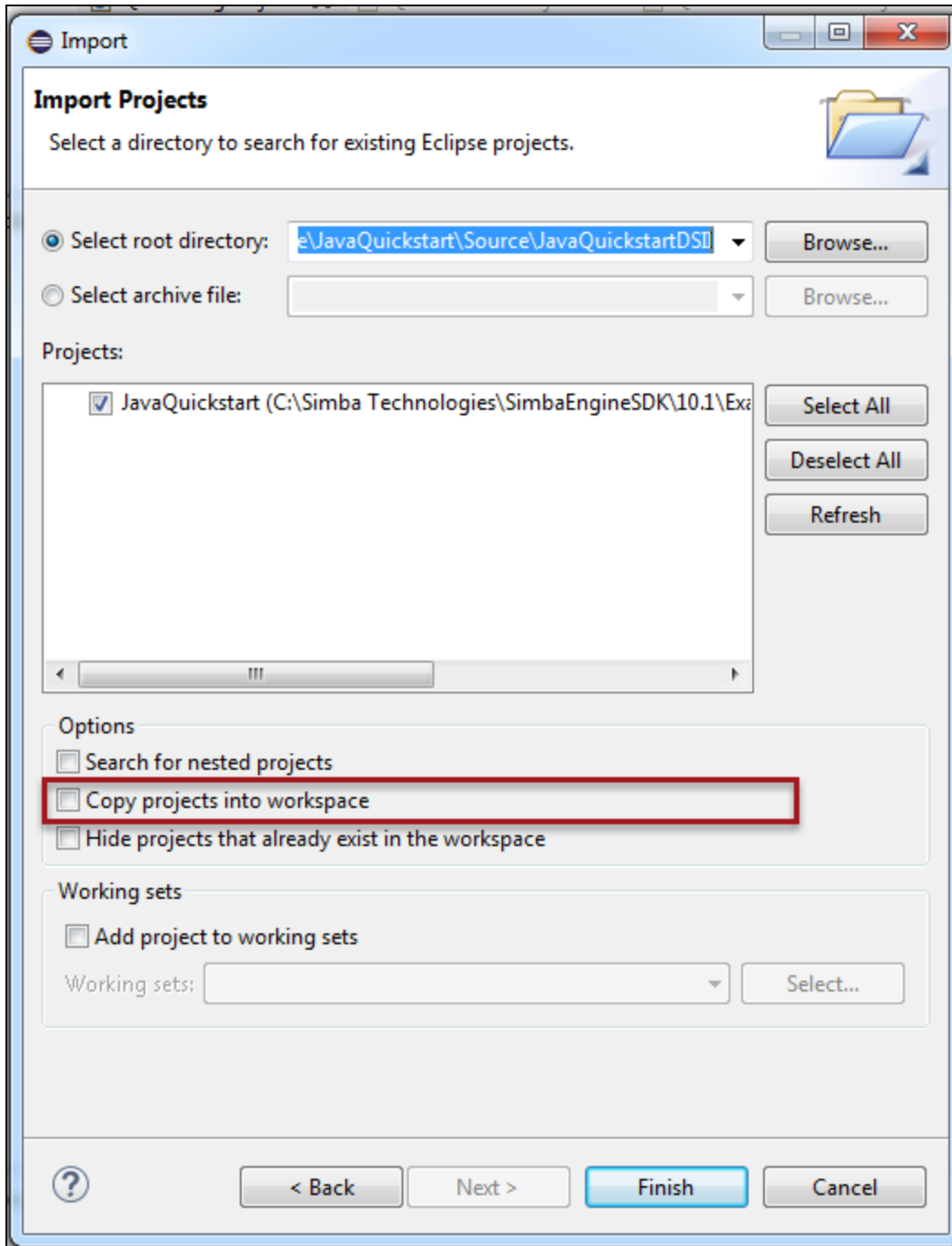
Important:
Important:

Uncheck the box **Copy projects into workspace**.

If this box is checked, the project is copied and the build will not succeed due to relative paths in the build files. To put the project in a different location, copy the entire project structure.

Note:

The Product includes a sample connector called **JavaQuickJSON**, which shows you how to write a JDBC connector. This guide uses the **JavaUltraLight** connector, which shows you how to write an ODBC connector using Java.



3. Update the classpath.
 - a. Select the **JavaUltraLight** project, then select **Project > Properties > Java Build Path > Libraries**.
 - b. Select **SIMBAENGINE_DIR > Edit > Variable > New**.
 - c. Create a new classpath variable called **SIMBAENGINE_DIR** with the value `[INSTALL_DIR]\DataAccessComponents`, for example: `C:\Simba Technologies\SimbaEngineSDK\10.4\DataAccessComponents`.

- d. A message is displayed, saying that the classpath variables have changed and that a full rebuild is recommended. Click **No**.
 - e. Click **OK** to close the properties windows.
4. Build the connector using the following steps:
 - a. Click **Run > External Tools > External Tools Configurations**.
 - b. In the External Tools Configurations window, double-click **Ant Build**. A setup page for the new Ant build configuration is displayed.
 - c. In the Name field, type **JavaUltraLight**.
 - d. On the Main tab, in the Buildfile section, click **Browse Workspace**.
 - e. In the Choose Location window, click **JavaUltraLightBuilderODBC.xml** and then click **OK**.
 - f. Click **Apply**, then click **Run**.

The JavaUltraLight connector is built using Ant. The resulting library is placed in `[INSTALL_DIR]\Examples\Source\JavaUltraLight\Lib\JavaUltraLight.jar`.

Related Links

[Java Server Configuration](#)

Update the Windows Registry

1. In Microsoft Visual Studio, click **File > Open > File** and navigate to `[INSTALL_DIR]\SimbaEngineSDK\10.4\Examples\Source\JavaUltraLight\Source`.
2. Open one of the following files:
 - For 32-bit Windows, open `SetupMyJavaUltraLightDSII-32on32.reg`.
 - Or, for a 32-bit ODBC connector on 64-bit Windows, open `SetupMyJavaUltraLightDSII-32on64.reg`.
 - For a 64-bit ODBC connector on 64-bit Windows, open `SetupMyJavaUltraLightDSII-64on64.reg`.
3. In the file, replace `[INSTALL_DIRECTORY]` with the Product installation directory. Use double backslashes in the path.

Example:

If the Product installation directory is `C:\Simba Technologies`, replace all instances of `[INSTALL_DIR]` with `C:\\Simba Technologies`.
4. Make sure that the `-Djava.class.path` path value for `JavaUltraLight.jar` points to the correct location.
5. Beside the line that starts with `"Driver"=` verify that the path to the connector `dll` file is correct.
6. Click **Save** and then close the file.
7. Update the Windows registry by double-clicking the registry file that you just modified.

A message is displayed that says that the keys and values have been successfully added to the registry.

Enable Debugging

To enable debugging for the JavaUltraLight sample connector:

1. In the Registry file that you edited above, edit the value data for the JNIconfig value so that the value for `-Djava.class.path` is correct.
2. Save, close, and then double-click the Registry file that you modified in step 1.

A message appears indicating that the keys and values have been successfully added to the Registry.

Examine the Windows Registry

Note:

You cannot use the ODBC Data Source Administrator to configure this data source because a configuration dialog has not been provided for the JavaUltraLight connector. If you try to use the ODBC Data Source Administrator, you will see the following error message:

"The setup routines for the JavaUltraLightDSIIDriver ODBC connector could not be found. Please reinstall the driver."

The Product installer automatically adds or updates the following registry keys that define Data Source Names (DSNs) and connector locations:

- **ODBC Data Sources** - lists each DSN/connector pair
- **JavaUltraLightDSII** - defines the Data Source Name (DSN). The ODBC Driver Manager uses this key to connect the connector to the database.
- **ODBC Drivers** - lists the connectors that are installed
- **JavaUltraLightDSIIDriver** - defines the connector and its setup location. The ODBC Driver Manager uses this key to connect to and configure the connector.

Note:

The installer for your custom connector must create similar registry keys.

To view the registry keys for the JavaUltraLight connector:

1. From a command line, run `regedit.exe`.
2. In the registry editor, navigate to one of the following root directories:
 - **HKEY_LOCAL_MACHINE\SOFTWARE\ODBC** for 64-bit connectors on 64-bit machines and 32-bit connectors on 32-bit machines.
 - Or, **HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432NODE\ODBC** for 32-bit connectors on 64-bit machines.
3. To view the registry keys that are related to ODBC connectors, look in the **ODBCINST.INI** subkey.

Related Links

[Bitness and the Windows Registry](#)

Connect to the Data Store

To connect to the data store and test the JavaUltraLight connector, any ODBC application can be used. This section shows how to use the ODBCTest tool, which is included in the Microsoft Data Access (MDAC) 2.8 Software Development Kit (SDK):

<http://www.microsoft.com/downloads/details.aspx?FamilyID=5067faf8-0db4-429a-b502-de4329c8c850&displaylang=en>

Note:

Before testing the data source, make sure that the PATH environment variable includes the path to the jvm.dll, for example C:\Program Files\Java\jdk1.7.0_09\jre\bin\server\.

To connect to the data store using the JavaUltraLight connector:

1. Navigate to the folder containing the ODBC Test application, by default:

C:\Program Files (x86)\Microsoft Data Access SDK 2.8\Tools

2. Navigate to the folder that corresponds to your connector's architecture: **amd64**, **ia64** or **x86**.

Example:

If you built the 32-bit version of your connector on a 64-bit machine, select the **x86** version.

3. Click one:

- **odbcte32.exe** to launch the ANSI version
- Or, **odbct32w.exe** to launch the Unicode version.



Important: Important:

It is important to run the correct version of the ODBC Test tool for ANSI or Unicode and 32-bit or 64-bit.

4. In the ODBC Test tool, click **Conn > Full Connect**.
The Full Connect window opens.
5. In the Full Connect dialog, select **JavaUltraLightDSII** from the list of data sources, and then click **OK**.
6. In the ODBC Test window, enter `SELECT * FROM ULResultSet`.
7. Click  and  to output a simple result set. The results are displayed in the window.

You have successfully used the JavaUltraLight connector to connect to the sample data store and retrieve data.

Related Links

[Bitness and the Windows Registry](#)

[Testing Your Simba SDK ODBC Connector with ODBC Test](#)

Set Up a Custom Project

Once the JavaUltraLight has been built and tested, you can create a new project for your custom ODBC connector.



Important:

It is very important that you create your own project directory. You might be tempted to simply modify the sample project files, but we strongly recommend that you create your own project directory. If you simply modify the sample project files:

- All your changes will be lost when you install a new version of the SDK.
- You will lose your frame of reference for debugging.
There may be times, for debugging purposes, that you will need to see if the same error occurs using the sample connectors. If you have modified the sample connectors, this won't be possible.

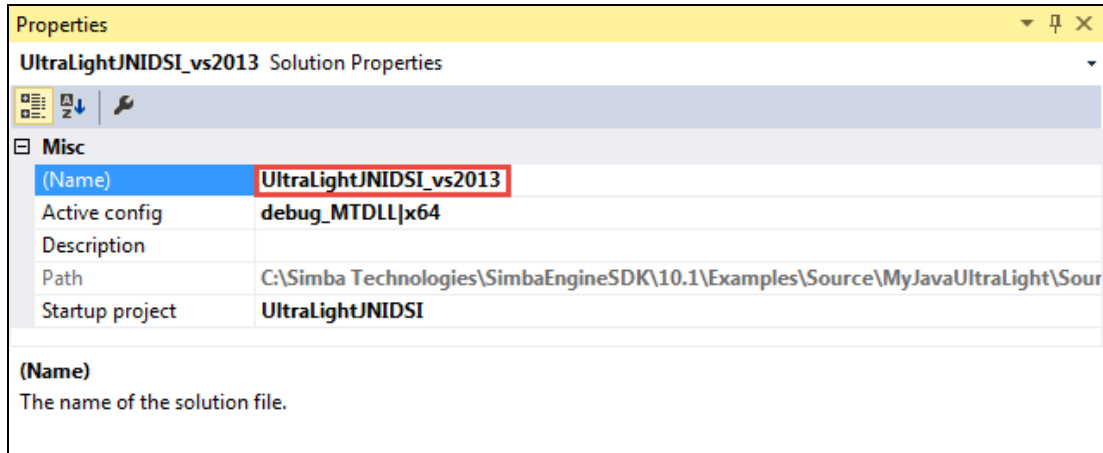
To set up a custom project:

1. In Windows Explorer, copy the following directory and paste it to the same location:

`[INSTALL_DIR]\SimbaEngineSDK\10.4\Examples\Source\JavaUltraLight`

where `[INSTALL_DIR]` is the Product installation directory. This will create a new directory called `JavaUltraLight - Copy`.

2. Rename the directory to the name of your custom ODBC connector. This name will be referred to as `[PROJECT]` for the rest of these steps.
3. In the `[PROJECT] > Source` directory, rename the `JavaUltraLightJNIDSI` directory to `[PROJECT]JNIDSI`.
4. In the `[PROJECT] > Source` directory, rename the `JavaUltraLightDSII` directory to `[PROJECT]DSII`.
5. Update the `UltraLightJNIDSI_VS201x.sln` file:
 - a. In your `[PROJECT]JNIDSI` directory, right-click the `UltraLightJNIDSI_VS201x.sln` file.
 - b. Select **Open With > Microsoft Visual Studio Version Selector**.
 - c. In the Microsoft Visual Studio menu, click **View > Solution Explorer**.
 - d. Using the Solution Explorer, in the **Properties** window, rename the `UltraLightJNIDSI_VS201x` solution to `[PROJECT]JNIDSI_VS201x`.



- e. Rename the `UltraLightJNIDSi` project to `[PROJECT]JNIDSi`.
 - f. Change the branding in `Simba::JNIDSi::SetConfigurationBranding()` in `main_windows.cpp`.
 - g. Click **File > Save All**.
6. Update the `JavaUltraLightBuilder.xml` file:
- a. In Windows Explorer, open your `[PROJECT]DSII` directory and then rename the `JavaUltraLightBuilder.xml` file to `[PROJECT]Builder.xml`. This is the Apache Ant builder file (.xml) for your new ODBC connector.
 - b. Using a text editor, open `[PROJECT]Builder.xml` and replace every instance of “JavaUltraLight” and “JavaUL” in the source code with the name of your new ODBC connector.
 - c. Update the copyright information for the “doc” target.
 - d. Save and close the file.
7. Open the `.project` file in a text editor and replace the `JavaUltraLight` within the `<name>` tags with your project name. Save and close the file.

Build the C++ Component

1. In Microsoft Visual Studio, click **Build > Build Solution** or press **F7** to build the connector.

When you build your new project, “TODO” messages appear in the Output window along with the build information. If the Output window is not displayed automatically, you can open it by selecting **Debug > Windows > Output**.

TODO #1: Update full Java connector name. (UltraLightJNIDSi.cpp)

TODO #2: Find or create the Java Virtual Machine. (UltraLightJNIDSi.cpp)

Build the Java component

You can use ANT or tools that support ANT, such as the Eclipse IDE, to build the Java component. As an example, these instructions describe how to build `JavaUltraLight` using the Eclipse IDE.

1. In Eclipse, select **File > Import**.
2. In the Import window, click **General > Existing Projects into Workspace** and then click **Next**.
3. Choose **Select root directory**, and then click **Browse** to navigate to `[INSTALL_DIR]\SimbaEngineSDK\10.4\Examples\Source\<YourProjectName>\Source\<YourProjectName>DSII`. Then click **Finish**.
4. To see your new Java project in the Project Explorer window, click **Window > Show View > Project Explorer**.
5. Click **Run > External Tools > External Tools Configurations**.
6. In the **External Tools Configurations** window, double-click **Ant Build**.

A setup page for the new Ant build configuration is displayed.

7. In the Name field, type your project name.
8. On the Main tab, in the Buildfile section, click **Browse Workspace**.
9. In the Choose Location window, click `[Project]Builder.xml` and then click **OK**.
10. Click **Apply**.
11. Still in the Ant build configuration, switch to the Environment tab. Add an environment variable called **SIMBAENGINE_DIR** with the value `[INSTALL_DIR]\SimbaEngineSDK\10.4\DataAccessComponents`. Click **Apply**.
12. Click **Run**. This will build the connector using Apache Ant.

Search your Java workspace for 'TODO' to find the following comments that mark locations where changes to your connector code need to be made:

TODO #1: Set the connector properties.	(ULDriver.java)
TODO #2: Set the connector-wide logging details.	(ULDriver.java)
TODO #3: Set the connection-wide logging details	(ULConnection.java)
TODO #4: Check Connection Settings.	ULConnection.java)
TODO #5: Establish A Connection.	(ULConnection.java)
TODO #6: Update configuration file.	(ULConnection.java)
TODO #7: Create and return your Metadata Sources.	(ULDataEngine.java)
TODO #8: Open A Table.	(ULDataEngine.java)
TODO #9: Assign a unique component ID.	(ULDriver.java)
TODO #10: Update Messages properties file.	(ULDriver.java)
TODO #11: Create an ExceptionBuilder.	(ULDriver.java)
TODO #12: Register the UltraLightmessages.	(ULDriver.java)

Over the next four days, you will visit each "TODO" and modify the source code.

Update the Registry for a Custom Connector

You must update the Windows registry to add the information for your new custom Java ODBC connector before you can test it.

Note:

You cannot use the ODBC Data Source Administrator to configure this data source.

To update the Windows Registry:

1. In Microsoft Visual Studio, click **File > Open > File** and navigate to `[INSTALL_DIRECTORY]\SimbaEngineSDK\10.4\Examples\Source\JavaUltraLight\Source`.
2. Open one of the following files:
 - For 32-bit Windows, open `SetupMyJavaUltraLightDSII-32on32.reg`.
 - Or, for a 32-bit ODBC connector on 64-bit Windows, open `SetupMyJavaUltraLightDSII-32on64.reg`.
 - For a 64-bit ODBC connector on 64-bit Windows, open `SetupMyJavaUltraLightDSII-64on64.reg`.
3. In the file, replace `[INSTALL_DIRECTORY]` with the Product installation directory. Use double backslashes in the path.

Example:

If the Product installation directory is `C:\Simba Technologies`, replace all instances of `[INSTALL_DIRECTORY]` with `C:\\Simba Technologies`.

4. To enable debugging, update the value data for the `JNIConfig` value so that the value for `djava.class.path` is correct. For more information, see [Enable Debugging](#).
5. Next, update the ODBC Data Sources section to add your new data source. Under the **[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\ODBC Data Sources]** section, change `"MyJavaUltraLightDSII"="MyJavaUltraLightDSIIDriver"` to the name of your new data source and new connector. For example, `"[PROJECT]DSII"="[PROJECT]DSIIDriver"`
6. Then, modify the data source definition for that data source. Change the line that says **[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\MyJavaUltraLightDSII]** so that it contains your new data source name. For example, **[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\[PROJECT]DSII]**.
7. Beside the line that starts with "Driver", update the path to the `.dll` file.
8. Update the ODBC Drivers section to add your new connector. Under the **[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\ODBC Drivers]** section, change `"MyJavaUltraLightDSIIDriver"="Installed"` to match the name of your new connector. For example, `"[PROJECT]DSIIDriver"="Installed"`
9. Modify the connector definition for that connector. Change the line that says **[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\MyJavaUltraLightDSIIDriver]** so that it contains your new connector name. For example, **[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\[PROJECT]DSIIDriver]**.
10. Beside the line that starts with "Driver", update the path to the `.dll` file.

11. Click **Edit > Find and Replace > Quick Replace**. Then, replace “JavaUltraLight” in the whole file with the name of your new connector.
12. Click **Save** and then close the file.
13. In the Registry Editor (`regedit.exe`), click **File > Import**, navigate to the registry file that you just modified and then click **Open**.

A message is displayed that says that the keys and values have been successfully added to the registry.

Enable Debugging

During the development of your connector, it may be necessary for you to trace through your connector during execution to locate problems. Most Java applications will either have a shell script/batch file to launch the application or have a configuration file where JVM options can be added

To enable debugging for the JavaUltraLight sample connector:

Add the following JVM options to enable debugging with the Eclipse IDE:

- `-Xdebug`
- `-Xrunjdwp:transport=dt_socket,address=localhost:8000,suspend=n,server=y`

Add these to the registry key `HKEY_LOCAL_MACHINE\SOFTWARE\Simba\JavaUltraLight\Driver\JNIconfig`

Each value needs to be separated by a pipe “|” character. For example:

```
-Djava.class.path=C:\Simba  
Technologies\SimbaEngineSDK\10.4\Examples\Source\JavaUltraLight\Lib\JavaUltraLight.jar|-  
Xdebug|-Xrunjdwp:transport=dt_socket,address=localhost:8000,suspend=n,server=y
```

Once options are added to your application, launch your application. You can now attach the Eclipse debugger to the running application and debug your connector.

If you need to debug the initialization of your connector, set the `suspend` parameter to `y`. A good breakpoint to start with is inside the `ULDriver` constructor. Launch your JDBC-enabled application. The JVM will suspend execution until a debugger has been attached.

See the Eclipse documentation for instructions on debugging Remote Java Applications.

Test Your Custom Connector

To test your custom connector, follow the instructions in [Connect to the Data Store](#), connecting to your custom connector instead of the JavaUltraLight connector.

Summary of Day One

You have successfully completed the following tasks:

- Built and tested the JavaUltraLight sample connector.
This verifies that your installation and development environment are properly configured.
- Created, built, and tested a custom connector project by copying the JavaUltraLight connector.
You can use this project as a framework to create your custom Java ODBC connector.

Day Two

Day Two instructions explain how to customize your ODBC connector, enable logging, and establish a connection to your data store.

Finding the TODO messages

When the custom project is built, TODO messages display in the Output window.

To rebuild the whole solution, select **Build > Rebuild Solution**. If the Output window is not open, select **Debug > Windows > Output**.

Double click a TODO message to jump to the relevant section of code.

Find or create the Java Virtual Machine

TODO #2 Find or create the Java Virtual Machine. (UltraLightJNIDSi.cpp)

For the purposes of prototyping, this TODO is purely informational. There is nothing to change here right now, although you may want to add processing at this point for a commercial connector.

The `JvmFactory()` implementation in `UltraLightJNIDSi.cpp` in the `UltraLightJNIDSi` project is the first hook that is called from Simba's JNIDSi layer to find or create an instance of the Java VM during initialization of the bridge. This method is called soon after the Driver Manager calls `LoadLibrary()` on your ODBC connector. After that, the C++ to Java proxies are initialized and an instance of your DSI implementation is created when the name of the Java Connector is retrieved from `GetFullJavaDriverName()`. The name returned by this method must match the fully-qualified name of the Java connector class in `ULDriver.java`, for example `com.simba.javaultralight.core.ULDriver`.

Set the Connector Name

TODO #1 Set the connector name. (ULDriver.java)

Set the constant `DRIVER_NAME` to the name of your connector. Typically, this is the same name you used to replace "JavaUltraLight" in [Set Up a Custom Project](#).

Set the Connector Properties

TODO #2 Set the connector properties.
(QSDriver.java)

TODO #3: Set the connection properties.

1. In Eclipse, in your project, go to the TODO #2 message in the `ULDriver.java` file.
2. Look at `setProperty()` where you will set up the general properties for your connector.
3. Change the `DSI_DRIVER_DRIVER_NAME` to the name of your new connector.

4. Go to the TODO #3 message in the `ULConnection.java` file. Look at `setProperty()` where you will set up the general properties for your connection.
5. Adjust the connection properties as required by your connector.

Set the Logging Details

TODO #4	Set the connector-wide logging details. (<code>ULDriver.java</code>)
TODO #5	Set the connection-wide logging details. (<code>ULConnection.java</code>)

1. Go to the TODO #4 message.
2. Change the connector log's file name.
3. Go to the TODO #5 message.
4. Change the connection log's file name.

For more information about how to enable logging, see [Developing Connectors for SQL-capable Data Stores](#).

Note:

By default, the Product JavaUltraLight Connector maintains two kinds of log files: one for all connector-based calls and one for each connection created. Update these TODO's if you do not require such fine granularity in logging.

Check the Connection Settings

TODO #6	Check Connection Settings. (<code>QSConnection.java</code>)
---------	---

When the Simba ODBC layer is given a connection string from an ODBC-enabled application, the Simba ODBC layer parses the connection string into key-value pairs. Then, the entries in the connection string and the DSN are sent to `updateConnectionSettings()` function for validation.

When a connection occurs, a connection URL is passed to the JDBC connector. As an example, take the connection string `"jdbc:simba://User=user;Password=pass"`. This connection string is broken down into key-value pairs and stored in a `ConnSettingRequestMap`, in this case that map would contain two entries: `{"User", "user"}` and `{"Password", "pass"}`. This map is then passed down to the DSII.

1. Go to the TODO #6 message to jump to the relevant section of code.
2. The `updateConnectionSettings()` function should validate that the key-value pairs within the `requestMap` are sufficient to create a connection. Use the `verifyRequiredSetting()` or `verifyOptionalSetting()` utility functions to do this.
3. If any of the values received are invalid, you should throw an exception.
4. If there are no further entries required, simply leave the `responseMap` empty.

Establish a Connection

TODO #7 Establish A Connection. (QSConnection.java)

Once `ULConnection`'s `updateConnectionSettings()` returns a `responseMap` without any required settings (if there are only optional settings, a connection can still occur), the Simba ODBC layer will call `QSConnection`'s `connect()` passing in all the connection settings received from the application. This is where you should authenticate the user against your data store using the information provided within the `requestMap` parameter.

Should authentication fail, you should throw a `BadAuthException`. You can also use the utility functions supplied: `getRequiredSetting()` and `getOptionalSetting()`.

Summary of Day Two

You have successfully authenticated the user against your data store and established a connection.

Update the Configuration File

TODO #6 Update configuration file. (QSConnection.java)

The purpose of this configuration file is to enable server-specific behavior during runtime. When the connector is configured to be a server, then the connection settings in `QSConnection`'s `updateConnectionSettings()` and `connect()` need to be augmented. This file only needs to exist if the connector is set to be a server.

For the purposes of prototyping, this TODO is mostly informational. If you know that you will use the client/server configuration for accessing your data source, rename the configuration file to use an appropriate name. Otherwise, there is nothing to change here right now, though you may wish to revisit this later.

For more information about setting up a Java connector as a server, see [Java Server Configuration](#).

You have now authenticated the user against your data store.

Day Three

The Day Three instructions explain how to return the data used to pass catalog information back to the ODBC-enabled application.

Create and Return Metadata Sources

Your custom ODBC connector uses metadata sources, provided by the Product, to handle SQL catalog functions.

Overview of SQL Catalog Functions

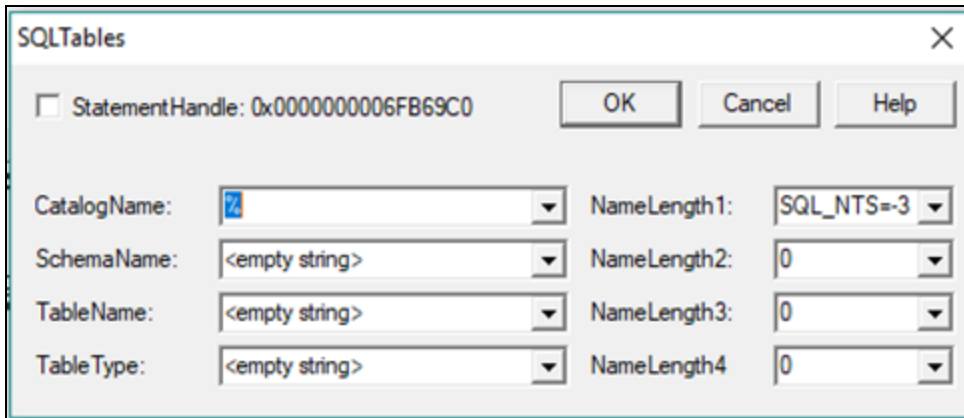
ODBC applications need to understand the structure of a data store in order to execute SQL queries against it. This information is provided using catalog functions. For example, an application might request a result set containing information about all the tables in the data store, or all the columns in a particular table. Each catalog function returns data as a result set.

Most ODBC-enabled applications require a connector to implement the following catalog functions. You may wish to implement additional catalog functions in your custom connector.

Catalog Function	Description
SQLGetTypeInfo	Returns information about data types supported by the data source.
SQLTables (CATALOG_ONLY)	If CatalogName is SQL_ALL_CATALOGS and SchemaName and TableName are empty strings, the result set contains a list of valid catalogs for the data source. (All columns except the TABLE_CAT column contain NULLs.)
SQLTables (SCHEMA_ONLY)	If SchemaName is SQL_ALL_SCHEMAS and CatalogName and TableName are empty strings, the result set contains a list of valid schemas for the data source. (All columns except the TABLE_SCHEM column contain NULLs.)
SQLTables (TABLE_TYPE_ONLY)	If TableType is SQL_ALL_TABLE_TYPES and CatalogName, SchemaName, and TableName are empty strings, the result set contains a list of valid table types for the data source. (All columns except the TABLE_TYPE column contain NULLs.)
SQLTables	Returns the list of table, catalog, or schema names, and table types, stored in a specific data source.
SQLColumns	Returns a list of columns in one or more tables.

Example: Using Catalog Functions with the JavaUltraLightconnector

1. In the ODBC Test application, connect to the JavaUltraLight connector.
2. To send the SQLTables (CATALOG_ONLY) catalog function, select **Catalog > SQLTables**.
3. Enter SQL_ALL_CATALOGS for the **CatalogName**, then select the correct value for **NameLength1**. For example:



4. Click **OK**.
5. Select  to retrieve the results.

The following list of valid catalogs for the JavaUltraLight data source are returned:

"TABLE_QUALIFIER", "TABLE_OWNER", "TABLE_NAME", "TABLE_TYPE", "REMARKS"

For more information on SQL catalog functions, see [https://msdn.microsoft.com/en-us/library/ms713520\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/ms713520(v=vs.85).aspx).

Implementing Metadata Sources to Handle Catalog Functions

SQL catalog functions are represented in the DSI by metadata sources: there is one metadata source for each of the catalog functions.

`ULDataEngine.MakeNewMetadataTable()` is responsible for creating the metadata sources. Metadata sources are used return the catalog metadata about your data store to the ODBC application for the ODBC catalog functions.

Double click the **TODO #8 Create and return your Metadata Sources** message to jump to the relevant section of code.

`ULDataEngine.makeNewMetadataTable()` creates sources that are used to return data to the application for each of the ODBC catalog functions. Each ODBC catalog function is mapped to a unique `MetadataSourceId`, which is then mapped to an underlying `IMetadataSource` that you implement and return. Each `IMetadataSource` instance is responsible for the following:

- Creating a data structure that holds the data relevant for your data store: `Constructor`
- Navigating the structure on a row-by-row basis: `moveToNextRow()`
- Retrieving data: `getMetadata()`. For more information, see [Data Retrieval](#).

Required Metadata Sources

All custom ODBC connectors must implement the following metadata sources, as they are required by ODBC applications:

Metadata Source	Description
DSI_TABLES_METADATA	List of all tables defined in the data source.
DSI_CATALOGONLY_METADATA	List of all catalogs defined in the data source, if catalogs are supported.
DSI_SCHEMAONLY_METADATA	List of all schemas defined in the data source. This source is constructed via the <code>ULMetadataHelper</code> and SQL Engine.
DSI_TABLETYPEONLY_METADATA	List of all table types (TABLE,VIEW,SYSTEM) defined within the data source.
DSI_COLUMNS_METADATA	List of all columns defined across all tables in the data source.
DSI_TYPE_INFO_METADATA	List of the supported types by the data source. This means the actual types that can be stored in the data source, not necessarily the types that can be returned by the connector. For instance, a conversion may result in a type being returned that is not stored in the data source.

Most catalog types are created using the metadata helper.

Handle DSI_TYPE_INFO_METADATA

The underlying ODBC catalog function `SQLGetTypeInfo` is handled as follows:

1. When called with `TYPE_INFO`, `ULDataEngine.makeNewMetadataTable()` returns an instance of `ULTypeInfoMetadataSource()`.
2. The `JavaUltraLight` connector example exposes support for the following types:

SQL_BIT

SQL_CHAR

SQL_DOUBLE

SQL_INTEGER	SQL_LONGVARBINARY	SQL_LONG_VARCHAR
SQL_NUMERIC	SQL_REAL	SQL_SMALLINT
SQL_TINYINT	SQL_TYPE_DATE	SQL_TYPE_TIME
SQL_TYPE_TIMESTAMP	SQL_VARBINARY	SQL_VARCHAR
SQL_WCHAR	SQL_WLONGVARCHAR	SQL_WVARCHAR

3. For your custom Java ODBC connector, you may need to change the types returned and the parameters for the types in `ULTypeInfoMetadataSource.initializeDataTypes()`.

Handle the other MetadataSources

The other ODBC catalog functions (including `SQLTables (CATALOG_ONLY)`, `SQLTables (TABLETYPE_ONLY)`, `SQLTables (SCHEMA_ONLY)`, `SQLTables` and `SQLColumns`) are handled automatically as follows.

When called with the corresponding metatable IDs:

When called with the corresponding metatable ID's, `ULDataEngine.makeNewMetadataSource()` returns a new instance of one of the following respective `DSIMetadataSource`-derived classes:

- **ULCatalogOnlyMetadataSource**: returns a list of all catalogs. The sample implementation returns one row of information with one column containing the name of a fake catalog. This demonstrates how to return a catalog name.
- **DSITableTypeOnlyMetadataSource**: (default implementation by Simba) returns metadata about all tables of a particular type (TABLE, SYSTEM TABLE, and VIEW) in the datasource. This class provides two constructors which allow for returning the default set of table types (listed above) or for specifying your own set of table types.
- **ULSchemaOnlyMetadataSource**: returns a list of all schemas. The sample implementation returns one row of information with one column containing the name of a fake schema. This demonstrates how to return a schema name.
- **ULTablesMetadataSource**: returns metadata about all of the tables in the data source. The sample hard codes and returns information for the hard coded person table to demonstrate how to return table metadata.
- **ULColumnsMetadataSource**: returns metadata for the columns in the data source. The sample hard codes and returns information for the three columns in the person table consisting of the name column, an integer column, and a numeric column.

When called with any other `MetadataSourceID`

When called with any other `MetadataSourceID`, which doesn't correspond to these tables, `ULDataEngine.makeNewMetadataSource()` returns a new instance of `DSIEmptyMetadataSource` to indicate that no metadata is available for the specified table ID.

You can now retrieve type metadata from within your data store.

Summary of Day Three

Your custom ODBC connector can now return type metadata. You can use a ODBC-enabled application to connect to your connector and retrieve type metadata from within your data store

Related Links

For more details on *Fetching Metadata for Catalog Functions*, see [Simba SDK Driver Development](#).

Day Four

Day Four instructions explain how to enable data retrieval from within the connector.

Enable Data Retrieval

`ULDataEngine.openTable()` is the entry point where the Simba SQL Engine requests that tables involved in the query be opened. It is called during the preparation of a SQL statement.

TODO #9: Open A Table. (`ULDataEngine.java`)

Note:

The SQL Engine component of the Product allows applications to execute SQL commands on data stores that are not SQL-capable.

`ULTable` is an implementation of `DSIExtResultSet`, an abstract class that provides for basic forward-only result set traversal. The main role of `ULTable` is to translate the stored data from your native data format into SQL Data types.

The JavaUltraLight connector is implemented to support Tabbed Unicode Files. It translates the text from UTF16-LE strings into the SQL Data types defined for each column within the configuration dialog.

The following sections explain how to implement data retrieval in your custom Java ODBC connector.

Modify the OpenTable Method

The `ULDataEngine.openTable()` method is called during the preparation of a SQL statement. Modify this method to check that the supplied catalog, schema and table names are valid and correspond to a table defined in your data store. If the inputs are not valid, return `null` to indicate that the table does not exist. If the inputs are valid, a new instance of `ULTable` is returned.

Modify ULTable

This section tells you how to modify `ULTable` so that it can work with your data store.

Return the catalog, schema and table names for your table

- `ULTable.QSTable()`: Modify the constructor to take in the catalog, schema and table names and save them in member variables.
- `ULTable.getCatalogName()`: Returns `UltraLight.CATALOG` (because it has only a single catalog).
- `ULTable.getSchemaName()`: Returns `null` (because it does not support schemas).
- `ULTable.getTableName()`: Returns `m_tableName`.

Return the Columns Defined for Your Table

Modify `ULTable.initializeColumns()` so that, for each column defined in the table, you define the `ColumnMetadata` in terms of SQL types.

Example pseudo code for a custom `ULTable.InitializeColumns()` method

```

Get all the column information from your data store for the table
For Each Defined Column { // Change the parameter of this method
to the SQL Type that // maps to your data store type. TypeMetadata
typeMetadata = TypeMetadata.createTypeMetadata(Types.VARCHAR); //
Depending on SQL type, set different properties: if (character
type) { typeMetadata.setIntervalPrecision(m_settings.m_maxColumn-
nSize); } else if (exact numeric type) { typeMetadata.setScale(
scale ); } ColumnMetadata columnMetadata = new ColumnMetadata
(typeMetadata); columnMetadata.setCatalogName(m_catalogName);
columnMetadata.setSchemaName(m_schemaName); colum-
nMetadata.setTableName(m_tableName); columnMetadata.setName
("column name"); columnMetadata.setLabel("localized column name");
columnMetadata.setNullable(Nullable.NULLABLE); if ( character type
) { columnMetadata.setColumnLength(m_settings.m_maxColumnSize); }
m_columns.add(columnMetadata); }

```

Implement Navigation

- The following methods are responsible for navigating a data structure containing information about one table in your data store, then retrieving data from that table. Modify these methods for your data store.
- `ULTable.moveToNextRow()`
- `ULTable.getData()`

In your custom implementation:

- We recommend that you implement a class that provides a streaming interface for the data in the table within your data store.
- Provide the ability to navigate forward from one table row to the next.
- Provide the ability to navigate across columns within the row
- Provide the ability to read the data associated with the current row and column combination.

In the JavaUltraLight Connector, `ULTable` uses a `TabbedUnicodeFileReader`, which provides an interface to navigate between lines within a Unicode text file. This class preprocesses each row in the file to determine the starting file offset of each column in the row. Its `getData` method takes a column index and uses it to calculate the exact position in the file where the column's data resides. The method

repositions the file and retrieves the data as if from a byte-buffer. For more information, see [Data Retrieval](#).

Close the Connection

`ULTable.closeCursor()` is a callback method called from Simba SQL Engine to indicate that data retrieval has completed and that you may now do any tasks related to closing the connection to your data store.

Summary of Day Four

You can now execute queries and retrieve data from your data store. You can use any ODBC-enabled application to execute queries and see the results returned from your data store.

Day Five

Day Five explains how to rebrand and productize your custom Java ODBC connector.

Productize Your Custom Connector

Day Five instructions explain how to productize and rebrand your custom Java ODBC connector.

Assign a Component ID

For the purpose of prototyping, this TODO is purely informational. The default component ID is usually sufficient for most connectors. The component ID is used to identify the component from which an error has been generated so that the correct component name can be included in the error message.

TODO #9 Assign a unique component ID. (ULDriver.java)

Update the Error Messages

All the error messages used within your DSI implementation are stored in a file called `messages.properties`. Update each exception thrown within your DSI implementation and change the parameters to match as well.

TODO #10 Update Messages properties file. (ULDriver.java)

Extend the ExceptionBuilder Class

For the purpose of prototyping, this TODO is purely informational. The `ExceptionBuilder` that is created by default should be sufficient for the vast majority of connectors. However, if you wish to extend the `ExceptionBuilder` class, this is the location where you would instantiate your class.

TODO #11 Create an ExceptionBuilder. (ULDriver.java)

Configure the Error Messages

For the purpose of prototyping, this TODO is mostly informational. By default, the connector's `messages.properties` file resides in the same package as the `ULDriver` class. You can modify the code to look in a different package location for the messages file or to customize the name of the file.

TODO #12 Register the UltraLight messages. (ULDriver.java)

All error messages returned by the connector begin with the component name. Simply change the "UltraLight" to a name relating to your connector. This rebrands your converted Product JavaUltraLight Connector for your organization.

First rename all packages, files, and classes by changing all instances of the following items:

- The word "UltraLight" to the name you chose as a part of TODO #9.
- The letters "UL" to a two letter abbreviation of your choice.

Now you can update the full Java connector name for this TODO to match the path and name of your connector. Replace the package name “javaultralight” with your re-branded name as well as the “UL” for the Connector name with your two letter abbreviation.

TODO #2 Update full Java connector name. (UltraLightJNIDS!.cpp)

Create a Connector Configuration Dialog

The connector configuration dialog is presented to the user when they use the ODBC Data Source Administrator to create a new ODBC DSN or configure an existing one.

The C++ Product UltraLight Connector project contains an example ODBC configuration dialog that you can look at, as an example. You can find the source under the `Setup` folder within the Product UltraLight Connector project.

To see the connector configuration dialog that you created, run the ODBC Data Source Administrator. To do this, open the Control Panel, select Administrative Tools, and then select Data Sources (ODBC). If your Control Panel is set to view by category, then Administrative Tools is located under System and Security.

Note:

If you are using 64-bit Windows with 32-bit applications, you must use the 32-bit ODBC Data Source Administrator. You cannot access the 32-bit ODBC Data Source Administrator from the start menu or control panel in 64-bit Windows.

Only the 64-bit ODBC Data Source Administrator is accessible from the start menu or control panel. On 64-bit Windows, to launch the 32-bit ODBC Data Source Administrator you must run

`C:\WINDOWS\SysWOW64\odbcad32.exe.`

For more information, see [32-bit vs 64-bit ODBC Data Source Administrator](#).

You are now done with all of the TODO's in the project. You have created your own, custom ODBC connector using Product by modifying and customizing the JavaUltraLight sample connector. Now, you have a read-only connector that connects to your data store.

Reference

This section contains more implementation details for creating your custom ODBC connector in Java.

Data Retrieval

In the Data Store Interface (DSI), the following two methods retrieve data from your data store:

- Each `IMetadataSource` implementation of `getMetadata()`
- `QSTable::getData()`

Both methods will provide a way to uniquely identify a column within the current row. For `IMetadataSource`, the Simba SQL Engine will pass in a unique column tag (see `MetadataSourceColumnTag`). For `QSTable`, the Simba SQL Engine will pass in the column index.

In addition, both methods accept the parameters described below.

data parameter

The `DataWrapper` into which you must copy your cell's value. This class is a wrapper around an Object managed by the Simba SQL Engine. You simply call its `set<Data Type>()` and `get<Data Type>()` methods to store and access the data according to the `java.sql.Type`. The data you set must be represented as the Object or primitive data type that is accepted by the set methods for that `java.sql.Type`. If your data is not stored as the appropriate type, you will need to write code to convert from your native format.

The type of this parameter is governed by the metadata for the column that is returned by the class. Thus, if you create the `TypeMetadata` of column 1 in `QSTable::initializeColumns()` as `Types.INTEGER`, then when `QSTable::getData()` is called for column 1, you will be passed a `DataWrapper` that wraps a Long data type. For `IMetadataSource`, the type is associated with the column tag (see `MetadataSourceColumnTag`).

Note:

It is important to note that while Java does not natively support unsigned integer-types (i.e. the types represented by `TINYINT`, `SMALLINT`, `INTEGER`), the Simba SQL Engine allows for unsigned data types to be retrieved through the C++ to Java bridge. By up-casting to a larger signed type for each of the integer-types, unsigned values can be stored until they are retrieved and converted to the correct unsigned SQL Type at the C++ end of the bridge. By default, the `TypeMetadata` for the column is set to treat the integer-types as signed. To enable unsigned data, you will need to call `TypeMetadata::setSigned(false)` when creating the `ColumnMetadata` for the column in `QSTable::initializeColumns()`.

offset parameter

Character, wide character, and binary data types can be retrieved in parts. This value specifies where, in the current column, the value should be copied from. The value is usually 0.

maxSize parameter

The maximum size (in bytes) that can be copied into the `in_data` parameter. For character or binary data, copying data that is greater than this size can result in a data truncation warning or a heap-violation.

Java Server Configuration

Your custom ODBC connector can be recompiled as a server and deployed in a client-server configuration. The connection settings for the connector are normally retrieved directly from the ODBC DSN. However, when the connector is a server, the settings cannot be retrieved directly because the DSN refers to the client instead of a specific connector. Also, to enforce security, clients do not have control over server-specific settings.

The JavaUltraLight sample connector uses a configuration file to store connection information for client-server configurations. When the connector's `server` flag is set, the key/value pairs in the configuration file provide extra information about the connection settings that are passed in during a connection.

To set up the JavaUltraLight server on Windows:

1. Build the JavaUltraLightJNIDSI using a server configuration (i.e. `Debug_Server` or `Release_Server`). This builds the server executable.
2. Build the JavaUltraLight DSII using the ANT build script, as you would for a standalone connector.
3. Edit the file `qsconfig.properties` so that the server flag is set to true and the `[INSTALL_DIR]` is replaced to point to the correct DBF location. It should include the following string values:

`SERVER=true`

`DBF=[INSTALL_DIR]\Examples\Databases\JavaUltraLight`

Note:

Double slashes are needed in the path since the values are read in as Java Strings and the backslashes need to be escaped.

4. Place the `qsconfig.properties` in the same directory as the `JavaJavaUltraLight.jar` file.
5. Ensure that the JNIConfig is configured correctly, as you would for a standalone connector. For more information, see [Simba Client/Server Developer Guide](#).

To set up the JavaUltraLightserver on non-Windows:

1. Build JavaUltraLight using the debug (or release) server configuration:

```
./mk.sh BUILDSEVER=1 MODE=debug
```

2. Build the JavaUltraLightDSII using the ANT build script, as you would for a standalone connector.
3. Edit the file `qsconfig.properties` so that the server flag is set to true and replace the DBF with the correct path. It should include the following string values:

`SERVER=true`

`DBF=[INSTALL_DIR]/Examples/Databases/JavaUltraLight`

4. Place the `qsconfig.properties` in the same directory as the `JavaUltraLight.jar` file.
5. Ensure that the JNIConfig is configured correctly, as you would for a standalone connector. For more information, see [Simba Client/Server Developer Guide](#).

Once you have configured the client and server, you should can connect to your data source.

32-bit vs 64-bit ODBC Data Source Administrator

On a 64-bit Windows machine, you can execute both 64-bit and 32-bit applications. Many applications are available in 32-bit versions only, and running 32-bit applications on 64-bit operating systems is common.

However, in a single running process, all of the code must be either 32-bit or 64-bit. A connector must have the same bitness as the application that loads it.



Important:

- 64-bit applications can only load 64-bit connectors and 32-bit applications can only load 32-bit connectors.
- In a single running process, all of the code must be either 64-bit or 32-bit.

Note:

- Microsoft Excel is available in 32-bit and 64-bit versions.

Using the Correct ODBC Data Source Administrator

When using 64-bit Windows, it is very important that you configure 32-bit connectors with the 32-bit ODBC data source administrator, and 64-bit connectors with the 64-bit ODBC Data Source Administrator. This can cause confusion, where what appears to be a perfectly configured ODBC DSN does not work because it is loading the wrong kind of connector.

On a 64-bit Windows machine, use the Control Panel to launch the 64-bit ODBC data source administrator.

To launch the 32-bit ODBC data source administrator, run the following:

```
C:\WINDOWS\SysWOW64\odbcad32.exe.
```

Tip:

Create a shortcut to the 32-bit ODBC Data Source Administrator on your Desktop or Start menu if you configure 32-bit data sources frequently.

Bitness and the Windows Registry

32-bit applications can run on 64-bit machines, but they cannot use 64-bit ODBC connectors. A 64-bit application must use a 64-bit ODBC connector, and a 32-bit application must use a 32-bit ODBC connector.

On machines running 64-bit Windows operating systems, system-wide information about 64-bit ODBC connectors is stored in **HKEY_LOCAL_MACHINE/SOFTWARE/ODBC**, and system-wide information about 32-bit ODBC connectors is stored in **HKEY_LOCAL_MACHINE/SOFTWARE/WOW6432NODE/ODBC**.

On machines running 32-bit Windows operating systems, system-wide information about ODBC connectors is stored in **HKEY_LOCAL_MACHINE/SOFTWARE/ODBC**. This is the same location as 64-bit applications running on 64-bit machines.

Note:

32-bit Windows operating systems cannot run 64-bit applications or connectors.

The ODBC.INI Key

The ODBC Data Source Administrator uses information in the following key to connect a connector to a database:

- **HKEY_LOCAL_MACHINE/SOFTWARE/ ODBC/ODBC.INI** for 64-bit applications on 64-bit machines, or 32-bit applications on 32-bit machines.
- **HKEY_LOCAL_MACHINE/SOFTWARE/WOW6432NODE/ODBC/ ODBC.INI** for 32-bit applications on 64-bit machines.

This key contains a key for each Data Source Name (DSN). For more information about this key, see [https://msdn.microsoft.com/en-us/library/ms715391\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/ms715391(v=vs.85).aspx).

The ODBC Data Sources key

The **ODBC.INI** key also contains a key named **ODBC Data Sources** that lists the data sources. The values for the data sources must match the name of each DSN key. For more information about this key, see [https://msdn.microsoft.com/en-us/library/ms709335\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/ms709335(v=vs.85).aspx).

For more information on Windows Registry entries for data sources, see <https://msdn.microsoft.com/en-us/library/ms712603%28v=vs.85%29.aspx>.

The ODBCINST.INI Key

The ODBC Data Source Administrator uses information in the following key to define each connector's name and setup location:

- **HKEY_LOCAL_MACHINE/SOFTWARE/ODBC/ODBCINST.INI** for 64-bit applications on 64-bit machines, or 32-bit applications on 32-bit machines
- **HKEY_LOCAL_MACHINE/SOFTWARE/WOW6432NODE/ODBC/ODBCINST.INI** for 32-bit applications on 64-bit machines

This key contains a connector specification key for each connector. For more information about connector specification keys, see [https://msdn.microsoft.com/en-us/library/ms715391\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/ms715391(v=vs.85).aspx).

The ODBC Connectors Key

The **ODBCINST.INI** key also contains a key named **ODBC Drivers** that lists the installed connectors.

For more information about this key, see <https://msdn.microsoft.com/en-us/library/ms714818%28v=vs.85%29.aspx>.

Contact Us

For more information or help using this product, please contact our Technical Support staff. We welcome your questions, comments, and feature requests.

**Note:**

To help us assist you, prior to contacting Technical Support please prepare a detailed summary of the Product version and development platform that you are using.

You can contact Technical Support via the Insightsoftware Support Community at <https://insightsoftware.com/>.

Third-Party Trademarks

Simba, the Simba logo, Simba SDK, and Simba Technologies are registered trademarks of Simba Technologies Inc. in Canada, United States and/or other countries. All other trademarks and/or servicemarks are the property of their respective owners.

Kerberos is a trademark of the Massachusetts Institute of Technology (MIT).

Linux is the registered trademark of Linus Torvalds in Canada, United States and/or other countries.

Mac and macOS are trademarks or registered trademarks of Apple, Inc. or its subsidiaries in Canada, United States and/or other countries.

Red Hat, Red Hat Enterprise Linux, and CentOS are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in Canada, United States and/or other countries.

SUSE is a trademark or registered trademark of SUSE LLC or its subsidiaries in Canada, United States and/or other countries.

Ubuntu is a trademark or registered trademark of Canonical Ltd. or its subsidiaries in Canada, United States and/or other countries.

All other trademarks are trademarks of their respective owners.