[vertical list of authors]

[cover art/text goes here]

Contents

Copyright

Second Edition (July 2005)

About this guide

For general information about the Derby documentation, such as a complete list of books, conventions, and further reading, see *Getting Started with Derby* .

Purpose of this document

This book describes how to use the Derby tools and utilities. The tools and utilities covered in this book include:

- ij
- the import and export utilities
- the database class loading utilities
- sysinfo
- dblook

Audience

This book is for:

- developers, who might use the tools when developing applications
- system administrators, who might use the tools to run backup scripts or to import large amounts of data
- end-users, who might use one of the tools to run ad-hoc queries against a database

How this guide is organized

This guide includes the following sections:

- What are the Derby tools and utilities?

  Overview of the tools and utilities, and Derby and JDBC basics for new or infrequent users.

- Using ij

  How to get started with ij, a JDBC and SQL scripting tool.

- ij properties reference

  Reference for ij properties.

- ij commands and errors reference

  Reference for ij commands and errors.

- Using the bulk import and export procedures

  Reference and how-to instructions for using bulk import and export.

- sysinfo

  Reference information on the utility that provides information about your Derby environment.

- dblook

  dblook is Derby's Data Definition Language (DDL) Generation Utility, which is more informally called a schema dump tool. It is a simple utility that dumps the DDL of a user-specified database to either a console or to a file. The generated DDL can then be used to recreate all or parts of a database, view a subset of a database's objects (for example, those which pertain to specific tables or schemas), or document a database's schema.

What are the Derby tools and utilities?

This section covers some basics about Java, Derby, and JDBC for new users or infrequent users (such as those who only use the Derby tools and are not developing applications).

For more complete information on these topics, see the *Derby Developer's Guide* .

Overview

Derby is a database management system (DBMS), accessed by applications through the JDBC API.

Included with the product are some standalone Java tools and utilities that make it easier to use and develop applications for Derby.

These tools and utilities include:

- `ij`

  `ij` is Derby's interactive JDBC scripting tool. It is a simple utility for running scripts against a Derby database. You can also use it interactively to run ad hoc queries. `ij` provides several commands for ease in accessing a variety of JDBC features.

  `ij` can be used in an embedded or a client/server environment.

- *The import and export utilities*

  These server-side utilities allow you to import data directly from files into tables and to export data from tables into files. (You can use the utilities in a client/server environment.)

- *Database class utilities*

  These utilities allow you to store application logic in a database and to boot an application using the stored classes.

- *sysinfo*

  `sysinfo` provides information about your version of Derby and your environment.

- *dblook*

  `dblook` is Derby's Data Definition Language (DDL) Generation Utility, also called a schema dump tool. It is a simple utility for the dumping the DDL of a user-specified database to either a console or to a file. The generated DDL can then be used for such things as recreating all or parts of a database, viewing a subset of a database's objects (for example, those which pertain to specific tables and schemas), or documenting a database's schema.

JVM and classpath for Derby tools

`ij`, `sysinfo`, and `dblook` are tools that can be used in an embedded or a client/server environment. The import and export utilities and database class utilities are database-side utilities, which means that they run in the same JVM as Derby, although the import and export utilities can also be used in a client/server environment.

Java 2 Platform, Standard Edition, Version 1.3

All Derby tools require Java 2 Platform, Standard Edition, Version 1.3 or later.

Classpath

Derby classpath requirements:

- To use `ij`, you must have *derbytools.jar* in your classpath. If you are using the embedded driver, you must also include *derby.jar*.
- To use the import and export utilities and the database class utilities, you must have *derby.jar* in your classpath.
- To use sysinfo, either *derby.jar* or *derbytools.jar* must be in your classpath.
- To use Derby tools from a client with the Derby Network Server, you must have *derbyclient.jar* and *derbytools.jar* in your classpath. See the *Derby Server and Administration Guide* for more information.

About Derby databases

A Derby database consists of platform-independent files stored in a directory that has the same name as the database.

JDBC basics

Most of the Derby tools are JDBC applications. A JDBC application is one that uses the classes in the *java.sql* package to interact with a DBMS.

When you work with JDBC applications, you need to know about several concepts.

JDBC drivers overview

Before a JDBC application interacts with a database, it must cause the JDBC driver to be loaded in the Java session. Derby provides the following JDBC drivers for use with the Derby database engine:

- *org.apache.derby.jdbc.EmbeddedDriver*

  For embedded environments, when Derby runs in the same JVM as the application

- *org.apache.derby.jdbc.ClientDriver*

  For using the Network Client to connect to the Derby Network Server

You can use `ij` to connect to any database that supplies a JDBC driver. For those databases, you would need to load the supplied JDBC driver.

Database connection URLs

A JDBC URL provides a way of identifying a database so that the appropriate driver recognizes it and connects to it. In the Derby documents, a JDBC URL is referred to as a database connection URL.

After the driver is loaded, an application must specify the correct database connection URL to connect to a specific database. The Derby database connection URL allows you to accomplish tasks other than simply connecting. For more information about the Derby database connection URLs, see the *Derby Developer's Guide* .

A JDBC URL always starts with *jdbc:*. After that, the format for the database connection URL depends on the JDBC driver.

Here is the format for the database connection URL for connecting to an existing database using the embedded driver:

- jdbc:derby:*databaseName*;*URLAttributes*

The format for the database connection URL for connecting to an existing database using

the Network Client is:

- jdbc:derby://*host:port*/*databaseName*

URL attributes can be passed to the Network Client driver by using double quotes (") around the database name portion of the URL, as follows:

- jdbc:derby://*host:port*/*"databaseName;URLAttributes";*

The italicized items stand for something the user fills in:

- *databaseName*

  The name of the database you want to connect to

- *URLAttributes*

  One or more of the supported attributes of the database connection URL, such as ;*territory=ll_CC* or ;*create=true*. For more information, see the *Derby Developer's Guide* .

- *host*

  The name of the machine where the server is running. It can be the name of the machine or the address.

- *port*

  The port number of the server framework

About Protocols

Officially, the portion of the database connection URL called the protocol is *jdbc:*, just as *http://* is a protocol in Web URLs. However, the second portion of the database connection URL (everything between *jdbc:* and *databaseName*), which is called the subprotocol, is informally considered part of the protocol. Later in this book you might see references to protocol. Consider protocol to be everything that comes before *databaseName*.

For complete information about the database connection URL, see the *Derby Developer's Guide* .

Tools and localization

The Derby tools provide support for common localization features such as localized message files and GUI, locale-appropriate formatting of data, codesets, unicode identifiers and data, and database territories.

For general information about international Derby systems, see the *Derby Developer's Guide* .

About locales

In the Derby documentation, we refer to three locales:

- *Java System locale*

  This is the locale of your machine, which is automatically detected by your JVM. For Derby and Derby tools, the Java system locale determines the default locale.

- *Database territory*

  This is the territory associated with your database when it is created. By default, this is the same as the java system locale . The database territory determines the language of database errors.

- *`ij` or `dblook` Session locale*

  This locale is associated with your `ij` or `dblook` session. This locale determines

the localized display format for numbers, dates, times, and timestamps.

Database territory

To specify a database territory, use the *territory* attribute on the URL connection when creating the database.
Note: You cannot modify a database's territory after the database has been created.

For information about database territories, see the Internationalization appendix in the *Derby Developer's Guide* .

Specifying an alternate codeset

You can specify an alternate codeset for your tool session.

Use the *derby.ui.codeset* property when starting `ij` or `dblook`. This property can be useful when working with scripts created on a different system.

Formatting display of locale-sensitive data

To display dates, timestamps, numbers, and times in the format of the `ij` Session locale, use the *LocalizedDisplay* command.
Note: These options do not change how Derby *stores* locale-sensitive data, simply how the tool displays the data.

The following example demonstrates using *localizedDisplay* in an *en_US* locale:

```
ij> VALUES CURRENT_DATE;
1
----------
2001-08-06
1 row selected
ij> localizeddisplay on;
ij> VALUES CURRENT_DATE;
1
------------------
September 6, 2001
1 row selected
```

Using ij

ij is Derby's interactive JDBC scripting tool. It is a simple utility for running scripts against a Derby database.

ij is a Java application, which you start from a command window such as an MS-DOS Command Window or the UNIX shell. ij provides several commands for ease in accessing a variety of JDBC features through scripts.

Starting ij

Derby provides batch and shell scripts for users in Windows and UNIX environments. If you put the appropriate script in your path, you will be able to start ij with a simple command. These scripts use the ij.protocol property, which automatically loads a driver and simplifies the process of connecting to a database. The scripts are found in the %DERBY_INSTALL%/bin/ directory. You can also customize the ij scripts to suit your environment.

If you are starting ij from a command line, be sure that the derbytools.jar file is in your classpath. If you are using Derby as a database server, start the server before connecting to the Derby database. You can start ij by running the ij scripts in the /frameworks/embedded/bin/ directory or in the /frameworks/NetworkServer/bin/ directory.

To start ij, run the script provided or use this command:

```
java
 [<
options
>] org.apache.derby.tools.ij [-p <
propertyFile
>] [<
inputFile
>]
```

The command line items are:

- java

  The JVM you want to run (java is the name of the JVM program).

- options

  The options that the JVM uses. You can use the -D command to set ij properties (see Starting ij using properties) or system properties, such as Derby properties.

- propertyFile

  A file you can use to set ij properties (instead of the -D command). The property file should be in the format created by the java.tools.Properties.save methods, which is the same format as the derby.properties file.

- inputFile

  A file from which to read commands. The ij tool exits at the end of the file or an exit command. Using an input file causes ij to print out the commands as it runs them. If you reroute standard input, ij does not print out the commands. If you do not supply an input file, ij reads from the standard input.

For detailed information about ij commands, see ij commands and errors reference .

Starting ij using properties

You set ij properties in any of the following ways:

1. by using the -D command on the command line
2. by specifying a properties file using the `-p propertyfile` option on the command line

Remember: ij property names are case-sensitive, while commands are case-insensitive.

The following examples illustrate how to use ij properties:

To start ij by using a properties file called `ij.properties`, use the following command:

```
java org.apache.derby.tools.ij -p ij.properties
```

To start ij with a maximumDisplayWidth of 1000:

```
java -Dij.maximumDisplayWidth=1000 org.apache.derby.tools.ij
```

To start ij with an ij.protocol of **jdbc:derby:** and an ij.database of **sample**, use the following command:

```
java -Dij.protocol=jdbc:derby: -Dij.database=sample
    org.apache.derby.tools.ij
```

To start ij with two named connections, using the ij.connection.connectionName property, use the following command:

```
java -Dij.connection.sample=jdbc:derby:sample
-Dij.connection.History=jdbc:derby:History
-Dderby.system.home=c:\derby\demo\databases
org.apache.derby.tools.ij
```

To see a list of connection names and the URL's used to connect to them, use the following command: (If there is a connection that is currently active, it will show up with an * after its name.)

```
ij version 10.1
ij(HISTORY)> show connections;
HISTORY* -    jdbc:derby:History
SAMPLE -     jdbc:derby:sample
* = current connection
ij(HISTORY)>
```

To start ij to connect to the Derby Network Server, you must specify the `user` and `password` attributes on the URL. For more information, see the *Derby Server and Administration Guide* .

Getting started with ij

This section discusses the use of the ij tool.

Connecting to a Derby database

To connect to a Derby database, you need to perform the following steps:

1. Load the appropriate driver.
2. Provide a database connection URL for the database.

In ij, there are three ways of accomplishing these steps:

- Full database connection URL

  ij can work with any JDBC driver. For drivers supplied by other vendors, you need to load the driver separately. For drivers supplied by Derby, you can load the driver by specifying the full database connection URL in the connection. You do not need to load the driver explicitly in a second step.

  To connect, specify the full database connection URL in a Connect command, ij.connection.connectionName property, or ij.database property.

  The protocol of the database connection URL must correspond to the driver provided by Derby (see Database connection URLs) or, if you are using another driver, to that driver. The following example shows how to connect to a Derby database by using the **Connect** command:

```
D:>java org.apache.derby.tools.ij
ij version 10.1
ij> connect 'jdbc:derby:sample';
ij>
```

- Protocol and short database connection URL

  For drivers supplied by Derby, specifying a protocol automatically loads the appropriate driver. You do not need to load the driver explicitly in a separate step. You specify a protocol with a property (ij.protocol or ij.protocol.protocolName) or command (Protocol).

  To connect, specify the "short form" of the database connection URL in a Connect command, ij.connection.connectionName property, or ij.database property. A short form of the database connection URL eliminates the protocol (For more information, see About Protocols).

```
D:>java org.apache.derby.tools.ij
ij version 10.1
ij> protocol 'jdbc:derby:';
ij> connect 'sample';
ij>

D:>java -Dij.protocol.myprotocolName=jdbc:derby:
    org.apache.derby.tools.ij
ij version 10.1
ij> connect 'sample' protocol myprotocolName;
ij>
```

- Driver and full database connection URL

  If you are using the drivers supplied by Derby, use the driver names listed in JDBC drivers overview . The Derby drivers are implicitly loaded when a supported protocol is used. Any other driver has to be explicitly loaded. You can load a driver explicitly with an ij property (ij.Driver), a system property (`jdbc.drivers`), or a command (Driver).

  To connect, specify the full database connection URL in a Connect command, ij.connection.connectionName property, or ij.database property.

```
D:>java org.apache.derby.tools.ij
ij version 10.1
ij> driver 'sun.jdbc.odbc.JdbcOdbcDriver';
ij> connect 'jdbc:odbc:myOdbcDataSource';
ij>
```

Specifying the driver name and database connection URL

Specifying the Driver Name and database connection URL , summarizes the different ways to specify the driver name and database connection URL.

Table1. Specifying the Driver Name and database connection URL

Action	System Property	ij Property	ij Command
loading the driver implicitly	'	*ij.connection.connectionName* (plus full URL) *ij.database* (plus full URL) *ij.protocol* *ij.protocol.protocolName* (plus protocol clause in Connect command)	*Protocol Connect* (plus full URL)
loading the driver explicitly	*jdbc.drivers*	-D *ij.Driver*	*Driver*
specifying the database connection URL	'	*ij.connection.connectionName* *ij.database*	*Connect*

Using ij commands

ij accepts a number of different commands that let you execute SQL statements or run scripts. Each ij statement must end with a semicolon.

For complete information about ij commands, see ij commands and errors reference .

Other uses for ij

ij is a JDBC-neutral scripting tool with a small command set. It can be used to access any JDBC driver and database accessible through that driver.

The main benefit of a tool such as ij is that it is easy to run scripts for creating a database schema and automating other repetitive database tasks.

In addition, ij accepts and processes SQL commands interactively for ad hoc database access.

Running ij scripts

You can run scripts in ij in any of the following ways:

- Name an input file as a command-line argument.

  For example:

  ```
  java -Djdbc.drivers=org.apache.derby.jdbc.EmbeddedDriver
      org.apache.derby.tools.ij <myscript.sql>
  ```

- Redirect standard input to come from a file.

  For example:

  ```
  java -Djdbc.drivers=org.apache.derby.jdbc.EmbeddedDriver
      org.apache.derby.tools.ij < <myscript.sql>
  ```

- Use the Run command from the ij command line.

  For example:

  ```
  ij> run 'myscript.sql';
  ```

Note: If you name an input file as a command-line argument or if you use the Run command, ij echoes input from a file. If you redirect standard input to come from a file, ij does not echo commands.

You can save output in any of the following ways:

- By redirecting output to a file:

```
java -Djdbc.drivers=org.apache.derby.jdbc.EmbeddedDriver
    org.apache.derby.tools.ij <myscript.sql> > <myoutput.txt>
```

- By setting the `ij.outfile` property:

```
java -Dij.outfile=<myoutput.txt> org.apache.derby.tools.ij
    <myscript.sql>
```

ij exits when you enter the Exit command or, if you give a command file on the Java invocation line, when the end of the command file is reached. When you use the Exit command, ij automatically shuts down an embedded Derby system by issuing a `connect jdbc:derby:;shutdown=true` request. It does not shut down Derby if it is running in a server framework.

ij properties reference

When starting up`ij`, you can specify properties on the command line or in a properties file, as described in Starting ij using properties .

ij.connection.connectionName

Function

Creates a named connection to the given database connection URL when `ij` starts up; it is equivalent to the Connect AS *Identifier* command. The database connection URL can be of the short form if an *ij.protocol* is specified. This property can be specified more than once per session, creating multiple connections. When `ij` starts, it displays the names of all the connections created in this way. It also displays the name of the current connection, if there is more than one, in the `ij` prompt.

Syntax

```
ij.connection.
connectionName
=
databaseConnectionURL
```

The *databaseConnectionURL* is not a string; do not enclose it in quotation marks.

Example

```
D:> java -Dij.connection.sample1=jdbc:derby:sample
-Dij.connection.anotherConn=jdbc:derby:anotherDB;create=true
    org.apache.derby.tools.ij
ij version 10.1
ANOTHERCONN* -  jdbc:derby:anotherDB;create=true
SAMPLE1 -       jdbc:derby:sample
* = current connection
ij(ANOTHERCONN)>
```

See also

- Connect

ij.database

Function

Creates a connection with a generated name to the given database connection URL when `ij` starts up, thus creating an initial connection (with a generated name) for the `ij` session. (If you have specified an *ij.protocol*, you can use a shortened form of the URL.) After it boots, `ij` displays the generated name of the connection made with this property.

Syntax

```
ij.database=
databaseConnectionURL
```

The *databaseConnectionURL* is not a string; do not enclose it in quotation marks.

Example

```
java -Dij.protocol=jdbc:derby: -Dij.connection.sample1=sample
    -Dij.connection.anotherConn=anotherDB
    -Dij.database=wombat;create=true org.apache.derby.tools.ij
ij version 10.1
CONNECTION2* -  jdbc:derby:wombat;create=true
ANOTHERCONN -   jdbc:derby:anotherDB
SAMPLE1 -       jdbc:derby:sample
* = current connection
ij(CONNECTION2)>
```

ij.driver

Function

Loads the JDBC driver that the class specifies.

Syntax

```
ij.driver=
JDBCDriverClassName
```

Notes

The driver property is a synonym.

Example

```
D:>java -Dij.driver=sun.jdbc.odbc.JdbcOdbcDriver
org.apache.derby.tools.ij
ij version 10.1
ij> Connect 'jdbc:odbc:MyODBCDataSource';
ij>
```

See also

- Driver

ij.maximumDisplayWidth

Function

Specifies the maximum number of characters used to display any column. The default value is 128. Values with display widths longer than the maximum are truncated and terminated with an *&* character.

Syntax

```
ij.maximumDisplayWidth=
numberOfCharacters
```

Example

```
java -Dij.maximumDisplayWidth=1000 org.apache.derby.tools.ij
```

See also

- MaximumDisplayWidth

ij.outfile

Function

Specifies a file to which the system should direct output for a session. Specify the file name relative to the current directory, or specify the absolute path.

Syntax

```
ij.outfile=
fileName
```

Example

```
java -Dij.outfile=out.txt org.apache.derby.tools.ij myscript.sql
```

ij.protocol

Function

Specifies the protocol and subprotocol portions of the database connection URL for connections. Automatically loads the appropriate driver for recognized subprotocol. The recognized protocol is:

- jdbc:derby:

Allows you to use a short form of a database name.

Syntax

```
ij.protocol=
protocolForEnvironment
```

Example

```
D:>java -Dij.protocol=jdbc:derby:
    org.apache.derby.tools.ij
ij version 10.1
ij> Connect 'newDB;create=true';
ij>
```

See also

- Protocol

ij.protocol.protocolName

Function

This property is similar to the *ij.protocol* property. The only difference is that it associates a name with the value, thus allowing you to define and use more than one protocol. (See Connect .)

Syntax

```
ij.protocol.
protocolName
=
protocolForEnvironment
```

Example

```
D:>java -Dij.protocol.derby=jdbc:derby:
```

```
-Dij.protocol.emp=jdbc:derby: org.apache.derby.tools.ij
ij version 10.1
ij> Connect 'newDB' protocol derby as new;
ij>
```

See also

- Protocol

ij.showErrorCode

Function

Set this property to *true* to have ij display the *SQLException ErrorCode* value with error messages. The default is *false*.

Error codes denote the severity of the error. For more information, see the *Derby Reference Manual* .

Syntax

```
ij.showErrorCode=
trueOrFalse
```

Example

```
java -Dij.showErrorCode=true -Dij.protocol=jdbc:derby:
    org.apache.derby.tools.ij
ij version 10.1
ij> Connect 'sample';
ij> VLUES 1;
ERROR 42X01: Syntax error: Encountered "VLUES"
at line 1, column 1. (errorCode = 30000)
ij>
```

ij.URLCheck

Function

This property determines whether `ij` checks for invalid or non-Derby URL attributes. Set this property to *false* to prevent `ij` from validating URL attributes. The default value is *true*.

When the *ij.URLCheck* property is set to *true*, you are notified whenever a connection URL contains an incorrectly specified attribute. For example if the attribute name is misspelled or cased incorrectly `ij` prints a message.
Note: `ij` checks attribute *values* if the attribute has pre-defined values. For example, the attribute *unicode* has the pre-defined values of *true* or *false*. If you try to set the attribute unicode to a value other than true or false, `ij` displays an error. For example:

```
ij> Connect 'jdbc:derby:anyDB;create=true;unicode=falj';
ERROR XJ05B: JDBC attribute 'unicode' has an invalid value 'falj',
valid values are '{true|false}'.
ij>
```

Syntax

```
ij.URLCheck={ false | true }
```

Example
By default, `ij` displays messages about invalid attributes:

```
java org.apache.derby.tools.ij
ij version 10.1
ij> connect 'mydb;uSer=naomi';
URL Attribute [uSer=naomi]
   Case of the Derby  attribute is incorrect.
```

The following command line specifies to turn off URL attribute checking in `ij`.

```
java -Dij.URLCheck=false org.apache.derby.tools.ij
ij version 10.1
ij> connect 'mydb;uSer=naomi';
ij>
```

Typically, you would only explicitly turn off the URL checker if you were using `ij` with a non-Derby JDBC driver or database.

Notes

URLCheck does not check the validity of properties, only database connection URL *attributes*.

`ij` recognizes the following attributes:

- *bootPassword*
- *create*
- *databaseName*
- *dataEncryption*
- *encryptionAlgorithm*
- *encryptionProvider*
- *territory*
- *logDevice*
- *password*
- *shutdown*
- *unicode*
- *upgrade*
- *user*

derby.ui.codeset

Function

Set this property to a supported character encoding value when using one of the Derby tools with a language not supported by your default system.

Syntax

```
derby.ui.codeset=
derbyval
```

where *derbyval* is a supported character encoding value, for example, UTF8 (see Sample Character Encodings).

Example

The following command line specifies to run `ij` using the Japanese territory (*territory=ja_JP*) using Japanese Latin Kanji mixed encoding (*codeset=Cp939*):

```
java -Dderby.ui.territory=ja_JP -Dderby.ui.codeset=Cp939
    -Dij.protocol=jdbc:derby:
    org.apache.derby.tools.ij
```

The following table contains a sampling of character encodings supported by the IBM Application Developer Kit. To see the full list, go to *http://java.sun.com/j2se/1.3/docs/guide/intl/encoding.doc.html* .

Table1. Sample Character Encodings

Character Encoding	Explanation
8859_1	ISO Latin-1
8859_2	ISO Latin-2
8859_7	ISO Latin/Greek
Cp1257	Windows Baltic
Cp1258	Windows Vietnamese
Cp437	PC Original
EUCJIS	Japanese EUC
GB2312	GB2312-80 Simplified Chinese
JIS	JIS
KSC5601	KSC5601 Korean
MacCroatian	Macintosh Croatian
MacCyrillic	Macintosh Cyrillic
SJIS	PC and Windows Japanese
UTF8	Standard UTF-8

ij commands and errors reference

This section describes the commands and errors eithin the ij tool.

ij commands

`ij` accepts several commands to control its use of JDBC. It recognizes a semicolon as the end of an `ij` or SQL command; it treats semicolons within SQL comments, strings, and delimited identifiers as part of those constructs, not as the end of the command. Semicolons are required at the end of an `ij` or SQL statement.

All `ij` commands, identifiers, and keywords are case-insensitive.

Commands can span multiple lines without any special escaping for the ends of lines. This means that if a string spans a line, the new lines will show up in the value in the string.

`ij` treats any command that it does not recognize as an SQL command to be passed to the underlying connection, so syntactic errors in `ij` commands will cause them to be handed to the SQL engine and will probably result in SQL parsing errors.

Conventions for ij examples

Examples in this document show input from the keyboard or a file in bold text and console output from the DOS prompt or the `ij` application in regular text.

```
C:\> REM This example is from a DOS prompt:
C:\> java -Dij.protocol=jdbc:derby: org.apache.derby.tools.ij
ij version 10.1
ij> connect 'menuDB;create=true';
ij> CREATE TABLE menu(course CHAR(10), item CHAR(20), price INTEGER);
0 rows inserted/updated/deleted
ij> disconnect;
ij> exit;
C:\>
```

ij SQL command behavior

Any command other than those documented in the `ij` command reference are handed to the current connection to execute directly. The statement's closing semicolon, used by `ij` to determine that it has ended, is not passed to the underlying connection. Only one statement at a time is passed to the connection. If the underlying connection itself accepts semicolon-separated statements (which Derby does not), they can be passed to the connection using `ij`'s Execute command to pass in a command string containing semicolon-separated commands.

`ij` uses the result of the JDBC execute request to determine whether it should print a number-of-rows message or display a result set.

If a JDBC execute request causes an exception, it displays the *SQLState*, if any, and error message.

Setting the `ij` property *ij.showErrorCode* to *true* displays the *SQLException*'s error code (see ij properties reference).

The number-of-rows message for inserts, updates, and deletes conforms to the JDBC specification for any SQL statement that does not have a result set. DDL (data definition language) commands typically report "0 rows inserted/updated/deleted" when they successfully complete.

To display a result set, `ij` formats a banner based on the JDBC *ResultSetMetaData* information returned from *getColumnLabel* and *getColumnWidth*. Long columns wrap the screen width, using multiple lines. An *&* character denotes truncation (`ij` limits displayed width of a column to 128 characters by default; see MaximumDisplayWidth).

`ij` displays rows as it fetches them. If the underlying DBMS materializes rows only as they are requested, ij displays a partial result followed by an error message if there is a error in fetching a row partway through the result set.

`ij` verifies that a connection exists before issuing statements against it and does not execute SQL when no connection has yet been made.

There is no support in `ij` for the JDBC feature multiple result sets.

ij command example

```
ij> INSERT INTO menu VALUES ('appetizer','baby greens',7),
('entree','lamb chops ',6),('dessert','creme brulee',14);
3 rows inserted/updated/deleted
ij> SELECT * FROM menu;
COURSE    |ITEM                |PRICE
-----------------------------------------
entree    |lamb chop           |14
dessert   |creme brulee        |6
appetizer |baby greens         |7

3 rows selected
ij>
```

Absolute

Syntax

```
ABSOLUTE
int
Identifier
```

Description
Moves the cursor to the row specified by the *int*, then fetches the row. The cursor must have been created with the Get Scroll Insensitive Cursor command. It displays a banner and the values of the row.
Note: This command works only in a Java 2 Platform, Standard Edition, v 1.2 (J2SE) or higher environment.

Example

```
ij> autocommit off;
ij> get scroll insensitive cursor scrollCursor as
'SELECT * FROM menu FOR UPDATE OF price';
ij> absolute 3 scrollCursor;
COURSE    |ITEM                |PRICE
-----------------------------------------
entree    |lamb chop           |14
```

After Last

Syntax

```
AFTER LAST
```

```
Identifier
```

Description

Moves the cursor to after the last row, then fetches the row. (Since there is no current row, it returns the message: `No current row."`

The cursor must have been created with the Get Scroll Insensitive Cursor command.
Note: This command works only in a Java 2 Platform, Standard Edition, v 1.2 (J2SE) or higher environment.

Example

```
ij> get scroll insensitive cursor scrollCursor as
'SELECT * FROM menu FOR UPDATE OF price';
ij> after last scrollcursor;
No current row
```

Async

Syntax

```
ASYNC

Identifier

String
```

Description

The ASYNC command lets you execute an SQL statement in a separate thread. It is used in conjunction with the Wait For command to get the results.

You supply the SQL statement, which is any valid SQL statement, as a *String*. The *Identifier* you must supply for the async SQL statement is used in the Wait For command and is a case-insensitive `ij` identifier; it must not be the same as any other identifier for an async statement on the current connection. You cannot reference a statement previously prepared and named by the `ij` Prepare command in this command.

`ij` creates a new thread in the current connection to issue the SQL statement. The separate thread is closed once the statement completes.

Example

```
ij>async aInsert 'INSERT into menu values (''entree'',''chicken'',11)';
ij>INSERT INTO menu VALUES ('dessert','ice cream',3);
1 rows inserted/updated/deleted.
ij>wait for aInsert;
1 rows inserted/updated/deleted.
-- the result of the asynchronous insert
```

Autocommit

Syntax

```
AUTOCOMMIT { ON | OFF }
```

Description

Turns the connection's auto-commit mode on or off. JDBC specifies that the default auto-commit mode is `ON`. Certain types of processing require that auto-commit mode be `OFF`. For information about auto-commit, see the *Derby Developer's Guide* .

If auto-commit mode is turned on when there is a transaction outstanding, that work is committed when the current transaction commits, not at the time auto-commit is turned on. Use Commit or Rollback before turning on auto-commit when there is a transaction outstanding, so that all prior work is completed before the return to auto-commit mode.

Example

```
ij> autocommit off;
ij> DROP TABLE menu;
0 rows inserted/updated/deleted
ij> CREATE TABLE menu (course CHAR(10), item CHAR(20), price INT);
0 rows inserted/updated/deleted
ij> INSERT INTO menu VALUES ('entree', 'lamb chop', 14),
('dessert', 'creme brulee', 6),
('appetizer', 'baby greens', 7);
3 rows inserted/updated/deleted
ij> commit;
ij> autocommit on;
ij>
```

Before First

Syntax

```
BEFORE FIRST
int
Identifier
```

Description

Moves the cursor to before the first row, then fetches the row. (Since there is no current row, it returns the message `No current row`.)

The cursor must have been created with the Get Scroll Insensitive Cursor command.
Note: This command works only in a Java 2 Platform, Standard Edition, v 1.2 (J2SE) or higher environment.

Example

```
ij> get scroll insensitive cursor scrollCursor as
'SELECT * FROM menu FOR UPDATE OF price';
ij> before first scrollcursor;
No current row
```

Close

Syntax

```
CLOSE

Identifier
```

Description

Closes the named cursor. The cursor must have previously been successfully created with the `ij` Get Cursor or Get Scroll Insensitive Cursor commands.

Example

```
ij> get cursor menuCursor as 'SELECT * FROM menu';
ij> next menuCursor;
COURSE     |ITEM                |PRICE
-------------------------------------------
entree     |lamb chop           |14
ij> next menuCursor;
COURSE     |ITEM                |PRICE
-------------------------------------------
dessert    |creme brulee        |6
ij> close menuCursor;
ij>
```

Commit

Syntax

```
COMMIT
```

Description

Issues a *java.sql.Connection.commit* request. Use this command only if auto-commit is off. A *java.sql.Connection.commit* request commits the currently active transaction and initiates a new transaction.

Example

```
ij> commit;
ij>
```

Connect

Syntax

```
CONNECT

String

 [ PROTOCOL

Identifier

 ]
   [ AS

Identifier

 ] [ USER

String


   PASSWORD

String
```

```
]
```

Description

Takes the value of the string database connection URL and issues a `java.sql.DriverManager.getConnection` request to set the current connection to that database connection URL.

You have the option of specifying a name for your connection. Use the Set Connection command to switch between connections. If you do not name a connection, the system generates a name automatically.

You also have the option of specifying a named protocol previously created with the Protocol command or the ij.protocol.protocolName property.

If the connection requires a user name and password, supply those with the optional user and password parameters.

If the connect succeeds, the connection becomes the current one and ij displays a new prompt for the Next command. If you have more than one open connection, the name of the connection appears in the prompt.

All further commands are processed against the new connection.

Example

```
ij> connect 'jdbc:derby:menuDB;create=true';
ij> -- we create a new table in menuDB:
CREATE TABLE menu(course CHAR(10), item CHAR(20), price INTEGER);
ij> protocol 'jdbc:derby:';
ij> connect 'sample' as sample1;
ij(SAMPLE1)> connect 'newDB;create=true' as newDB;
ij(NEWDB)> show connections;
CONNECTION0 - jdbc:derby:menuDB
NEWDB* - jdbc:derby:anotherDB
SAMPLE1 - jdbc:derby:newDB
ij>
ij> connect 'jdbc:derby:sample' user 'sa' password 'cloud3x9';
ij>
```

Disconnect

Syntax

```
DISCONNECT [ ALL | CURRENT |

Identifier

]
```

Description

Issues a `java.sql.Connection.close` request against the current connection. There must be a current connection at the time the request is made.

If ALL is specified, all known connections are closed and there is no current connection.

Disconnect CURRENT is the same as Disconnect.

If a connection name is specified with an identifier, the command disconnects the named connection. The name must be the name of a connection in the current session provided with the ij.connection.connectionName property or with the Connect command.

If the ij.database property or the Connect command without the AS clause was used, you can supply the name the system generated for the connection. If the current connection is the named connection, when the command completes, there will be no current connection and you must issue a Set Connection or Connect command.

A Disconnect command issued against a Derby connection does not shut down the database or Derby (but the Exit command does).

Example

```
ij> connect 'jdbc:derby:menuDB;create=true';
ij> -- we create a new table in menuDB:
CREATE TABLE menu(course CHAR(10), ITEM char(20), PRICE integer);
0 rows inserted/updated/deleted
ij> disconnect;

ij> protocol 'jdbc:derby:';
ij> connect 'sample' as sample1;
ij> connect 'newDB;create=true' as newDB;
SAMPLE1 -    jdbc:derby:sample
NEWDB* -    jdbc:derby:newDB;create=true
* = current connection
ij(NEWDB)> set connection sample1;
ij> disconnect sample1;
ij> disconnect all;
ij>
```

Driver

Syntax

```
DRIVER

String
```

Description

Takes the value of the string and issues a *Class.forName* request to load the named class. The class is expected to be a JDBC driver that registers itself with *java.sql.DriverManager*.

If the Driver command succeeds, a new `ij` prompt appears for the next command.

Example

```
ij> -- load the Derby driver so that a connection
-- can be made:
driver 'org.apache.derby.jdbc.EmbeddedDriver';
ij> connect 'jdbc:derby:menuDB;create=true';
ij>
```

Elapsedtime

Syntax

```
ELAPSEDTIME { ON | OFF }
```

Description

When `elapsedtime` is turned on, `ij` displays the total time elapsed during statement execution. The default value is OFF.

Example

```
ij> elapsedtime on;
ij> VALUES current_date;
1
----------
1998-07-15
ELAPSED TIME = 2134 milliseconds
ij>
```

Execute

Syntax

```
EXECUTE {

String

 |

Identifier

 }
[ USING {

String

 |

Identifier

 } ]
```

Description

Has several uses:

- To execute an SQL command that has the same name as an `ij` command, using the Execute *String* style. The String is passed to the connection without further examination or processing by `ij`. *Normally, you execute SQL commands directly, not with the Execute command*.
- To execute a named command previously prepared with the `ij` Prepare command, using the Execute *Identifier* style.
- To execute either flavor of command when that command contains dynamic parameters, taking values from the Using portion of the command. In this style, the Using portion's *String* or previously prepared *Identifier* is executed, and it must have a result set as its result. Each row of the result set is applied to the input parameters of the command to be executed, so the number of columns in the Using's result set must match the number of input parameters in the Execute 's statement. The results of each execution of the Execute statement are displayed as they are made. If the Using's result set contains no rows, the Execute 's statement is not executed.

  When auto-commit mode is on, the Using's result set is closed upon the first execution of the Execute statement. To ensure multiple-row execution of the Execute command, use the Autocommit command to turn auto-commit off.

Example

```
ij> autocommit off;
ij> prepare menuInsert as 'INSERT INTO menu VALUES (?, ?, ?)';
ij> execute menuInsert using 'VALUES
    (''entree'', ''lamb chop'', 14),
    (''dessert'', ''creme brulee'', 6)';
1 row inserted/updated/deleted
```

```
1 row inserted/updated/deleted
ij> commit;
```

Exit

Syntax

```
EXIT
```

Description

Causes the `ij` application to complete and processing to halt. Issuing this command from within a file started with the Run command or on the command line causes the outermost input loop to halt.

`ij` automatically shuts down a Derby database running in an embedded environment (issues a *Connect 'jdbc:derby:;shutdown=true*' request) on exit.

`ij` exits when the Exit command is entered or if given a command file on the Java invocation line, when the end of the command file is reached.

Example

```
ij> disconnect;
ij> exit;
C:\>
```

First

Syntax

```
FIRST

Identifier
```

Description

Moves the cursor to the first row in the *ResultSet*, then fetches the row. The cursor must have been created with the Get Scroll Insensitive Cursor command. It displays a banner and the values of the row.
Note: This command works only in a Java 2 Platform, Standard Edition, v 1.2 (J2SE) or higher environment.

Example

```
ij> get scroll insensitive cursor scrollCursor as
'SELECT * FROM menu FOR UPDATE OF price';
ij> first scrollcursor;
COURSE    |ITEM                |PRICE
-----------------------------------------
entree    |lamb chop           |14
```

Get Cursor

Syntax

```
GET [WITH {HOLD|NOHOLD}] CURSOR
```

```
Identifier

 AS

String
```

WITH HOLD is the default attribute of the cursor. For a non-holdable cursor, use the WITH NOHOLD option.

Note: WITH NOHOLD is only available in Java 2 Platform, Standard Edition, v 1.4 (J2SE) or higher.

Description

Creates a cursor with the name of the *Identifier* by issuing a *java.sql.Statement.executeQuery* request on the value of the *String*.

If the *String* is a statement that does not generate a result set, the behavior of the underlying database determines whether an empty result set or an error is issued. If there is an error in executing the statement, no cursor is created.

`ij` sets the cursor name using a *java.sql.Statement.setCursorName* request. Behavior with respect to duplicate cursor names is controlled by the underlying database. Derby does not allow multiple open cursors with the same name.

Once a cursor has been created, the `ij` Next and Close commands can be used to step through its rows, and if the connection supports positioned update and delete commands, they can be issued to alter the rows.

Example

```
ij> -- autocommit needs to be off so that the positioned update
ij> -- can see the cursor it operates against.
ij> autocommit off;
ij> get cursor menuCursor as
'SELECT * FROM menu FOR UPDATE OF price';
ij> next menuCursor;
COURSE    |ITEM                |PRICE
-------------------------------------------
entree    |lamb chop           |14
ij> next menuCursor;
COURSE    |ITEM                |PRICE
-------------------------------------------
dessert   |creme brulee        |6
ij> UPDATE menu SET price=price+1 WHERE CURRENT OF menuCursor;
1 row inserted/updated/deleted
ij> next menuCursor;
COURSE    |ITEM                |PRICE
-------------------------------------------
appetizer |baby greens salad   |7
ij> close menuCursor;
ij> commit;
ij>
```

Get Scroll Insensitive Cursor

Syntax

```
GET SCROLL INSENSITIVE [WITH {HOLD|NOHOLD}]
   CURSOR

Identifier

 AS
```

```
String
```

WITH HOLD is the default attribute of the cursor. For a non-holdable cursor, use the WITH NOHOLD option.

Note: WITH NOHOLD is only available in Java 2 Platform, Standard Edition, v 1.4 (J2SE) or higher.

Description
Creates a scroll insensitive cursor with the name of the *Identifier*. (It does this by issuing a *createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY*) call and then executing the statement with *java.sql.StatementExecuteQuery* request on the value of the String.)
Note: This command only works in a Java 2 Platform, Standard Edition, v 1.2 (J2SE) or higher environment.

Scroll insensitive cursors are not updatable.

If the *String* is a statement that does not generate a result set, the behavior of the underlying database determines whether an empty result set or an error is issued. If there is an error in executing the statement, no cursor is created.

ij sets the cursor name using a *java.sql.Statement.setCursorName* request. Behavior with respect to duplicate cursor names is controlled by the underlying database. Derby does not allow multiple open cursors with the same name.

Once a scrolling cursor has been created, you can use the follow commands to work with the result set:

- Absolute
- After Last
- Before First
- Close
- First
- Last
- Next
- Previous
- Relative

Example

```
ij> autocommit off;
ij> get scroll insensitive cursor scrollCursor as
'SELECT * FROM menu FOR UPDATE OF price';
ij> absolute 5 scrollCursor;
COURSE    |ITEM                |PRICE
-----------------------------------------------
entree    |lamb chop           |14
ij> after last scrollcursor;
No current row
ij> before first scrollcursor;
No current row
ij> first scrollcursor;
COURSE    |ITEM                |PRICE
-----------------------------------------------
entree    |lamb chop           |14
ij> last scrollcursor;
COURSE    |ITEM                |PRICE
-----------------------------------------------
dessert   |creme brulee        |6
ij> previous scrollcursor;
COURSE    |ITEM                |PRICE
-----------------------------------------------
entree    |lamb chop           |14
```

```
ij> relative 1 scrollcursor;
COURSE    |ITEM                |PRICE
-----------------------------------------
dessert   |creme brulee        |6
ij>>previous scrollcursor;
COURSE    |ITEM                |PRICE
-----------------------------------------
dessert   |creme brulee        |6
ij> next scrollcursor;
COURSE    |ITEM                |PRICE
-----------------------------------------
dessert   |creme brulee        |6
```

Help

Syntax

```
HELP
```

Description

Prints out a brief list of the `ij` commands.

Last

Syntax

```
LAST

Identifier
```

Description

Moves the cursor to the last row in the *ResultSet*, then fetches the row. The cursor must have been created with the Get Scroll Insensitive Cursor command. It displays a banner and the values of the row.
Note: This command works only in a Java 2 Platform, Standard Edition, v 1.2 (J2SE) or higher environment.

Example

```
ij> get scroll insensitive cursor scrollCursor as
'SELECT * FROM menu FOR UPDATE OF price';
ij> absolute 5 scrollCursor;
COURSE    |ITEM                |PRICE
-----------------------------------------
entree    |lamb chop           |14
ij> last scrollcursor;
COURSE    |ITEM                |PRICE
-----------------------------------------
dessert   |creme brulee        |6
```

LocalizedDisplay

Syntax

```
LOCALIZEDDISPLAY { on | off }
```

Description

Specifies to display locale-sensitive data (such as dates) in the native format for the `ij` locale. The `ij` locale is the same as the Java system locale.

Example

The following demonstrates *LocalizedDisplay* in an English locale:

```
ij> VALUES CURRENT_DATE;
1
----------
2000-05-01
1 row selected
ij> localizeddisplay on;
ij> VALUES CURRENT_DATE;
1
------------------
May 1, 2000
1 row selected
```

MaximumDisplayWidth

Syntax

```
MAXIMUMDISPLAYWIDTH
integer_value
```

Description

Sets the display width for column to the specified value.

Example

```
ij> maximumdisplaywidth 3;
ij> VALUES 'NOW IS THE TIME!';
1
---
NOW
ij> maximumdisplaywidth 30;
ij> VALUES 'NOW IS THE TIME!';
1
----------------
NOW IS THE TIME!
```

Next

Syntax

```
NEXT

Identifier
```

Description

Fetches the next row from the named cursor created with the Get Cursor command or Get Scroll Insensitive Cursor . It displays a banner and the values of the row.

Example

```
ij> get cursor menuCursor as 'SELECT * FROM menu';
ij> next menuCursor;
COURSE    |ITEM                |PRICE
--------------------------------------------
entree    |lamb chop           |14
ij>
```

Prepare

Syntax

```
PREPARE

Identifier

 AS

String
```

Description

Creates a *java.sql.PreparedStatement* using the value of the String, accessible in `ij` by the *Identifier* given to it. If a prepared statement with that name already exists in `ij`, an error will be returned and the previous prepared statement will remain. Use the Remove command to remove the previous statement first. If there are any errors in preparing the statement, no prepared statement is created.

Any SQL statements allowed in the underlying connection's prepared statement can be prepared with this command.

Example

```
ij> prepare seeMenu as 'SELECT * FROM menu';
ij> execute seeMenu;
COURSE    |ITEM                |PRICE
-------------------------------------------
entree    |lamb chop           |14
dessert   |creme brulee        |6

2 rows selected
ij>
```

Previous

Syntax

```
PREVIOUS

Identifier
```

Description

Moves the cursor to the row previous to the current one, then fetches the row. The cursor must have been created with the Get Scroll Insensitive Cursor command. It displays a banner and the values of the row.
Note: This command works only in a Java 2 Platform, Standard Edition, v 1.2 (J2SE) or higher environment.

Example

```
ij> get scroll insensitive cursor scrollCursor as
'SELECT * FROM menu FOR UPDATE OF price';
ij> last scrollcursor;
```

```
COURSE    |ITEM                |PRICE
-------------------------------------------
dessert   |creme brulee        |6
ij> previous scrollcursor;
COURSE    |ITEM                |PRICE
-------------------------------------------
entree    |lamb chop           |14
```

Protocol

Syntax

```
PROTOCOL

String

 [ AS

Identifier

 ]
```

Description

Specifies the protocol, as a String, for establishing connections and automatically loads the appropriate driver. *Protocol* is the part of the database connection URL syntax appropriate for your environment, including the JDBC protocol and the protocol specific to Derby. For further information about the Derby database connection URL, see the *Derby Developer's Guide* . Only Derby protocols are supported. Those protocols are listed in ij.protocol .

Providing a protocol allows you to use a shortened database connection URL for connections. You can provide only the database name instead of the full protocol. In addition, you do not need to use the Driver command or specify a driver at start-up, since the driver is loaded automatically.

If you name the protocol, you can refer to the protocol name in the Connect command.

Example

```
ij> protocol 'jdbc:derby:';
ij> connect 'sample';
```

Readonly

Syntax

```
READONLY { ON | OFF }
```

Description

Sets the current connection to a "read-only" connection, as if the current user were defined as a *readOnlyAccess* user. (For more information about database authorization, see the *Derby Developer's Guide* .)

Example

```
ij> connect 'jdbc:derby:menuDB';
ij> readonly on;
ij> SELECT * FROM menu;
COURSE    |ITEM                |PRICE
-------------------------------------------
```

```
entree    |lamb chop           |14
dessert   |creme brulee        |6
appetizer |baby greens         |7
entree    |lamb chop           |14
entree    |lamb chop           |14
dessert   |creme brulee        |6
6 rows selected
ij> UPDATE menu set price = 3;
ERROR 25502: An SQL data change is not permitted for a read-only
connection, user or database.
```

Relative

Syntax

```
RELATIVE
int
Identifier
```

Description

Moves the cursor to the row that is *int* number of rows relative to the current row, then fetches the row. The cursor must have been created with the Get Scroll Insensitive Cursor command. It displays a banner and the values of the row.
Note: This command works only in a Java 2 Platform, Standard Edition, v 1.2 (J2SE) or higher environment.

Example

```
ij> -- autocommit needs to be off so that the positioned update
ij> -- can see the cursor it operates against.
ij> autocommit off;
ij> get scroll insensitive cursor scrollCursor as
'SELECT * FROM menu FOR UPDATE OF price';
ij> last scrollcursor;
COURSE    |ITEM                |PRICE
-------------------------------------------
dessert   |creme brulee        |6
ij> previous scrollcursor;
COURSE    |ITEM                |PRICE
-------------------------------------------
entree    |lamb chop           |14
ij> relative 1 scrollcursor;
COURSE    |ITEM                |PRICE
-------------------------------------------
dessert   |creme brulee        |6
```

Remove

Syntax

```
REMOVE

Identifier
```

Description

Removes a previously prepared statement from ij. The identifier is the name by which the statement was prepared. The statement is closed to release its database resources.

Example

```
ij> prepare seeMenu as 'SELECT * FROM menu';
ij> execute seeMenu;
COURSE    |ITEM                |PRICE
-------------------------------------------
entree    |lamb chop           |14
dessert   |creme brulee        |6

2 rows selected
ij> remove seeMenu;
ij> execute seeMenu;
IJ ERROR: Unable to establish prepared statement SEEMENU
ij>
```

Rollback

Syntax

```
ROLLBACK
```

Description

Issues a *java.sql.Connection.rollback* request. Use only if auto-commit is off. A *java.sql.Connection.rollback* request undoes the currently active transaction and initiates a new transaction.

Example

```
ij> autocommit off;
ij> INSERT INTO menu VALUES ('dessert', 'rhubarb pie', 4);
1 row inserted/updated/deleted
ij> SELECT * from menu;
COURSE    |ITEM                |PRICE
-------------------------------------------
entree    |lamb chop           |14
dessert   |creme brulee        |7
appetizer |baby greens         |7
dessert   |rhubarb pie         |4

4 rows selected
ij> rollback;
ij> SELECT * FROM menu;
COURSE    |ITEM                |PRICE
-------------------------------------------
entree    |lamb chop           |14
dessert   |creme brulee        |7
appetizer |baby greens         |7

3 rows selected
ij>
```

Run

Syntax

```
RUN

String
```

Description

Assumes that the value of the string is a valid file name, and redirects `ij` processing to read from that file until it ends or an Exit command is executed. If the end of the file is reached without `ij` exiting, reading will continue from the previous input source once the end of the file is reached. Files can contain Run commands.

`ij` prints out the statements in the file as it executes them.

Any changes made to the `ij` environment by the file are visible in the environment when processing resumes.

Example

```
ij> run 'setupMenuConn.ij';
ij> -- this is setupMenuConn.ij
-- ij displays its contents as it processes file
ij> connect 'jdbc:derby:menuDB';
ij> autocommit off;
ij> -- this is the end of setupMenuConn.ij
-- there is now a connection to menuDB and no autocommit.
-- input will now resume from the previous source.
;
ij>
```

Set Connection

Syntax

```
SET CONNECTION

Identifier
```

Description

Allows you to specify which connection to make current when you have more than one connection open. Use the Show Connections command to display open connections.

If there is no such connection, an error results and the current connection is unchanged.

Example

```
ij> protocol 'jdbc:derby:';
ij> connect 'sample' as sample1;
ij> connect 'newDB;create=true' as newDB;
ij (NEWDB)> show connections;
SAMPLE1 -    jdbc:derby:sample
NEWDB* -    jdbc:derby:newDB;create=true
* = current connection
ij(NEWDB)> set connection sample1;
ij(SAMPLE1)> disconnect all;
ij>
```

Show Connections

Syntax

```
SHOW CONNECTIONS
```

Description

If there are no connections, the command returns "No connections available".

Otherwise, the command displays a list of connection names and the URLs used to connect to them. The currently active connection, if there is one, is marked with an * after its name.

Example

```
ij> connect 'sample' as sample1;
ij> connect 'newDB;create=true' as newDB;
ij(NEWDB)> show connections;
SAMPLE1 -        jdbc:derby:sample
NEWDB* -        jdbc:derby:newDB;create=true
* = current connection
ij(NEWDB)>
```

Wait For

Syntax

```
WAIT FOR

Identifier
```

Description

Displays the results of a previously started asynchronous command.

The identifier for the asynchronous command must have been used in a previous Async command on this connection. The Wait For command waits for the SQL statement to complete execution, if it has not already, and then displays the results. If the statement returns a result set, the Wait For command steps through the rows, not the Async command. This might result in further execution time passing during the result display.

Example

See Async .

Comment

Syntax

```
--
Text
```

Description

You can use a comment anywhere within an `ij` command and as permitted by the underlying connection within SQL commands. The comment is ended at the first new line encountered in the text.

Comments are ignored on input and have no effect on the output displayed.

Example

```
ij> -- this is a comment;
-- the semicolons in the comment are not taken as the end
-- of the command; for that, we put it outside the --:
;
ij>
```

Identifier

Syntax

```
Identifier
```

Description

Some `ij` commands require identifiers. These `ij` identifiers are case-insensitive. They must begin with a letter in the range A-Z, and can consist of any number of letters in the range A-Z, digits in the range 0-9, and underscore (_) characters.

These identifiers exist within the scope of `ij` only and are distinct from any identifiers used in SQL commands, except in the case of the Get Cursor command. The Get Cursor command specifies a cursor name to use in creating a result set.

`ij` does not recognize or permit delimited identifiers in `ij` commands. They can be used in SQL commands.

Example

```
These are valid ij identifiers:
foo1
exampleIdentifier12345
another_one
```

String

Syntax

```
'Text'
```

Description

Some `ij` commands require strings. `ij` strings are represented by the same literal format as SQL strings and are delimited by single quotation marks. To include a single quotation mark in a string, you must use two single quotation marks, as shown in the examples below. `ij` places no limitation on the lengths of strings, and will treat embedded new lines in the string as characters in the string.

Some `ij` commands execute SQL commands specified as strings. Therefore, you must double any single quotation marks within such strings, as shown in the second example below.

The cases of letters within a string are preserved.

Example

```
This is a string in ij     And this is its value
'Mary''s umbrella'          Mary's umbrella
'hello world'               hello world

--returns Joe's
execute 'VALUES ''Joe''''s''';
```

ij errors

ij might issue messages to inform the user of errors during processing of statements.

ERROR SQLState

When the underlying JDBC driver returns an *SQLException*, `ij` displays the *SQLException* message with the prefix "ERROR SQLState". If the *SQLException* has no SQLState associated with it, the prefix "ERROR (no SQLState)" is used.

WARNING SQLState

Upon completion of execution of any JDBC request, `ij` will issue a *getWarnings* request and display the SQLWarnings that are returned. Each *SQLWarning* message is displayed with the prefix "WARNING *SQLState*". If an *SQLWarning* has no SQLState associated with it, the prefix "WARNING (no SQLState)" is used.

IJ ERROR

When `ij` runs into errors processing user commands, such as being unable to open the file named in a Run command or not having a connection to disconnect from, it prints out a message with the prefix "IJ ERROR".

IJ WARNING

`ij` displays warning messages to let the user know if behavior might be unexpected. `ij` warnings are prefixed with "IJ WARNING".

JAVA ERROR

When an unexpected Java exception occurs, `ij` prints a message with the prefix "JAVA ERROR".

Using the bulk import and export procedures

You might want to import or export a large amount of data between files and the database. Instead of having to use INSERT and SELECT statements, you can use Derby procedures to import data directly from files into tables and to export data from tables into files.

Bulk Import/Export overview

Derby provides import and export system procedures that you can use to import and export data in delimited data file format.

- Use export procedures to write data from a database to one or more files that are stored outside of the database. You can use a procedure to export data from a table into a file or export data from a SELECT statement result into a file.
- Use import procedures to import data from a file into a table. If the target table already contains data, you can replace or append to the existing data.

You can perform an Import or Export operation from `ij` or from within an SQL statement.

Options for running the import and export procedures

You can run the import/export procedures from within an SQL statement using `ij` or any Java application.

Import/Export reads and writes only text files. Import does not support read-once streams (live data feeds), because it reads the first line of the file to determine the number of columns, then reads it again to import the data.
Note: These server-side utilities exhibit different behavior in client/server mode. Typically, you use them to import data into and export data from a locally running Derby. However, you can use the import/export procedures when Derby is running in a server framework if you specify import and export files that are accessible to the server.

Bulk import/export requirements and considerations

The table must exist.
For you to import data into a table, the table must already exist in Derby. The table does not have to be empty. If the table is not empty, bulk import performs single inserts which results in slower performance.

Create indexes and primary key, foreign key, and unique constraints first.
To avoid a separate create index step, create indexes and primary keys on tables before you import data. However, if your memory and disk spaces resources are limited, you can build the indexes and primary keys after importing data.

Data types.
You can import and export only data of the non-binary, *built-in* data types. Derby implicitly converts the strings to the data type of the receiving column. If any of the implicit conversions fail, the whole import is aborted. For example, "3+7" cannot be converted into an integer. An export that encounters a runtime error stops.
Note: You cannot import or export the binary data types: BLOB, CLOB, CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, or LONG VARCHAR FOR BIT DATA.

Locking during import.
Import procedures use the same isolation level as the connection in which they are executed to insert data into tables. During import, the entire table is exclusively locked irrespective of the isolation level.

Locking during export.
Export procedures use the same isolation level as the connection in which they are executed to fetch data from tables.

Import behavior on tables with triggers.
The import procedure fires INSERT triggers when data is appended to the table. The REPLACE option is not allowed when triggers are enabled on the table.

Restrictions on the REPLACE option.
If a table that receives imported data already contains data, you can either replace or append to the existing data. You can use the REPLACE option on tables that have dependent tables, but the replaced data must maintain referential integrity, otherwise the import operation will be rolled back. You cannot use the REPLACE option if the table has enabled triggers.

Restrictions on tables.
You cannot use import procedures to import data into a system table or a declared temporary table.

Bulk import and export

You can use import and export procedures which are executable from ij or any Java program. You must have derbytools.jar in your classpath to use the import or export procedures from ij.

To invoke an import or export procedure, you must be connected to the database into which data is imported or from which data is exported. Because the procedures will issue a COMMIT or a ROLLBACK statement, you should perform either a COMMIT or ROLLBACK to complete all transactions and release all table-level locks prior to invoking the import or export procedure. Other user applications that access the table with a separate connection do not need to disconnect.

Note: Imports are transactional. If an error occurs during bulk import, all changes are rolled back.

Bulk-Import

Derby provides two import procedures you can use to perform bulk-import operations:

1. To import data from a file to a table, use the SYSCS_UTIL.SYSCS_IMPORT_TABLE procedure. The procedure definition is:

```
SYSCS_UTIL.SYSCS_IMPORT_TABLE (IN
schemaName
 VARCHAR(128),
IN
tableName
 VARCHAR(128), IN
fileName
 VARCHAR(32672),
IN
columnDelimiter
 CHAR(1), IN
characterDelimiter
  CHAR(1),
IN
codeset
 VARCHAR(128), IN
replace
 SMALLINT)
```

   No Result is returned from the procedure.
2. To import data from a file to a subset of columns in a table, use the SYSCS_UTIL.SYSCS_IMPORT_DATA procedure. To import data to subset of columns in a table, you specify *insertColumns* on the table into which data will be imported and/or specify *columnIndexes* to import data fields from a file to columns in a table. The procedure definition is:

```
SYSCS_UTIL.SYSCS_IMPORT_DATA (IN
schemaName
 VARCHAR(128),
IN
tableName
 VARCHAR(128), IN
insertColumns
 VARCHAR(32672),
IN
columnIndexes
```

```
 VARCHAR(32672), IN
fileName
 VARCHAR(32672),
IN
columnDelimiter
 CHAR(1), IN
characterDelimiter
  CHAR(1),
IN
codeset
 VARCHAR(128), IN
replace
 SMALLINT)
```

No result is returned from the procedure.

Arguments to the import procedure

- *schemaName*

  Specifies the schema of the table. You can pass a NULL value to use the default schema name.

- *tableName*

  Specifies the table name of the table into which the data is to be imported. This table cannot be a system table or a declared temporary table. The string must exactly match case of the table name. Passing a null will result in an error.

- *insertColumns*

  Specifies the comma separated column names of the table into which the data will be imported. You can pass a NULL value to import into all columns of the table.

- *columnIndexes*

  Specifies the comma separated column indexes (numbered from one) of the input data fields that will be imported. You can pass a NULL value to use all input data fields in the file.

- *fileName*

  Specifies the file that contains the data to be imported. If the path is omitted, the current working directory is used. The specified location of the file should refer to the server side location if using the Network Server. Passing a null will result in an error.

- *columnDelimiter*

  Specifies a column delimiter. The specified character is used in place of a comma to signify the end of a column. You can pass a NULL value to use the default value of a comma.

- *characterDelimiter*

  Specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. You can pass a NULL value to use the default value of a double quotation mark.

- *codeset*

  Specifies the code set of the data in the input file. The code set name should be one of the Java-supported character encoding sets. Data is converted from the specified code set to the database code set (`UTF-8`). You can pass a NULL value to interpret the data file in the same code set as the JVM in which it is being executed.

- *replace*

  A non-zero value for the `replace` parameter will import in REPLACE mode, while a zero value will import in INSERT mode. REPLACE mode deletes all existing data from the table by truncating the table and inserts the imported data. The table definition and the index definitions are not changed. You can only import with REPLACE mode if the table already exists. INSERT mode adds the imported data

to the table without changing the existing table data. Passing a null value will result in an error.

If you create a schema, table, or column name as a non-delimited identifier, you must pass the name to the import procedure using all upper-case characters. If you created a schema, table, or column name as a delimited identifier, you must pass the name to the import procedure using the same case that was used when it was created.

Bulk-Export

Derby provides two export procedures you can use to perform bulk-export operations:

1. To export all the data from a table to a file, use the SYSCS_UTIL.SYSCS_EXPORT_TABLE procedure. The procedure definition is:

```
SYSCS_UTIL.SYSCS_EXPORT_TABLE (IN
schemaName
 VARCHAR(128),
IN
tableName
 VARCHAR(128), IN
fileName
 VARCHAR(32672),
IN
columnDelimiter
 CHAR(1), IN
characterDelimiter
 CHAR(1),
IN
codeset
 VARCHAR(128))
```

   No Result is returned from the procedure.
2. To export the result of a SELECT statement to a file, use the SYSCS_UTIL.SYSCS_EXPORT_QUERY procedure. The procedure definition is:

```
SYSCS_UTIL.SYSCS_EXPORT_QUERY(IN
selectStatement
 VARCHAR(32672),
IN
tableName
 VARCHAR(128), IN
fileName
 VARCHAR(32672),
IN
columnDelimiter
 CHAR(1), IN
characterDelimiter
 CHAR(1),
IN
codeset
 VARCHAR(128))
```

   No result is returned from the procedure.

Arguments to the export procedure

- *schemaName*

  Specifies the schema of the table. You can pass a NULL value to use the default schema name.

- *selectStatement*

  Specifies the SELECT statement query that returns the data to be exported. Passing a NULL value will result in an error.

- *tableName*

  Specifies the table name of the table or view from which the data is to be exported. This table cannot be a system table or a declared temporary table. The string must exactly match the case of the table name. Passing a null will result in an error.

- *fileName*

Specifies the file to which the data is to be exported. If the path is omitted, the current working directory is used. If the name of a file that already exists is specified, the export utility overwrites the contents of the file; it does not append the information. The specified location of the file should refer to the server-side location if using the Network Server. Passing a null will result in an error.

- *columnDelimiter*

  Specifies a column delimiter. The specified character is used in place of a comma to signify the end of a column. You can pass a NULL value to use the default value of a comma.

- *characterDelimiter*

  Specifies a character delimiter. The specified character is used in place of double quotation marks to enclose a character string. You can pass a NULL value to use the default value of a double quotation mark.

- *codeset*

  Specifies the code set of the data in the export file. The code set name should be one of the Java-supported character encoding sets. Data is converted from the database code page to the specified code page before writing to the file. You can pass a NULL value to write the data in the same code page as the JVM in which it is being executed.

If you create a schema, table, or column name as a non-delimited identifier, you must pass the name to the export procedure using all upper-case characters. If you created a schema or table name as a delimited identifier, you must pass the name to the export procedure using the same case that was used when it was created.

Examples of bulk import and export

All examples in this section are run using the `ij` utility.

The following example shows how to import data into the *staff* table in a sample database from the `myfile.del` file.

```
CALL SYSCS_UTIL.SYSCS_IMPORT_TABLE (null,'staff','myfile.del',null,null,
null,0);
```

The following example shows how to import data into the *staff* table in a sample database from a delimited data file `myfile.del` . This example defines the percentage character (%) as the string delimiter, and a semicolon as the column delimiter.

```
CALL SYSCS_UTIL.SYSCS_IMPORT_TABLE
(null,'staff','c:\output\myfile.del',';','%',
null,0);
```

The following example shows how to export data from the *staff* table in a sample database to the file `myfile.del` .

```
CALL SYSCS_UTIL.SYSCS_EXPORT_TABLE
(null,'staff','myfile.del',null,null,null);
```

The following example shows how to export employee data in department 20 from the

staff table in a sample database to the file `awards.del`.

```
CALL SYSCS_UTIL.SYSCS_EXPORT_QUERY ('select * from staff where dept=20',
'c:\output\awards.del',null,null,null);
```

The following example shows how to export data from the *staff* table to a delimited data file `myfile.del` with the percentage character (%) as the character delimiter, and a semicolon as the column delimiter from the *staff* table.

```
CALL SYSCS_UTIL.SYSCS_EXPORT_TABLE
(null,'staff','c:\output\myfile.del',';','%',
null);
```

Importing into tables with identity columns

You can use the SYSCS_UTIL.SYSCS_IMPORT_DATA procedure to import data into a table that contains an identity column. If the identity column is defined as GENERATED ALWAYS, an identity value is generated for a table row whenever the corresponding row field in the input file does not contain a value for the identity column. When a corresponding row field in the input file already contains a value for the identity column, the row cannot be inserted into the table and the import operation will fail. To prevent such scenarios, the following examples show how to specify arguments in the SYSCS_UTIL.SYSCS_IMPORT_DATA procedure to ignore data for the identity column from the file, and/or omit the column name from the insert column list.

If the REPLACE option is used during import, Derby resets its internal counter of the last identity value for a column to the initial value defined for the identity column.

Consider the following table that contains an identity column, *c2*:

```
CREATE TABLE tab1 (c1 CHAR(30), c2 INT GENERATED ALWAYS AS IDENTITY, c3
REAL,
 c4 CHAR(1))
```

- Suppose you want to import data into *tab1* from a file `myfile.del` that does not have identity column information and `myfile.del` contains three fields with the following data:

  ```
  Robert,45.2,J
  Mike,76.9,K
  Leo,23.4,I
  ```

  To import data from `myfile.del` into the *tab1* table, explicitly list the column names for *tab1* without the identity column *c2* and execute the SYSCS_UTIL.SYSCS_IMPORT_DATA procedure as follows:

  ```
  CALL SYSCS_UTIL.SYSCS_IMPORT_DATA (NULL, 'TAB1', 'C1,C3,C4' , null,
  'myfile.del',null, null,null,0)
  ```

- Suppose you want import data into *tab1* from a file `empfile.del` that also has identity column information and the file contains three fields with the following data:

  ```
  Robert,1,45.2,J
  Mike,2,23.4,I
  Leo,3,23.4,I
  ```

  To import data from `empfile.del` into the *tab1* table, explicitly specify an insert column list without the identity column *c2* and specify the column indexes without

identity column data and execute the SYSCS_UTIL.SYSCS_IMPORT_DATA procedure as follows:

```
CALL SYSCS_UTIL.SYSCS_IMPORT_DATA (NULL, 'TAB1', 'C1,C3,C4' ,
'1,3,4',
'empfile.del',null, null,null,0)
```

Executing import/export procedures from JDBC

You can execute import and export procedures from a JDBC program. The following code fragment shows how you might call the SYSCS_UTIL.SYSCS_EXPORT_TABLE procedure from Java. In this example, the procedure exports the *staff* table data in the default schema to the `staff.dat` file, using a percentage (%) character to specify a column delimiter.

```
PreparedStatement ps=conn.prepareStatement("CALL
SYSCS_UTIL.SYSCS_EXPORT_TABLE
(?,?,?,?,?,?)");
ps.setString(1,null);
ps.setString(2,"STAFF");
ps.setString(3,"staff.dat");
ps.setString(4,"%");
ps.setString(5,null);
ps.setString(6,null);
ps.execute();
```

File format for input and output

The default file format is a delimited text file with the following characteristics:

- Rows are separated by a new line.
- Fields are separated by a comma (,)
- Character-based fields are delimited with double quotes (")

Before performing import or export operations, you must ensure that the chosen delimiter character is not contained in the data to be imported or exported. If you chose a delimiter character that is part of the data to be imported/exported unexpected errors might occur. The following restrictions apply to column and character delimiters:

- Delimiters are mutually exclusive
- A delimiter cannot be a line-feed character, a carriage return, or a blank space.
- The default decimal point (.) cannot be a character delimiter.

The record delimiter is assumed to be a new-line character. The record delimiter should not be used as any other delimiter.

Character delimiters are permitted with the character-based fields (CHAR, VARCHAR, and LONG VARCHAR) of a file during import, any pair of character delimiters found between the enclosing character delimiters is imported into the database. For example, suppose you have the following character string:

```
"what a ""great"" day!"
```

The preceding character string gets imported into the database as:

```
What a "great" day!
```

During export, the rule applies in reverse. For example, suppose you have the following character string:

```
"The boot has a 3" heel."
```

The preceding character string gets exported to a file as:

```
"The boot has a 3""heel."
```

The following example file shows four rows and four columns in default file format:

```
1,abc,22,def
22,,"a is a zero-length string, b is null"
13,"hello",454,"world"
4,b and c are both null,,
```

The export procedure outputs the following values:

```
1,"abc",22,"def"
22,,"a is a zero-length string, b is null"
13,"hello",454,"world"
4,"b and c are both null",,
```

Treatment of NULLS

In a delimited file, a NULL value is exported as an empty field. The following example shows the export of a four-column row in which the third column is NULL:

```
7,95,,Happy Birthday
```

Import works the same way; an empty field is imported as a NULL value.

CODESET values for import/export

Import and export procedures accept arguments to specify codeset values. You can specify the codeset (character encoding) for import and export procedures to override the system default. The following table contains a sample of character encoding supported by JDK 1.x. To review the complete list of character encodings, refer to your Java documentation.

Table1. Sample character encodings
This table contains sample character encodings supported by JDK1.x.

Character Encoding	Explanation
8859_1	ISO Latin-1
8859_2	ISO Latin-2
8859_7	ISO Latin/Greek
Cp1257	Windows Baltic
Cp1258	Windows Vietnamese
Cp437	PC Original
EUCJIS	Japanese EUC
GB2312	GB2312-80 Simplified Chinese
JIS	JIS
KSC5601	KSC5601 Korean
MacCroatian	Macintosh Croatian
MacCyrillic	Macintosh Cyrillic
SJIS	PC and Windows Japanese
UTF-8	Standard UTF-8

The following example shows how to specify UTF-8 encoding to export from the *staff* table:

```
CALL SYSCS_UTIL.SYSCS_EXPORT_TABLE
(NULL,'STAFF','staff.dat',NULL,NULL,'UTF-8')
```

The following example shows how to specify UTF-8 encoding to import from the *staff* table:

```
CALL SYSCS_UTIL.SYSCS_IMPORT_TABLE
(NULL,'STAFF','staff.dat',NULL,NULL,'UTF-8',0)
```

Storing jar files in a database

`SQLJ.install_jar`, `SQLJ.remove_jar`, and `SQLJ.replace_jar`, are a set of procedures in the SQL schema that allow you to store jar files in the database.

Your jar file has a *physical name* (the name you gave it when you created it) and a *Derby name* (the Derby identifier you give it when you load it into a particular schema). Its Derby name is an *SQL92Identifier*; it can be delimited and must be unique within a schema. A single schema can store more than one jar file.

Adding a Jar File

To add a jar file using SQL syntax:

```
CALL SQL.install_jar('
jarFilePath
', qualifiedJarName, 0)
```

Removing a jar file

To remove a jar file using SQL syntax:

```
CALL SQLJ.remove_jar ('
jarFilePath
', qualifiedJarName, 0)
```

Replacing a jar file

To replace a jar file using SQL syntax:

```
CALL SQLJ.replace_jar('
jarFilePath
', qualifiedJarName, 0)
```

- jarFilePath

  The path and physical name of the jar file to add or use as a replacement. For example:

  d:/todays_build/tours.jar

- qualifiedJarName

  The Derby name of the jar file, qualified by the schema name. Two examples:

  ```
  MYSCHEMA.Sample1 -- a delimited identifier.
  ```

  ```
  MYSCHEMA."Sample2"
  ```

Installing a jar example

- Complete SQL example for installing a jar:

  ```
  CALL SQLJ.install_jar('d:\todays_build\tours.jar',
  'APP."ToursLogic!"', 0);
  ```

For more information about storing classes in a database, see the *Derby Developer's Guide* .

sysinfo

Use the sysinfo utility to display information about your Java environment and Derby (including version information). To use sysinfo, either *derby.jar* or *derbytools .jar* must be in your classpath.

```
java org.apache.derby.tools.sysinfo
```

sysinfo example

```
$ java org.apache.derby.tools.sysinfo
------------------ Java Information ------------------
Java Version: 1.4.2_07
Java Vendor: Sun Microsystems Inc.
Java home: c:\p4main\jdk142\jre
Java classpath: c:\Derby_10\lib\derby.jar;c:\Derby_10\lib\derbytools.jar;
c:\Derby_10\lib\derbyLocale_de_DE.jar;e:\Derby_10\lib\derbyLocale_es.jar;
e:\Derby_10\lib\derbyLocale_fr.jar;c:\Derby_10\lib\derbyLocale_it.jar;
e:\Derby_10\lib\derbyLocale_ja_JP.jar;e:\Derby_10\lib\derbyLocale_ko_KR.jar;
e:\Derby_10\lib\derbyLocale_pt_BR.jar;e:\Derby_10\lib\derbyLocale_zh_CN.jar;
e:\Derby_10\lib\derbyLocale_zh_TW.jar
OS name: Windows 2000
OS architecture: x86
OS version: 5.0
Java user name: user1
Java user home: C:\Documents and Settings\myhome
Java user dir: E:\p4main\systest\myrtst7
java.specification.name: Java Platform API Specification
java.specification.version: 1.4
--------- Derby Information --------
JRE - JDBC: J2SE 1.4.2 - JDBC 3.0
[c:\Derby_10\lib\derby.jar] 10.1.1.0 - (190628)
[c:\Derby_10\lib\derbytools.jar] 10.1.1.0 - (190628)
------------------------------------------------------
----------------- Locale Information -----------------
Current Locale : [English/United States [en_US]]
Found support for locale: [de_DE]
         version: 10.1.1.0 - (190628)
Found support for locale: [es]
         version: 10.1.1.0 - (190628)
Found support for locale: [fr]
         version: 10.1.1.0 - (190628)
Found support for locale: [it]
         version: 10.1.1.0 - (190628)
Found support for locale: [ja_JP]
         version: 10.1.1.0 - (190628)
Found support for locale: [ko_KR]
         version: 10.1.1.0 - (190628)
Found support for locale: [pt_BR]
         version: 10.1.1.0 - (190628)
Found support for locale: [zh_CN]
         version: 10.1.1.0 - (190628)
Found support for locale: [zh_TW]
```

When requesting help from Derby technical support or posting on the forum, include a copy of the information provided by sysinfo.

Using sysinfo to check the classpath

sysinfo provides an argument (*-cp*) which can be used to test the classpath.

```
java org.apache.derby.tools.sysinfo -cp
[ [ embedded ][ server ][ client] [ tools ] [ anyClass.class ] ]
```

If your environment is set up correctly, the utility shows output indicating success.

You can provide optional arguments with *-cp* to test different environments. Optional arguments to *-cp* are:

- embedded
- server
- client
- tools
- *classname*.class

If something is missing from your classpath, the utility indicates what is missing. For example, if you neglected to include the directory containing the class named *SimpleApp* to your classpath, the utility would indicate this when the following command line was issued (type all on one line):

```
$ java org.apache.derby.tools.sysinfo -cp embedded SimpleApp.class
FOUND IN CLASS PATH:

Derby embedded engine library (derby.jar)


NOT FOUND IN CLASS PATH:

user-specified class (SimpleApp)
(SimpleApp not found.)
```

dblook

Use the `dblook` utility to view all or parts of the Data Definition Language (DDL) for a given database. You must place the Derby `derbytools.jar` file in the classpath directory to use the `dblook` utility.

Using dblook

The syntax for the command to launch the `dblook` utility is:

```
java org.apache.derby.tools.dblook -d <databaseURL> [OPTIONS]
```

The value for *databaseUrl* is the complete URL for the database. Where appropriate, the URL includes any connection attributes or properties that might be required to access the database.

For example, to connect to the database 'myDB', the URL would simply be `'jdbc:derby:myDB';` to connect using the Network Server to a database 'C:\private\tmp\myDB' on a remote server (port 1527), the URL would be:

```
'jdbc:derby://localhost:1527/"C:\private\tmp\myDB"
;user=someusr;password=somepwd'
```

As with other Derby utilities, you must ensure that no other JVMs are started against the database when you call the `dblook` utility, or an exception will occur and will print to the `dblook.log`. If this exception is thrown, the `dblook` utility will quit. To recover, you must ensure that no other Derby applications running in a separate JVM are connected to the source database. These connections need to be shutdown. Once all existing JVMs running against the database have been shutdown, the `dblook` utility will execute successfully. You can also start the Derby Network server and run the `dblook` utility as a client application while other clients are connected to the server.

dblook options

The `dblook` utility options include:

-z <schemaName>

specifies the schema to which the DDL should be restricted. Only objects with the specified schema are included in the DDL file.

-t <tableOne> <tableTwo> ...

specifies the tables to which the DDL should be restricted. All tables with a name from this list will be included in the DDL file subject to `-z` limitations, as will the DDL for any keys, checks, or indexes on which the table definitions depend.

Additionally, if the statement text of any triggers or views includes a reference to any of the listed table names, the DDL for that trigger/view will also be generated, subject to `-z` limitations. If a table is not included in this list, then neither the table nor any of its keys, checks, or indexes will be included in the final DDL. If this parameter is not provided, all database objects will be generated, subject to `-z` limitations. Table names are separated by whitespace.

-td

specifies a statement delimiter for SQL statements generated by dblook. If a statement delimiter option is not specified, the default is the semicolon (';'). At the end of each DDL statement, the delimiter is printed, followed by a new line.

-o <filename>
specifies the file where the generated DDL is written. If this file is not specified, it defaults to the console (i.e. standard System.out).

-append
prevents overwriting the DDL output ('-o' parameter, if specified) and "dblook.log" files. If this parameter is specified, and execution of the `dblook` command leads to the creation of files with names identical to existing files in the current directory, `dblook` will append to the existing files. If this parameter is not set, the existing files will be overridden.

-verbose
specifies that all errors and warnings (both SQL and internal to `dblook`) should be echoed to the console (via System.err), in addition to being printed to the "dblook.log" file. If this parameter is not set, the errors and warnings only go to the "dblook.log" file.

-noview
specifies that CREATE VIEW statements should not be generated.

Generating the DDL for a database

The `dblook` utility generates all of the following objects when generating the DDL for a database:

- Checks
- Functions
- Indexes
- Jar files
- Keys (primary, foreign, and unique)
- Schemas
- Stored procedures
- Triggers
- Tables
- Views

Note: When dblook runs against a database that has jar files installed, it will create a new directory, called `DERBYJARS`, within the current directory, and that is where it will keep copies of all of the jars it encounters. In order to run the generated DDL as a script, this `DERBYJARS` directory must either 1) exist within the directory in which it was created, or 2) be moved manually to another directory, in which case the path in the generated DDL file must be manually changed to reflect to the new location.

The `dblook` utility ignores any objects that have system schemas (for example, SYS, SYSIBM), since DDL is not able to directly create nor modify system objects.

dblook examples

The following examples demonstrate how the various `dblook` utility options might be specified from a command line. Each example is preceded by a brief description of how the `dblook` utility behaves given the specified options.

```
Dump the DDL for everything in database 'sample' (in the current
directory)
to the console.
java org.apache.derby.tools.dblook -d jdbc:derby:sample


Dump the DDL for everything in database 'sample' (in the current
directory)
to the console,
and include error/warning messages (with the latter being printed via
System.err)
java org.apache.derby.tools.dblook
   -d jdbc:derby:sample -verbose
```

```
Dump the DDL for everything in database 'sample' (in the current
directory)
to a file called "myDB_DDL.sql" (in the current directory).
java org.apache.derby.tools.dblook -d jdbc:derby:sample -o myDB_DDL.sql


Dump the DDL for everything in database 'sample' (in a specified
directory)
to a file called "newDB.sql" (in a specified directory).
java org.apache.derby.tools.dblook -d
'jdbc:derby:c:\private\stuff\sample'
        -o 'C:\temp\newDB.sql'


Dump the DDL for all objects in database 'sample' with schema 'SAMP'
to the console.
java org.apache.derby.tools.dblook -d jdbc:derby:sample -z samp

Dump the DDL for all objects in remote database 'sample' on
'localhost:1527'
with schema 'SAMP' to the console.
java org.apache.derby.tools.dblook -d
        'jdbc:derby://localhost:1527/"C:\temp\sample";
        user=someusr;password=somepwd' -z samp


Dump the DDL for all objects with schema 'SAMP' in database 'sample'
that are related to the table name 'My Table' to the console.
java org.apache.derby.tools.dblook -d jdbc:derby:sample -z samp -t "My
Table"


Dump the DDL for all objects in database 'sample' that are related
to either of the table names 'STAFF' or 'My Table' to the console.
java org.apache.derby.tools.dblook -d jdbc:derby:sample -t "My Table"
staff


Dump the DDL for all objects in database 'sample' to myDB_DDL.sql,
with no statement delimiter (i.e. leave off the default semi-colon),
and append to the output files if they are already there.
java org.apache.derby.tools.dblook -d jdbc:derby:sample
        -o myDB_DDL.sql -td '' -append


Dump the DDL for all objects EXCEPT views in database 'sample'
to the console.
java org.apache.derby.tools.dblook -d jdbc:derby:sample -noview
```

Status messages are printed to the output (either a -o filename, if specified, or the console) as SQL script comments. These status messages serve as headers to show which types of database objects are being (or have been) processed by the dblook utility.

Trademarks

The following terms are trademarks or registered trademarks of other companies and have been used in at least one of the documents in the Apache Derby documentation library:

Cloudscape, DB2, DB2 Universal Database, DRDA, and IBM are trademarks of International Business Machines Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.