



**Faircom**<sup>®</sup>

c-tree **Plus**<sup>®</sup>  
**V9**

**c-treeSQL**

Java Stored Procedures,  
Triggers and User Defined  
Functions Guide



# **c-treeSQL**

Java Stored Procedures, Triggers and User Defined Functions  
Guide



Copyright © 1992-2008 FairCom Corporation All rights reserved. No part of this publication may be stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior written permission of FairCom Corporation. Printed in the United States of America.

Information in this document is subject to change without notice.

## Trademarks

c-tree, c-tree Plus, r-tree, the circular disk logo, and FairCom are registered trademarks of the FairCom Corporation. c-treeACE SQL, c-treeACE SQL ODBC, c-treeACE SQL ODBC SDK, c-treeVCL/CLX, c-tree ODBC Driver, c-tree Crystal Reports Driver, c-treeDBX, and c-treePHP are trademarks of FairCom Corporation. The following are third-party trademarks: AMD and AMD Opteron are trademarks of Advanced Micro Devices, Inc. Macintosh, Mac OS, and Xcode are trademarks of Apple Inc., registered in the U.S. and other countries. Borland, the Borland Logo, Delphi, C#Builder, C++Builder, Kylix, and CLX are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. Business Objects, the Business Objects logo, Crystal Reports, and Crystal Enterprise are trademarks or registered trademarks of Business Objects SA or its affiliated companies in the United States and other countries. DBstore is a trademark of Dharma Systems, Inc. HP and HP-UX are registered trademarks of the Hewlett-Packard Company. AIX, IBM, OS/2, OS/2 WARP, and POWER5 are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. Intel, Itanium, Pentium and Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. LynuxWorks, LynxOS and BlueCat are registered trademarks of LynuxWorks, Inc. Microsoft, the .NET logo, MS-DOS, Visual Studio, Windows, Windows Mobile, Windows server and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Novell and NetWare are registered trademarks of Novell, Inc., in the United States and other countries. QNX and Neutrino are registered trademarks of QNX Software Systems Ltd. in certain jurisdictions. Red Hat and the Shadow Man logo are registered trademarks of Red Hat, Inc. in the United States and other countries, used with permission. SCO and SCO Open Server, are trademarks or registered trademarks of The SCO Group, Inc. in the U.S.A. and other countries. SGI and IRIX are registered trademarks of Silicon Graphics, Inc., in the United States and/or other countries worldwide. Sun, Sun Microsystems, the Sun Logo, Solaris, SunOS, JDBC, Java and all Java-based trademarks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. UNIX and UnixWare are registered trademarks of The Open Group in the United States and other countries. Linux is a trademark of Linus Torvalds in the United States, other countries, or both. All other trademarks, trade names, company names, product names, and registered trademarks are the property of their respective holders.

FairCom welcomes your comments on this document and the software it describes. Send comments to:

Documentation Comments  
FairCom Corporation  
6300 W. Sugar Creek Drive  
Columbia, MO 65203

Portions © 1987-2008 Dharma Systems, Inc. All rights reserved. This software or web site utilizes or contains material that is © 1994-2007 DUNDAS DATA VISUALIZATION, INC. and its licensors, all rights reserved.

6/26/2008

# CONTENTS

---

<b>Introduction to Stored Procedures and Triggers and User Defined Functions .....</b>	<b>1</b>
1.1 Overview .....	1
1.2 Advantages of Stored Procedures .....	1
1.3 How c-treeSQL Interacts with Java .....	2
Creating Stored Procedures .....	2
Calling Stored Procedures .....	3
<hr/>	
<b>Quick Tour .....</b>	<b>5</b>
2.1 Introductory Tutorial .....	5
Init .....	6
Define .....	7
Manage .....	8
Done .....	10
Additional Resources .....	11
2.2 Relationships .....	12
Init .....	14
Define .....	15
Manage .....	18
Done .....	21
Additional Resources .....	22
2.3 Record/Row Locking .....	23
Init .....	24
Define .....	25
Manage .....	26
Done .....	28
Additional Resources .....	29
2.4 Transaction Processing .....	30
Init .....	31
Define .....	32
Manage .....	35
Done .....	38
Additional Resources .....	39
<hr/>	
<b>Using Stored Procedures .....</b>	<b>41</b>
3.1 Introduction .....	41
3.2 Stored Procedure Basics .....	42
What Is a Java Snippet? .....	42
Structure of Stored Procedures .....	42
Setting Up Your Environment to Write Stored Procedures .....	44
Writing Stored Procedures .....	45
Invoking Stored Procedures .....	45
Modifying and Deleting Stored Procedures .....	47
Debugging Stored Procedures .....	47
Transactions and Stored Procedures .....	48
Stored Procedure Security .....	48
Restrictions on Calling Java Methods in Stored Procedures .....	48
3.3 Using the c-treeSQL Java Classes .....	48
Passing Values to SQL Statements .....	49
Passing Values to and From Stored Procedures: Input and Output Parameters .....	50
Implicit Data Type Conversion Between SQL and Java Types .....	51
Executing an SQL Statement .....	52

---

Retrieving Data: the SQLCursor Class .....	54
Returning a Procedure Result Set: the RESULT Clause and DhSQLResultSet .....	57
Handling Null Values .....	58
Handling Errors .....	59
Calling Stored Procedures from Stored Procedures .....	60

---

<b>Using Triggers .....</b>	<b>63</b>
4.1 Introduction .....	63
4.2 Trigger Basics .....	63
Structure of Triggers .....	63
Triggers vs. Stored Procedures vs. Constraints .....	65
Typical Uses for Triggers .....	66
4.3 OLDROW and NEWROW Objects: Passing Values to Triggers .....	66
Restrictions on creating Triggers .....	67

---

<b>Using User Defined Scalar Functions .....</b>	<b>69</b>
5.1 Introduction .....	69
5.2 Create Function .....	69
5.3 Description .....	70
5.4 Invoking User Defined Scalar Functions .....	70
With Constants .....	71
With Constants and Column References .....	71
With parameter reference (ODBC/JDBC) .....	71
5.5 Drop Function .....	71
5.6 User Defined Scalar Function Security .....	71
5.7 Calling Scalar Functions from a User Defined Scalar Function .....	72

---

<b>Java Class Reference .....</b>	<b>73</b>
6.1 Introduction .....	73
6.2 DhSQLException .....	74
DhSQLException.getDiagnostics .....	75
6.3 DhSQLResultSet .....	76
DhSQLResultSet.insert .....	76
DhSQLResultSet.makeNULL .....	77
DhSQLResultSet.set .....	78
6.4 SQLCursor .....	79
SQLCursor.close .....	80
SQLCursor.fetch .....	80
SQLCursor.found .....	81
SQLCursor.getParam .....	82
SQLCursor.getValue .....	83
SQLCursor.makeNULL .....	84
SQLCursor.open .....	84
SQLCursor.registerOutParam .....	85
SQLCursor.rowCount .....	86
SQLCursor.setParam .....	86
SQLCursor.wasNULL .....	87
6.5 SQLStatement .....	88
SQLStatement.execute .....	89
SQLStatement.getParam .....	89
SQLStatement.makeNULL .....	90
SQLStatement.registerOutParam .....	91
SQLStatement.rowCount .....	92
SQLStatement.setParam .....	92
6.6 SQLPStatement .....	93

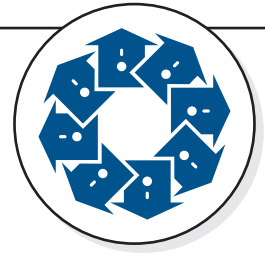
SQLPStatement.execute .....	94
SQLPStatement.getParam .....	94
SQLPStatement.makeNULL .....	95
SQLPStatement.registerOutParam .....	96
SQLPStatement.rowCount.....	97
SQLPStatement.setParam.....	97
<hr/>	
<b>Glossary .....</b>	<b>99</b>
<b>Index.....</b>	<b>107</b>

# FAIRCOM TYPOGRAPHICAL CONVENTIONS

Before you begin using this guide, be sure to review the relevant terms and typographical conventions used in the documentation.

The following formatted items identify special information.

Formatting convention	Type of Information
<b>Bold</b>	Used to emphasize a point or for variable expressions such as parameters.
CAPITALS	Names of keys on the keyboard. For example, SHIFT, CTRL, or ALT+F4.
<i>FairCom Terminology</i>	FairCom technology term.
<b>FunctionName()</b>	c-tree Function name.
<i>Parameter</i>	c-tree Function Parameter.
Code Examples	Code example or Command line usage.
<b>utility</b>	c-tree executable or utility.
<i>filename</i>	c-tree file or path name.
CONFIGURATION KEYWORDS	c-treeACE Configuration Keyword.
<b>BIG_ERR</b>	c-tree Error Code.



# Introduction to Stored Procedures and Triggers and User Defined Functions

## 1.1 Overview

Stored procedures and triggers provides the ability to write Java routines that contain SQL statements and store those routines with a database under the c-treeSQL Server - Java Edition. Tools and applications can then execute the procedures.

A stored procedure is a snippet of Java code embedded in an SQL `CREATE PROCEDURE` statement. The Java snippet can use all standard Java features as well as use c-treeSQL - supplied Java classes for processing any number of SQL statements.

A trigger is a special type of stored procedure that helps ensure referential integrity for a database. Like stored procedures, triggers also contain Java code (embedded in a `CREATE TRIGGER` statement) and use c-treeSQL Java classes. However, triggers are automatically invoked ("fired") by certain SQL operations (an insert, update, or delete operation) on the trigger's target table.

User defined functions extend the availability of built-in scalar functions with modules written by the user with the Java language. The user can transform data with custom routines to suit their unique business needs. These Java procedures are created with the c-treeSQL `CREATE FUNCTION` statement.

## 1.2 Advantages of Stored Procedures

Stored procedures and triggers provide a flexible, general mechanism to store a collection of SQL statements and Java program constructs in a database enforcing business rules and performing administrative tasks.

The ability to write stored procedures and triggers expands the flexibility and performance of applications that access a c-treeSQL environment:

- In a client/server environment, applications make only a single client/server request for the entire procedure, instead of one or more requests for each c-treeSQL statement in the stored procedure or trigger.
- Stored procedures and triggers are stored in compiled form (as well as source-code form), so execute much faster than a corresponding c-treeSQL script.

- Stored procedures can implement elaborate algorithms to enforce complex business rules. The details of the procedure implementation can change without requiring changes in an application that calls the procedure.

## 1.3 How c-treeSQL Interacts with Java

c-treeSQL's implementation of stored procedures allows use of standard Java programming constructs instead of requiring proprietary flow-control language. To do this, the c-treeSQL Server interacts with Java in the following ways:

- When you create a stored procedure, the c-treeSQL Server processes the Java code, submits it to the Java compiler, and receives the compiled results to store in the database.
- When applications call a stored procedure, the c-treeSQL Server interacts with the Java virtual machine to execute the stored procedure and receive any results.

### Creating Stored Procedures

The Java source code that makes up the body of a stored procedure is not a complete program, but a snippet that c-treeSQL converts to a Java class when it processes a `CREATE PROCEDURE` statement.

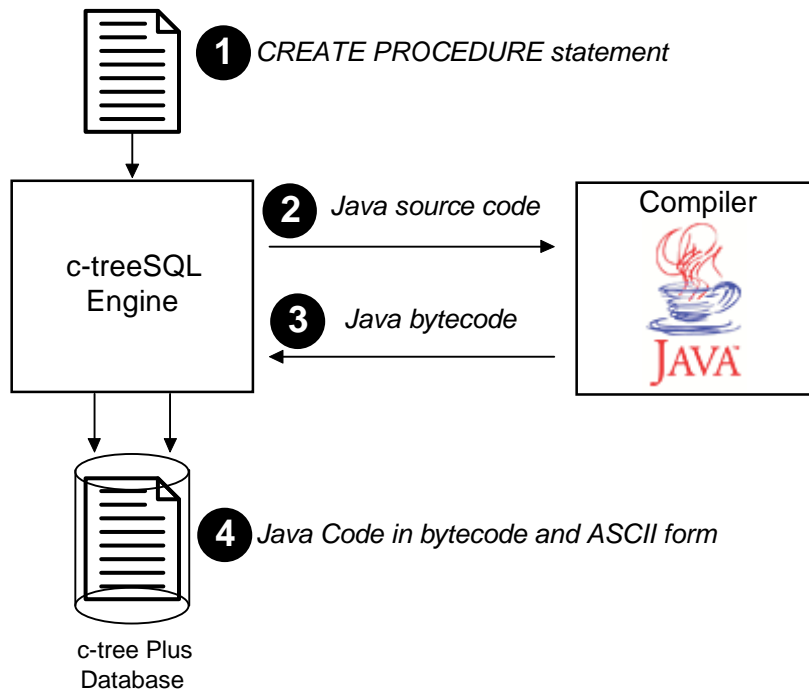
Creating a stored procedure involves the following steps:

1. Some application or tool (interactive SQL, a c-treeSQL script, or an application) issues a `CREATE PROCEDURE` statement containing a Java snippet.
2. c-treeSQL adds code to the Java snippet to create a complete Java class and submits the combined code to the Java compiler.
3. Presuming there are no Java compilation errors, the Java compiler sends compiled bytecode back to c-treeSQL. If there are compilation errors, c-treeSQL passes the first error generated back to the application or tool that issued the `CREATE PROCEDURE` statement.
4. c-treeSQL stores both the Java source code and the bytecode form of the procedure in the database.

The following figure illustrates the steps in creating a stored procedure.

Figure: Creating Stored Procedures

Figure 1: Creating Stored Procedures



## Calling Stored Procedures

Once a stored procedure is created and stored in the database, any application (or other stored procedure) can execute it by calling it. You can call stored procedures from ODBC applications, JDBC applications, .NET applications, or directly from interactive SQL.

For instance, the following example shows an excerpt from an ODBC application that calls a stored procedure (*order\_parts*) using the ODBC syntax **{ call procedure\_name ( param ) }**.

### Executing a Stored Procedure Through ODBC

```
SQLINTEGER Part_num;
SQLINTEGER Part_numInd = 0;

// Bind the parameter.
SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER, 0, 0,
                  &Part_num, 0, Part_numInd);

// Place the department number in Part_num.
Part_num = 318;

// Execute the statement.
SQLExecDirect (hstmt, "{call order_parts(?)}", SQL_NTS);
```

Executing a stored procedure involves the following steps:

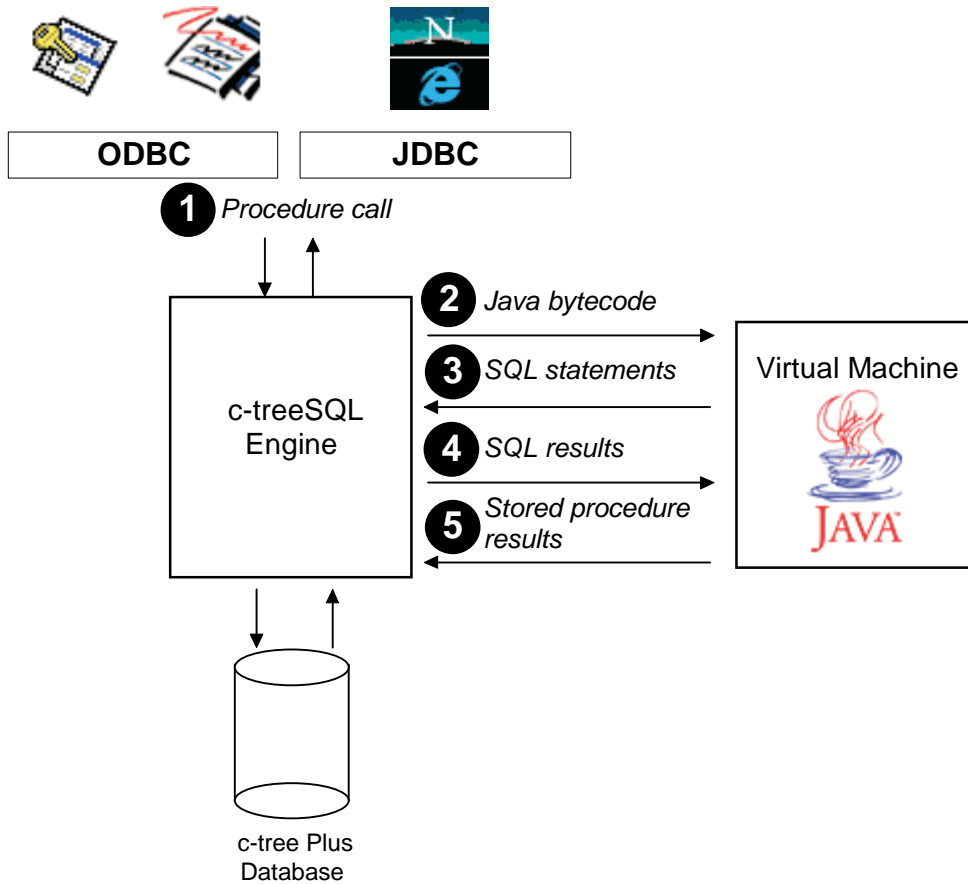
1. The application calls the stored procedure through its native calling mechanism (Example, for instance, used the ODBC call escape sequence).

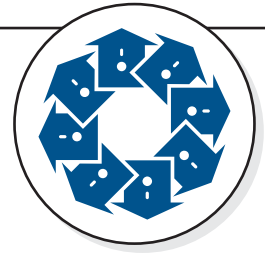
2. c-treeSQL retrieves the compiled bytecode-form of the procedure and submits it to the Java virtual machine for execution.
3. For every c-treeSQL statement in the procedure, the Java virtual machine calls c-treeSQL.
4. c-treeSQL manages interaction with the underlying database system and execution of the SQL statements, and returns any results to the Java virtual machine.
5. The Java virtual machine returns results (output parameters and result sets) of the procedure to c-treeSQL, which in turn passes them to the calling application.

The following figure illustrates the steps in executing a stored procedure.

Figure: Executing Stored Procedures

**Figure 2: Executing Stored Procedures**





## Quick Tour

### 2.1 Introductory Tutorial

```
..\sdk\sql.stored.procs\tutorials\SPTTutorial1.sql
```

This tutorial will take you through the basic use of the c-treeACE SQL Stored Procedures Technology.

Like all other examples in the c-tree tutorial series, this tutorial simplifies the creation and use of a database into four simple steps: Initialize(), Define(), Manage(), and You're Done() !

#### Tutorial #1: Introductory - Simple Single Table

We wanted to keep this program as simple as possible. This program does the following:

- Initialize() - Connects to the c-treeACE Database Engine.
- Define() - Defines and creates a "customer master" (custmast) table/file.
- Manage() - Adds a few rows/records; Reads the rows/records back from the database; displays the column/field content; and then deletes the rows/records.
- Done() - Disconnects from c-treeACE Database Engine.

Note these sections in our SQL Script:

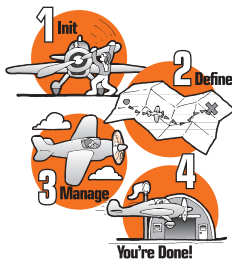
```
-- Initialize
ECHO INIT;

-- Define
ECHO DEFINE;

-- Manage
ECHO MANAGE;

-- Done
ECHO DONE;
```

We suggest opening the source code with your own editor.



Continue now to review these four steps.

## Init



First we need to open a connection to a database by providing the c-treeACE Database Engine with a user name, password and the database name.

Below is the code for **Initialize()**:

```
SET ECHO OFF

-- Initialize
ECHO INIT;

SET AUTOCOMMIT ON;
```

## Define



The Define() step is where specific data definitions are established by your application and/or process. This involves defining columns/fields and creating the tables/files with optional indices.

Below is the code for **Define()**:

```
DROP PROCEDURE Define;

CREATE PROCEDURE Define()
BEGIN
    SQLStatement st = new SQLStatement (
        "CREATE TABLE custmast (" +
            "cm_custnumb CHAR(4), " +
            "cm_custzipc CHAR(9), " +
            "cm_custstat CHAR(2), " +
            "cm_custrtng CHAR(1), " +
            "cm_custname VARCHAR(47), " +
            "cm_custaddr VARCHAR(47), " +
            "cm_custcity VARCHAR(47))"
    );
    st.execute();
END

-- Define
ECHO DEFINE;
CALL Define();
```

## Manage



The manage step provides data management functionality for your application and/or process.

Below is the code for **Manage()**:

```
DROP PROCEDURE Add_Records;
DROP PROCEDURE Display_Records;
DROP PROCEDURE Delete_Records;

CREATE PROCEDURE Add_Records (
    IN cm_custnumb CHAR(4),
    IN cm_custzipc CHAR(9),
    IN cm_custstat CHAR(2),
    IN cm_custrtnng CHAR(1),
    IN cm_custname VARCHAR(47),
    IN cm_custaddr VARCHAR(47),
    IN cm_custcity VARCHAR(47)
)
BEGIN
    SQLStatement st = new SQLStatement (
        "INSERT INTO custmast VALUES (?, ?, ?, ?, ?, ?, ?, ?) "
    );
    st.setParam (1, cm_custnumb);
    st.setParam (2, cm_custzipc);
    st.setParam (3, cm_custstat);
    st.setParam (4, cm_custrtnng);
    st.setParam (5, cm_custname);
    st.setParam (6, cm_custaddr);
    st.setParam (7, cm_custcity);
    st.execute();
END

CREATE PROCEDURE Display_Records ()
RESULT (
    Numb CHAR(4),
    Name CHAR(47)
)
BEGIN
    SQLCursor cur = new SQLCursor ("SELECT cm_custnumb, cm_custname FROM custmast");
    cur.open();
    cur.fetch();
    while (cur.found())
    {
        SQLResultSet.set(1, cur.getValue(1, CHAR));
        SQLResultSet.set(2, cur.getValue(2, CHAR));

        SQLResultSet.insert();
        cur.fetch();
    }
    cur.close();
END

CREATE PROCEDURE Delete_Records ()
BEGIN
    SQLStatement sp_DeleteTable = new SQLStatement ("DELETE FROM custmast");
    sp_DeleteTable.execute();
END
```

```
END

-- Manage
ECHO MANAGE;

ECHO Delete records...;
CALL Delete_Records ();

ECHO Add records...;
CALL Add_Records('1000', '92867', 'CA', '1', 'Bryan Williams', '2999 Regency', 'Orange');
CALL Add_Records('1001', '61434', 'CT', '1', 'Michael Jordan', '13 Main', 'Harford');
CALL Add_Records('1002', '73677', 'GA', '1', 'Joshua Brown', '4356 Cambridge', 'Atlanta');
CALL Add_Records('1003', '10034', 'MO', '1', 'Keyon Dooling', '19771 Park Avenue', 'Columbia');

ECHO Display records...;
CALL Display_Records();

ECHO Delete records...;
CALL Delete_Records ();
```

## Done

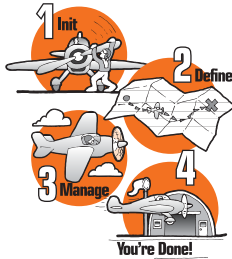


When an application and/or process has completed operations with the database, it must release resources by disconnecting from the database engine.

Below is the code for **Done()**:

```
-- Done  
ECHO DONE;
```

## Additional Resources



We encourage you to explore the additional resources listed here:

- Complete SQL script for this tutorial can be found in SPTTutorial1.sql in your installation directory, within the 'sdk\sql.stored.procs\tutorials' directory for your platform.  
Example for the Windows platform:  
C:\FairCom\V9.0.0\win32\sdk\sql.stored.procs\tutorials\SPTTutorial1.sql.
- Additional documentation may be found on the FairCom Web site at: [www.faircom.com](http://www.faircom.com)

## 2.2 Relationships

..\sdk\sql.stored.procs\tutorials\SPTTutorial2.sql

Now we will build some table/file relationships using the c-treeACE SQL Stored Procedures Technology.

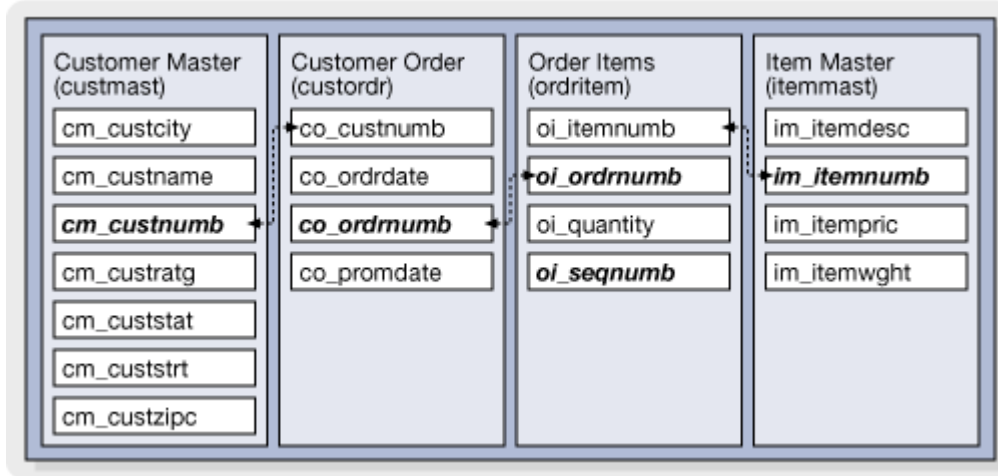
This tutorial will advance the concepts introduced in the first tutorial by expanding the number of tables. We will define key columns/fields and create specific indices for each table to form a relational model database.

Like all other examples in the c-tree tutorial series, this tutorial simplifies the creation and use of a database into four simple steps: Initialize(), Define(), Manage(), and You're Done() !

### Tutorial #2: Relational Model and Indexing

Here we add a bit more complexity, introducing multiple tables, with related indices in order to form a simple "relational" database simulating an Order Entry system. Here is an overview of what will be created:

#### Relational Model Tables



- Initialize() - Connects to the c-treeACE Database Engine.
- Define() - Defines and creates the "custmast", "custordr", "ordritem" and the "itemmast" tables/files with related indices.
- Manage() - Adds some related rows/records to all tables/files. Then queries the database.
- Done() - Disconnects from c-treeACE Database Engine.

Note these sections in our SQL Script:

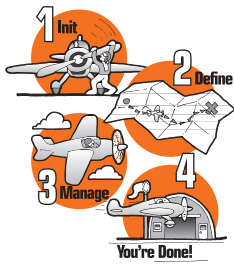
```
-- Initialize
ECHO INIT;

-- Define
ECHO DEFINE;

-- Manage
ECHO MANAGE;

-- Done
ECHO DONE;
```

We suggest opening the source code with your own editor.



Continue now to review these four steps.

## Init



First we need to open a connection to a database by providing the c-treeACE Database Engine with a user name, password and the database name.

Below is the code for **Initialize()**:

```
SET ECHO OFF

-- Initialize
ECHO INIT;

SET AUTOCOMMIT ON;
```

## Define



The Define() step is where specific data definitions are established by your application and/or process. This involves defining columns/fields and creating the tables/files with optional indices.

Below is the code for **Define()**:

```
DROP PROCEDURE Define;

CREATE PROCEDURE Define()
BEGIN
    int TABLE_ALREADY_EXIST = -20041;
    int INDEX_ALREADY_EXIST = -20028;

    try
    {
        SQLStatement cm = new SQLStatement (
            "CREATE TABLE custmast (" +
                "cm_custnumb CHAR(4), " +
                "cm_custzipc CHAR(9), " +
                "cm_custstat CHAR(2), " +
                "cm_custrtng CHAR(1), " +
                "cm_custname VARCHAR(47), " +
                "cm_custaddr VARCHAR(47), " +
                "cm_custcity VARCHAR(47)"
            );
        cm.execute();
    }
    catch (DhSQLException e)
    {
        if (e.sqlErr != TABLE_ALREADY_EXIST)
            throw e;
    }
    try
    {
        SQLStatement cm1 = new SQLStatement (
            "CREATE UNIQUE INDEX cm_custnumb_idx ON custmast (cm_custnumb)"
        );
        cm1.execute();
    }
    catch (DhSQLException e)
    {
        if (e.sqlErr != INDEX_ALREADY_EXIST)
            throw e;
    }
    try
    {
        SQLStatement co = new SQLStatement (
            "CREATE TABLE custordr (" +
                "co_ordrdate DATE, " +
                "co_promdate DATE, " +
                "co_ordrnumb CHAR(6), " +
                "co_custnumb CHAR(4)"
            );
        co.execute();
    }
}
```

```
catch (DhSQLException e)
{
    if (e.sqlErr != TABLE_ALREADY_EXIST)
        throw e;
}
try
{
    SQLIStatement col = new SQLIStatement (
        "CREATE UNIQUE INDEX co_ordrnumb_idx ON custordr (co_ordrnumb)"
    );
    col.execute();
}
catch (DhSQLException e)
{
    if (e.sqlErr != INDEX_ALREADY_EXIST)
        throw e;
}
try
{
    SQLIStatement co2 = new SQLIStatement (
        "CREATE INDEX co_custnumb_idx ON custordr (co_custnumb)"
    );
    co2.execute();
}
catch (DhSQLException e)
{
    if (e.sqlErr != INDEX_ALREADY_EXIST)
        throw e;
}

try
{
    SQLIStatement oi = new SQLIStatement (
        "CREATE TABLE ordritem (" +
        "oi_seqnumb SMALLINT, " +
        "oi_quantity SMALLINT, " +
        "oi_ordrnumb CHAR(6), " +
        "oi_itemnumb CHAR(5))"
    );
    oi.execute();
}
catch (DhSQLException e)
{
    if (e.sqlErr != TABLE_ALREADY_EXIST)
        throw e;
}
try
{
    SQLIStatement oil = new SQLIStatement (
        "CREATE UNIQUE INDEX oi_ordrnumb_idx ON ordritem (oi_ordrnumb, oi_seqnumb)"
    );
    oil.execute();
}
catch (DhSQLException e)
{
    if (e.sqlErr != INDEX_ALREADY_EXIST)
        throw e;
}
try
{
    SQLIStatement oi2 = new SQLIStatement (
        "CREATE INDEX oi_itemnumb_idx ON ordritem (oi_itemnumb)"
    );
    oi2.execute();
}
catch (DhSQLException e)
{
    if (e.sqlErr != INDEX_ALREADY_EXIST)
        throw e;
}
```

```
try
{
    SQLStatement im = new SQLStatement (
        "CREATE TABLE itemmast (" +
            "im_itemwght INTEGER, " +
            "im_itempric MONEY, " +
            "im_itemnumb CHAR(5), " +
            "im_itemdesc VARCHAR(47))"
    );
    im.execute();
}
catch (DhSQLException e)
{
    if (e.sqlErr != TABLE_ALREADY_EXIST)
        throw e;
}
try
{
    SQLStatement im1 = new SQLStatement (
        "CREATE UNIQUE INDEX im_itemnumb_idx ON itemmast (im_itemnumb)"
    );
    im1.execute();
}
catch (DhSQLException e)
{
    if (e.sqlErr != INDEX_ALREADY_EXIST)
        throw e;
}
END

-- Define
ECHO DEFINE;

ECHO Create tables...;
CALL Define();
```

## Manage



The manage step provides data management functionality for your application and/or process.

Below is the code for **Manage()**:

```
DROP PROCEDURE Add_CustomerMaster_Record;
DROP PROCEDURE Add_CustomerOrders_Record;
DROP PROCEDURE Add_OrderItems_Record;
DROP PROCEDURE Add_ItemMaster_Record;
DROP PROCEDURE Delete_Records;
DROP PROCEDURE Display_Records;

CREATE PROCEDURE Add_CustomerMaster_Record (
    IN cm_custnumb CHAR(4),
    IN cm_custzipc CHAR(9),
    IN cm_custstat CHAR(2),
    IN cm_custrtng CHAR(1),
    IN cm_custname VARCHAR(47),
    IN cm_custaddr VARCHAR(47),
    IN cm_custcity VARCHAR(47)
)
BEGIN
    SQLIStatement st = new SQLIStatement (
        "INSERT INTO custmast VALUES (?, ?, ?, ?, ?, ?, ?)"
    );
    st.setParam (1, cm_custnumb);
    st.setParam (2, cm_custzipc);
    st.setParam (3, cm_custstat);
    st.setParam (4, cm_custrtng);
    st.setParam (5, cm_custname);
    st.setParam (6, cm_custaddr);
    st.setParam (7, cm_custcity);
    st.execute();
END

CREATE PROCEDURE Add_CustomerOrders_Record (
    IN co_ordrdate DATE,
    IN co_promdate DATE,
    IN co_ordrnumb CHAR(6),
    IN co_custnumb CHAR(4)
)
BEGIN
    SQLIStatement st = new SQLIStatement (
        "INSERT INTO custordr VALUES (?, ?, ?, ?)"
    );
    st.setParam (1, co_ordrdate);
    st.setParam (2, co_promdate);
    st.setParam (3, co_ordrnumb);
    st.setParam (4, co_custnumb);
    st.execute();
END

CREATE PROCEDURE Add_OrderItems_Record (
    IN oi_sequnumb SMALLINT,
    IN oi_quantity SMALLINT,
    IN oi_ordrnumb CHAR(6),
```

```

    IN oi_itemnumb CHAR(5)
)
BEGIN
    SQLStatement st = new SQLStatement (
        "INSERT INTO ordritem VALUES (?, ?, ?, ?)"
    );
    st.setParam (1, oi_seqnumb);
    st.setParam (2, oi_quantity);
    st.setParam (3, oi_ordrnumb);
    st.setParam (4, oi_itemnumb);
    st.execute();
END

CREATE PROCEDURE Add_ItemMaster_Record (
    IN im_itemwght INTEGER,
    IN im_itempric MONEY,
    IN im_itemnumb CHAR(5),
    IN im_itemdesc VARCHAR(47)
)
BEGIN
    SQLStatement st = new SQLStatement (
        "INSERT INTO itemmast VALUES (?, ?, ?, ?)"
    );
    st.setParam (1, im_itemwght);
    st.setParam (2, im_itempric);
    st.setParam (3, im_itemnumb);
    st.setParam (4, im_itemdesc);
    st.execute();
END

CREATE PROCEDURE Delete_Records (
    IN tablename CHAR(256)
)
BEGIN
    SQLStatement st = new SQLStatement (
        "DELETE FROM " + tablename
    );
    st.execute();
END

CREATE PROCEDURE Display_Records ()
RESULT (
    Name    CHAR(47),
    Total   FLOAT
)
BEGIN
    SQLCursor cur = new SQLCursor (
        "SELECT cm_custname, SUM(im_itempric * oi_quantity) " +
        "FROM custmast, custordr, ordritem, itemmast " +
        "WHERE co_custnumb = cm_custnumb AND co_ordrnumb = oi_ordrnumb AND oi_itemnumb = im_itemnumb"
    +
        "GROUP BY cm_custnumb, cm_custname"
    );
    cur.open();
    cur.fetch();
    while (cur.found())
    {
        SQLResultSet.set(1, cur.getValue(1, CHAR));
        SQLResultSet.set(2, cur.getValue(2, FLOAT));

        SQLResultSet.insert();
        cur.fetch();
    }
    cur.close();
END

-- Manage
ECHO MANAGE;

```

```
ECHO Delete records...;
CALL Delete_Records('custmast');
CALL Delete_Records('custordr');
CALL Delete_Records('ordritem');
CALL Delete_Records('itemmast');

ECHO Add records...;
CALL Add_CustomerMaster_Record('1000', '92867', 'CA', '1', 'Bryan Williams', '2999 Regency',
'Orange');
CALL Add_CustomerMaster_Record('1001', '61434', 'CT', '1', 'Michael Jordan', '13 Main',
'Harford');
CALL Add_CustomerMaster_Record('1002', '73677', 'GA', '1', 'Joshua Brown', '4356 Cambridge',
'Atlanta');
CALL Add_CustomerMaster_Record('1003', '10034', 'MO', '1', 'Keyon Dooling', '19771 Park Avenue',
'Columbia');

CALL Add_CustomerOrders_Record('9/1/2002', '9/5/2002', '1', '1001');
CALL Add_CustomerOrders_Record('9/2/2002', '9/6/2002', '2', '1002');

CALL Add_OrderItems_Record(1, 2, '1', '1');
CALL Add_OrderItems_Record(2, 1, '1', '2');
CALL Add_OrderItems_Record(3, 1, '1', '3');
CALL Add_OrderItems_Record(1, 3, '2', '3');

CALL Add_ItemMaster_Record(10, 19.95, '1', 'Hammer');
CALL Add_ItemMaster_Record(3, 9.99, '2', 'Wrench');
CALL Add_ItemMaster_Record(4, 16.59, '3', 'Saw');
CALL Add_ItemMaster_Record(1, 3.98, '4', 'Pliers');

ECHO Display records...;
CALL Display_Records();
```

## Done

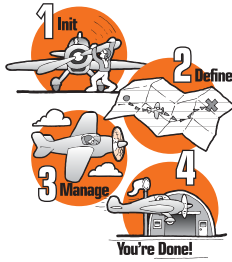


When an application and/or process has completed operations with the database, it must release resources by disconnecting from the database engine.

Below is the code for **Done()**:

```
-- Done  
ECHO DONE;
```

## Additional Resources



We encourage you to explore the additional resources listed here:

- Complete SQL script for this tutorial can be found in SPTTutorial2.sql in your installation directory, within the 'sdk\sql.stored.procs\tutorials' directory for your platform.  
Example for the Windows platform:  
C:\FairCom\V9.0.0\win32\sdk\sql.stored.procs\tutorials\SPTTutorial2.sql.
- Additional documentation may be found on the FairCom Web site at: [www.faircom.com](http://www.faircom.com)

## 2.3 Record/Row Locking

..\sdk\sql.stored.procs\tutorials\SPTTutorial3.sql

Now we will explore row/record locks using the c-treeACE SQL Stored Procedures Technology.

The functionality for this tutorial focuses on inserting/adding rows/records, then updating a single row/record in the customer master table under locking control. The application will pause after a LOCK is placed on a row/record. Another instance of this application should then be launched, which will block, waiting on the lock held by the first instance. Pressing the <Enter> key will enable the first instance to proceed. This will result in removing the lock thereby allowing the second instance to continue execution. Launching two processes provides a visual demonstration of the effects of locking and a basis for experimentation on your own.

Like all other examples in the c-tree tutorial series, this tutorial simplifies the creation and use of a database into four simple steps: Initialize(), Define(), Manage(), and you're Done() !

### Tutorial #3: Locking

Here we demonstrate the enforcement of data integrity by introducing record/row "locking".

- Initialize() - Connects to the c-treeACE Database Engine.
- Define() - Defines and creates a "customer master" (custmast) table/file.
- Manage() - Adds a few rows/records; Reads the rows/records back from the database; displays the column/field content. Then demonstrates an update operation under locking control, and a scenario that shows a locking conflict.
- Done() - Disconnects from c-treeACE Database Engine.

Note these sections in our SQL Script:

```
-- Initialize
ECHO INIT;

-- Define
ECHO DEFINE;

-- Manage
ECHO MANAGE;

-- Done
ECHO DONE;
```

We suggest opening the source code with your own editor.



Continue now to review these four steps.

## Init



First we need to open a connection to a database by providing the c-treeACE Database Engine with a user name, password and the database name.

Below is the code for **Initialize()**:

```
SET ECHO OFF

-- Initialize
ECHO INIT;

SET AUTOCOMMIT OFF;
```

## Define



The Define() step is where specific data definitions are established by your application and/or process. This involves defining columns/fields and creating the tables/files with optional indices.

Below is the code for **Define()**:

```
DROP PROCEDURE Define;

CREATE PROCEDURE Define()
BEGIN
    SQLStatement cm = new SQLStatement (
        "CREATE TABLE custmast (" +
            "cm_custnumb CHAR(4), " +
            "cm_custzipc CHAR(9), " +
            "cm_custstat CHAR(2), " +
            "cm_custrtng CHAR(1), " +
            "cm_custname VARCHAR(47), " +
            "cm_custaddr VARCHAR(47), " +
            "cm_custcity VARCHAR(47))"
    );
    cm.execute();
    SQLStatement cmi = new SQLStatement (
        "CREATE UNIQUE INDEX cm_custnumb_idx ON custmast (cm_custnumb)"
    );
    cmi.execute();
END

-- Define
ECHO DEFINE;

CALL Define();
COMMIT WORK;
```

## Manage



The manage step provides data management functionality for your application and/or process.

Below is the code for **Manage()**:

```
DROP PROCEDURE Add_CustomerMaster_Record;
DROP PROCEDURE Delete_Records;

CREATE PROCEDURE Add_CustomerMaster_Record (
    IN cm_custnumb CHAR(4),
    IN cm_custzipc CHAR(9),
    IN cm_custstat CHAR(2),
    IN cm_custrtng CHAR(1),
    IN cm_custname VARCHAR(47),
    IN cm_custaddr VARCHAR(47),
    IN cm_custcity VARCHAR(47)
)
BEGIN
    SQLIStatement st = new SQLIStatement (
        "INSERT INTO custmast VALUES (?, ?, ?, ?, ?, ?, ?)"
    );
    st.setParam (1, cm_custnumb);
    st.setParam (2, cm_custzipc);
    st.setParam (3, cm_custstat);
    st.setParam (4, cm_custrtng);
    st.setParam (5, cm_custname);
    st.setParam (6, cm_custaddr);
    st.setParam (7, cm_custcity);
    st.execute();
END

CREATE PROCEDURE Delete_Records (
    IN tablename CHAR(256)
)
BEGIN
    SQLIStatement st = new SQLIStatement (
        "DELETE FROM " + tablename
    );
    st.execute();
END

-- Manage
ECHO MANAGE;

ECHO Delete records...;
CALL Delete_Records('custmast');

ECHO Add records...;
CALL Add_CustomerMaster_Record('1000', '92867', 'CA', '1', 'Bryan Williams', '2999 Regency',
'Orange');
CALL Add_CustomerMaster_Record('1001', '61434', 'CT', '1', 'Michael Jordan', '13 Main',
'Harford');
CALL Add_CustomerMaster_Record('1002', '73677', 'GA', '1', 'Joshua Brown', '4356 Cambridge',
'Atlanta');
CALL Add_CustomerMaster_Record('1003', '10034', 'MO', '1', 'Keyon Dooling', '19771 Park Avenue',
'Columbia');
```

```
COMMIT WORK;
```

```
UPDATE custmast SET cm_custname = 'KEYON DOOLING' WHERE cm_custnumb = '1003';  
ECHO Issue a COMMIT WORK to commit changes and release locks
```

## Done

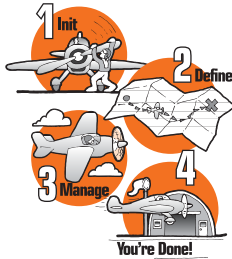


When an application and/or process has completed operations with the database, it must release resources by disconnecting from the database engine.

Below is the code for **Done()**:

```
/*  
 * Done ()  
 *  
 * This function handles the housekeeping of closing connection and  
 * freeing of associated memory  
 */
```

## Additional Resources



We encourage you to explore the additional resources listed here:

- Complete SQL script for this tutorial can be found in SPTTutorial3.sql in your installation directory, within the 'sdk\sql.stored.procs\tutorials' directory for your platform.  
Example for the Windows platform:  
C:\FairCom\V9.0.0\win32\sdk\sql.stored.procs\tutorials\SPTTutorial3.sql.
- Additional documentation may be found on the FairCom Web site at: [www.faircom.com](http://www.faircom.com)

## 2.4 Transaction Processing

..\sdk\sql.stored.procs\tutorials\SPTTutorial4.sql

Now we will discuss transaction processing as it relates to the c-treeACE SQL Stored Procedures Technology.

Transaction processing provides a safe method by which multiple database operations spread across separate tables/files are guaranteed to be atomic. By atomic, we mean that, within a transaction, either all of the operations succeed or none of the operations succeed. This "either all or none" atomicity insures that the integrity of the data in related tables/files is secure.

Like all other examples in the c-tree tutorial series, this tutorial simplifies the creation and use of a database into four simple steps: Initialize(), Define(), Manage(), and You're Done() !

### Tutorial #4: Transaction Processing

Here we demonstrate transaction control.

- Initialize() - Connects to the c-treeACE Database Engine.
- Define() - Defines and creates our four tables/files.
- Manage() - Adds rows/records to multiple tables/files under transaction control.
- Done() - Disconnects from c-treeACE Database Engine.

Note these sections in our SQL Script:

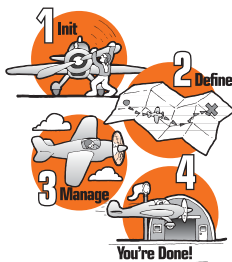
```
-- Initialize
ECHO INIT;

-- Define
ECHO DEFINE;

-- Manage
ECHO MANAGE;

-- Done
ECHO DONE;
```

We suggest opening the source code with your own editor.



Continue now to review these four steps.

## Init



First we need to open a connection to a database by providing the c-treeACE Database Engine with a user name, password and the database name.

Below is the code for **Initialize()**:

```
SET ECHO OFF

-- Initialize
ECHO INIT;

SET AUTOCOMMIT OFF;
```

## Define



The Define() step is where specific data definitions are established by your application and/or process. This involves defining columns/fields and creating the tables/files with optional indices.

Below is the code for **Define()**:

```
DROP PROCEDURE Define;

CREATE PROCEDURE Define()
BEGIN
    int TABLE_ALREADY_EXIST = -20041;
    int INDEX_ALREADY_EXIST = -20028;

    try
    {
        SQLIStatement cm = new SQLIStatement (
            "CREATE TABLE custmast (" +
            "cm_custnumb CHAR(4), " +
            "cm_custzipc CHAR(9), " +
            "cm_custstat CHAR(2), " +
            "cm_custrtng CHAR(1), " +
            "cm_custname VARCHAR(47), " +
            "cm_custaddr VARCHAR(47), " +
            "cm_custcity VARCHAR(47)"
        );
        cm.execute();
    }
    catch (DhSQLException e)
    {
        if (e.sqlErr != TABLE_ALREADY_EXIST)
            throw e;
    }
    try
    {
        SQLIStatement cm1 = new SQLIStatement (
            "CREATE INDEX cm_custnumb_idx ON custmast (cm_custnumb)"
        );
        cm1.execute();
    }
    catch (DhSQLException e)
    {
        if (e.sqlErr != INDEX_ALREADY_EXIST)
            throw e;
    }
    try
    {
        SQLIStatement co = new SQLIStatement (
            "CREATE TABLE custordr (" +
            "co_ordrdate DATE, " +
            "co_promdate DATE, " +
            "co_ordrnumb CHAR(6), " +
            "co_custnumb CHAR(4)"
        );
        co.execute();
    }
}
```

```

catch (DhSQLException e)
{
    if (e.sqlErr != TABLE_ALREADY_EXIST)
        throw e;
}
try
{
    SQLIStatement col = new SQLIStatement (
        "CREATE INDEX co_ordrnumb_idx ON custordr (co_ordrnumb)"
    );
    col.execute();
}
catch (DhSQLException e)
{
    if (e.sqlErr != INDEX_ALREADY_EXIST)
        throw e;
}
try
{
    SQLIStatement co2 = new SQLIStatement (
        "CREATE INDEX co_custnumb_idx ON custordr (co_custnumb)"
    );
    co2.execute();
}
catch (DhSQLException e)
{
    if (e.sqlErr != INDEX_ALREADY_EXIST)
        throw e;
}

try
{
    SQLIStatement oi = new SQLIStatement (
        "CREATE TABLE ordritem (" +
        "oi_sequnumb SMALLINT, " +
        "oi_quantity SMALLINT, " +
        "oi_ordrnumb CHAR(6), " +
        "oi_itemnumb CHAR(5))"
    );
    oi.execute();
}
catch (DhSQLException e)
{
    if (e.sqlErr != TABLE_ALREADY_EXIST)
        throw e;
}
try
{
    SQLIStatement oil = new SQLIStatement (
        "CREATE INDEX oi_ordrnumb_idx ON ordritem (oi_ordrnumb, oi_sequnumb)"
    );
    oil.execute();
}
catch (DhSQLException e)
{
    if (e.sqlErr != INDEX_ALREADY_EXIST)
        throw e;
}
try
{
    SQLIStatement oi2 = new SQLIStatement (
        "CREATE INDEX oi_itemnumb_idx ON ordritem (oi_itemnumb)"
    );
    oi2.execute();
}
catch (DhSQLException e)
{
    if (e.sqlErr != INDEX_ALREADY_EXIST)
        throw e;
}

```

```
try
{
    SQLStatement im = new SQLStatement (
        "CREATE TABLE itemmast (" +
            "im_itemwght INTEGER, " +
            "im_itempric MONEY, " +
            "im_itemnumb CHAR(5), " +
            "im_itemdesc VARCHAR(47)"
        );
    im.execute();
}
catch (DhSQLException e)
{
    if (e.sqlErr != TABLE_ALREADY_EXIST)
        throw e;
}
try
{
    SQLStatement im1 = new SQLStatement (
        "CREATE INDEX im_itemnumb_idx ON itemmast (im_itemnumb)"
    );
    im1.execute();
}
catch (DhSQLException e)
{
    if (e.sqlErr != INDEX_ALREADY_EXIST)
        throw e;
}
END

-- Define
ECHO DEFINE;

CALL Define();
COMMIT WORK;
```

## Manage



The manage step provides data management functionality for your application and/or process.

Below is the code for **Manage()**:

```

DROP PROCEDURE Add_CustomerMaster_Record;
DROP PROCEDURE Add_CustomerOrders_Record;
DROP PROCEDURE Add_OrderItems_Record;
DROP PROCEDURE Add_ItemMaster_Record;
DROP PROCEDURE Delete_Records;
DROP TRIGGER Validate_CustomerOrders_Record;
DROP TRIGGER Validate_OrderItems_Record;\

CREATE PROCEDURE Add_CustomerMaster_Record (
    IN cm_custnumb CHAR(4),
    IN cm_custzipc CHAR(9),
    IN cm_custstat CHAR(2),
    IN cm_custrtng CHAR(1),
    IN cm_custname VARCHAR(47),
    IN cm_custaddr VARCHAR(47),
    IN cm_custcity VARCHAR(47)
)
BEGIN
    SQLStatement st = new SQLStatement (
        "INSERT INTO custmast VALUES (?, ?, ?, ?, ?, ?, ?)"
    );
    st.setParam (1, cm_custnumb);
    st.setParam (2, cm_custzipc);
    st.setParam (3, cm_custstat);
    st.setParam (4, cm_custrtng);
    st.setParam (5, cm_custname);
    st.setParam (6, cm_custaddr);
    st.setParam (7, cm_custcity);
    st.execute();
END

CREATE PROCEDURE Add_CustomerOrders_Record (
    IN co_ordrdate DATE,
    IN co_promdate DATE,
    IN co_ordrnumb CHAR(6),
    IN co_custnumb CHAR(4)
)
BEGIN
    SQLStatement st = new SQLStatement (
        "INSERT INTO custordr VALUES (?, ?, ?, ?)"
    );
    st.setParam (1, co_ordrdate);
    st.setParam (2, co_promdate);
    st.setParam (3, co_ordrnumb);
    st.setParam (4, co_custnumb);
    st.execute();
END

CREATE PROCEDURE Add_OrderItems_Record (
    IN oi_seqnumb SMALLINT,
    IN oi_quantity SMALLINT,

```

```
        IN oi_ordrnumb CHAR(6),
        IN oi_itemnumb CHAR(5)
    )
BEGIN
    SQLStatement st = new SQLStatement (
        "INSERT INTO ordritem VALUES (?, ?, ?, ?)"
    );
    st.setParam (1, oi_sequnumb);
    st.setParam (2, oi_quantity);
    st.setParam (3, oi_ordrnumb);
    st.setParam (4, oi_itemnumb);
    st.execute();
END

CREATE PROCEDURE Add_ItemMaster_Record (
    IN im_itemwght INTEGER,
    IN im_itempric MONEY,
    IN im_itemnumb CHAR(5),
    IN im_itemdesc VARCHAR(47)
)
BEGIN
    SQLStatement st = new SQLStatement (
        "INSERT INTO itemmast VALUES (?, ?, ?, ?)"
    );
    st.setParam (1, im_itemwght);
    st.setParam (2, im_itempric);
    st.setParam (3, im_itemnumb);
    st.setParam (4, im_itemdesc);
    st.execute();
END

CREATE TRIGGER Validate_CustomerOrders_Record
BEFORE INSERT ON custordr
REFERENCING NEWROW
FOR EACH ROW
BEGIN
    SQLCursor cur = new SQLCursor ("SELECT cm_custnumb FROM custmast WHERE cm_custnumb = ?");

    String custnumb = (String) NEWROW.getValue(4, CHAR);
    cur.setParam(1, custnumb);

    cur.open();
    cur.fetch();
    if (!cur.found())
    {
        cur.close();
        DhSQLException ex = new DhSQLException (666, new String("Invalid customer number"));
        throw ex;
    }
    cur.close();
END

CREATE TRIGGER Validate_OrderItems_Record
BEFORE INSERT ON ordritem
REFERENCING NEWROW
FOR EACH ROW
BEGIN
    SQLCursor cur = new SQLCursor ("SELECT co_ordrnumb FROM custordr WHERE co_ordrnumb = ?");

    String ordrnumb = (String) NEWROW.getValue(3, CHAR);
    cur.setParam(1, ordrnumb);

    cur.open();
    cur.fetch();
    if (!cur.found())
    {
        cur.close();
        DhSQLException ex = new DhSQLException (666, new String("Invalid order number"));
        throw ex;
    }
    cur.close();
END
```

```

cur = new SQLCursor ("SELECT im_itemnumb FROM itemmast WHERE im_itemnumb = ?");

String itemnumb = (String) NEWROW.getValue(4, CHAR);
cur.setParam(1, itemnumb);

cur.open();
cur.fetch();
if (!cur.found())
{
    cur.close();
    DhSQLException ex = new DhSQLException (666, new String("Invalid item number"));
    throw ex;
}
cur.close();
END

CREATE PROCEDURE Delete_Records (
    IN tablename CHAR(256)
)
BEGIN
    SQLIStatement st = new SQLIStatement (
        "DELETE FROM " + tablename
    );
    st.execute();
END

-- Manage
ECHO MANAGE;

ECHO Delete records...;
CALL Delete_Records('custmast');
CALL Delete_Records('custordr');
CALL Delete_Records('ordritem');
CALL Delete_Records('itemmast');

ECHO Add records...;
CALL Add_CustomerMaster_Record('1000', '92867', 'CA', '1', 'Bryan Williams', '2999 Regency',
'Orange');
CALL Add_CustomerMaster_Record('1001', '61434', 'CT', '1', 'Michael Jordan', '13 Main',
'Harford');
CALL Add_CustomerMaster_Record('1002', '73677', 'GA', '1', 'Joshua Brown', '4356 Cambridge',
'Atlanta');
CALL Add_CustomerMaster_Record('1003', '10034', 'MO', '1', 'Keyon Dooling', '19771 Park Avenue',
'Columbia');
CALL Add_ItemMaster_Record(10, 19.95, '1', 'Hammer');
CALL Add_ItemMaster_Record(3, 9.99, '2', 'Wrench');
CALL Add_ItemMaster_Record(4, 16.59, '3', 'Saw');
CALL Add_ItemMaster_Record(1, 3.98, '4', 'Pliers');
COMMIT WORK;

CALL Add_CustomerOrders_Record ('09/01/2002', '09/05/2002', '1', '1001');
CALL Add_OrderItems_Record (1, 2, '1', '1');
CALL Add_OrderItems_Record (2, 1, '1', '2');

CALL Add_CustomerOrders_Record ('09/02/2002', '09/06/2002', '2', '9999');
ECHO NOTE: trigger execution failure expected
CALL Add_OrderItems_Record (1, 1, '2', '3');
ECHO NOTE: trigger execution failure expected
CALL Add_OrderItems_Record (2, 3, '2', '4');
ECHO NOTE: trigger execution failure expected

CALL Add_CustomerOrders_Record ('09/22/2002', '09/26/2002', '3', '1003');
CALL Add_OrderItems_Record (1, 2, '3', '3');
CALL Add_OrderItems_Record (2, 2, '3', '99');
ECHO NOTE: trigger execution failure expected
COMMIT WORK;

SELECT co_ordrnumb "Order", co_custnumb "Name" FROM custordr;
SELECT oi_ordrnumb "Order", oi_itemnumb "Item" FROM ordritem;

```

## Done

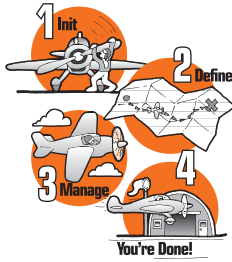


When an application and/or process has completed operations with the database, it must release resources by disconnecting from the database engine.

Below is the code for **Done()**:

```
-- Done  
ECHO DONE;  
DROP TRIGGER Validate_CustomerOrders_Record;  
DROP TRIGGER Validate_OrderItems_Record;  
COMMIT WORK;
```

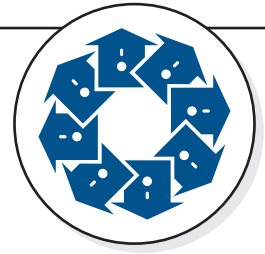
## Additional Resources



We encourage you to explore the additional resources listed here:

- Complete SQL script for this tutorial can be found in SPTTutorial4.sql in your installation directory, within the 'sdk\sql.stored.procs\tutorials' directory for your platform.  
Example for the Windows platform:  
C:\FairCom\V9.0.0\win32\sdk\sql.stored.procs\tutorials\SPTTutorial4.sql.
- Additional documentation may be found on the FairCom Web site at: [www.faircom.com](http://www.faircom.com)





## Using Stored Procedures

### 3.1 Introduction

This chapter describes basic stored procedure operations (such as creating and deleting) and provides a tutorial on using c-treeSQL - supplied classes. See "[Java Class Reference](#)" for detailed reference information on the classes and their methods.

Stored procedures extend the SQL capabilities of a database by adding control flow through Java program constructs that enforce business rules and perform administrative tasks.

Stored procedures can take advantage of the power of Java programming features. Stored procedures can:

- Receive and return input and output parameters
- Handle exceptions
- Include any number and kind of SQL statements to access the database
- Return a procedure result set to the calling application
- Make calls to other procedures
- Use predefined and external Java classes

c-treeSQL provides support for SQL statements in Java through several classes. The following table summarizes the functionality of these c-treeSQL - supplied classes. (See "[Java Class Reference](#)" for detailed reference information.)

#### Summary of c-treeSQL Java Classes

Functionality	c-treeSQL Java Class
Immediate (one-time) execution of c-treeSQL statements	<i>SQLStatement</i>
Prepared (repeated) execution of c-treeSQL statements	<i>SQLPStatement</i>
Retrieval of c-treeSQL result sets	<i>SQLCursor</i>
Returning a procedure result set to the application	<i>DhSQLResultSet</i>
Exception handling for c-treeSQL statements	<i>DhSQLException</i>

## 3.2 Stored Procedure Basics

This section discusses how to get started writing stored procedures.

The c-treeSQL `CREATE PROCEDURE` statement provides the basic framework for stored procedures. Use the `CREATE PROCEDURE` statement to submit a Java code snippet that will be compiled and stored in the database. The syntax for the `CREATE PROCEDURE` statement is:

```
CREATE PROCEDURE [ owner_name. ] procname
  ( [ parameter_decl [ , ... ] ] ) ] ]
  [ RESULT ( column_name data_type [ , ... ] ) ]
  [ IMPORT
    java_import_clause ]
  BEGIN
    java_snippet
  END

parameter_decl ::
  { IN | OUT | INOUT } parameter_name data_type
```

**Note:** When creating stored procedures, the key words `BEGIN`, `RESULT`, `IMPORT` and `END` should start at the first column of the line.

### What Is a Java Snippet?

The core of the stored procedure is the `java_snippet`. The snippet contains a sequence of Java statements. When it processes a `CREATE PROCEDURE` statement, c-treeSQL adds header and footer “wrapper” code to the Java snippet. This wrapper code:

- Declares a class with the name `username_procname_SP` (username is the user name of the database connection that issued the `CREATE PROCEDURE` statement and `procname` is the name supplied in the `CREATE PROCEDURE` statement).
- Declares a method within that class that includes the Java snippet.

When an application calls the stored procedure, the c-treeSQL Server calls the Java virtual machine to invoke the method of the `username_procname_SP` class.

### Structure of Stored Procedures

There are two parts to any stored procedure:

- The **procedure specification** must provide the name of the procedure and may include other optional clauses.
- The **procedure body** contains the Java code that executes when an application invokes the procedure.

A simple stored procedure requires only the procedure name in the specification and a statement that requires no parameters in the body, as shown in the following example. The procedure in the following example assumes a table called `HelloWorldTBL` exists, and inserts a string into that table.

## A Simple Stored Procedure Example

```
CREATE PROCEDURE HelloWorld ()
```

**Procedure  
Specification**

```
BEGIN
    SQLStatement Insert_HelloWorld
        = new SQLStatement ("INSERT INTO
            HelloWorldTBL(fld1) values
            ('Hello World!')");
    Insert_HelloWorld.execute();
END
```

**Procedure Body**

From interactive SQL, you could execute the procedure as follows:

```
ISQL> CREATE TABLE helloworldTBL (fld1 CHAR(100));
ISQL> CALL HelloWorld();
0 records returned
ISQL> SELECT * FROM helloworldTBL;
FLD1
----
Hello World!
1 record selected
```

The procedure specification can also contain other optional clauses:

- **Parameter** declarations specify the name and type of parameters that the calling application will pass and receive from the procedure. Parameters can be of type input, output, or both.
- The **procedure result set** declaration details the names and types of fields in a result set the procedure generates. The result set is a set of rows that contain data generated by the procedure. If a procedure retrieves rows from a database table, for instance, it can store the rows in a result set for access by applications and other procedures. (Note that the names specified in the result-set declaration are not used within the stored procedure body. Instead, methods of the c-treeSQL Java classes refer to fields in the result set by ordinal number, not by name.)
- The **import clause** specifies which packages the procedure needs from the Java core API. By default, the Java compiler imports the java.lang package. The `IMPORT` clause must list any other packages the procedure uses.

The following example shows a more complex procedure specification that contains these elements.

### Complete Stored Procedure Example

```
CREATE PROCEDURE new_sal
  (IN deptnum INTEGER,      Parameter
   IN pct_incr INTEGER)    } declarations

RESULT
  (empname CHAR(20),      Procedure result
   oldsal NUMERIC,        } set declaration
   newsal NUMERIC)

IMPORT
  import java.dbutils.SequenceType; } Import
clause
```

**Procedure  
Specification**

```
BEGIN
.
.
.
END
```

**Procedure  
Body**

### Setting Up Your Environment to Write Stored Procedures

Before you create stored procedures, you need to have a Java development environment running on your system. c-treeSQL stored procedures support the following environments:

- On Unix and Windows XP/2003/Vista: JavaSoft JDK™ Version 1.5

Also, make sure environment variables include the settings shown in the following table:

#### Java-Related Environment Variables

Variable	Windows XP/2003/Vista ( <i>ctsrvr.cfg</i> )	Unix ( <i>ctsrvr.cfg</i> )
JAVA_COMPILER (path for java compiler)	JAVA_COMPILER=c:\progra~1\j2sdk1.4.2\bin\javac.exe	JAVA_COMPILER=c:\progra~1\j2sdk1.4.2\bin\javac.exe
PATH	No specific path settings for stored procedures	setenv PATH \${JDKHOME}\bin

## Writing Stored Procedures

Use any text editor to write the `CREATE PROCEDURE` statement, and save the source code as a text file. That way, you can easily modify the source code and try again if it generates syntax or Java compilation errors.

Submit the file containing the `CREATE PROCEDURE` statement to interactive SQL as a script, as shown in the following example.

### Example Submitting Scripts to Create Stored Procedures

```
$ type helloworldscript.sql
SET ECHO ON;
SET AUTOCOMMIT OFF;
CREATE PROCEDURE HelloWorld ()

BEGIN
    SQLIStatement Insert_HelloWorld = new SQLIStatement
        ("INSERT INTO HelloWorldTBL(fld1) values ('Hello World!')");
    Insert_HelloWorld.execute();
END
commit work;
$ isql -s hello_world_script.sql example_db
SET AUTOCOMMIT OFF;
CREATE PROCEDURE HelloWorld ()

BEGIN
    SQLIStatement Insert_HelloWorld = new
        SQLIStatement ("INSERT INTO HelloWorld(fld1)
        values ('Hello World!')");
    Insert_HelloWorld.execute();
END
;
commit work;
$
```

Keep in mind that the Java snippet within the `CREATE PROCEDURE` statement does not execute as a standalone program. Instead, it executes in the context of an application call to the method of the class created by the c-treeSQL Server. This characteristic has the following implications:

- It is meaningless for a snippet to declare a main method, since it will never be executed.
- If the snippet declares any classes, it must instantiate them within the snippet to invoke their methods.
- The c-treeSQL Server redirects the standard output stream to a file, `sql_server.log`. Method invocations such as `System.out.println()` will not display messages on the screen, however, instead writes to that file.

## Invoking Stored Procedures

How applications call stored procedures depends on their environment.

### From ODBC

From ODBC, applications use the ODBC call escape sequence:

```
{ call proc_name [ ( parameter [ , ... ] ) ] }
```

Use parameter markers (question marks used as placeholders) for input or output parameters to the procedure. You can also use literal values for input parameters only. c-treeSQL stored procedures do not support return values in the ODBC escape sequence. See the *Microsoft ODBC Programmer's Reference, Version 3.0*, for more detail on calling procedures from ODBC applications.

Embed the escape sequence in an ODBC **SQLExecDirect()** call to execute the procedure. The following example shows a call to a stored procedure named *order\_parts* that passes a single input parameter using a parameter marker.

### Invoking a Stored Procedure from an ODBC Application

```
SQLINTEGER Part_num;
SQLINTEGER Part_numInd = 0;

// Bind the parameter.
SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER, 0, 0,
                  &Part_num, 0, Part_numInd);

// Place the department number in Part_num.
Part_num = 318;

// Execute the statement.
SQLExecDirect (hstmt, "{call order_parts(?)}", SQL_NTS);
```

### From JDBC

The JDBC call escape sequence is the same as in ODBC:

```
{ call proc_name [ ( parameter [ , ... ] ) ] }
```

Embed the escape sequence in a JDBC **CallableStatement.prepareCall()** method invocation. The following example shows the JDBC code parallel to the ODBC code excerpt shown in the previous example.

### Invoking a Stored Procedure from a JDBC Application

```
try {
    CallableStatement statement;
    int Part_num = 318;

    // Associate the statement with the procedure call
    // (conn is a previously-instantiated connection object)
    statement = conn.prepareCall("{call order_parts(?)}");

    // Bind the parameter.
    statement.setInt(1, Part_num);

    // Execute the statement.
    statement.execute();
}
```

### From .NET

From .NET applications, use **DharmaCommand()** object to execute stored procedures. The following C++ example illustrates use of a stored procedure with parameters.

### Invoking a Stored Procedure from a .Net Application

```
// Create a new connection object.
IDbConnection * dhCon = new DharmaConnection
("uid=systpe;pwd=asterix;database=sampled");

// Create a new command object.
IDbCommand *dhCmd = new DharmaCommand ("order_parts(?)",dhCon);

// Set the command type.
```

```

dhCmd->CommandType = CommandType::StoredProcedure;

// Construct a parameter and add to the command.
IDbDataParameter * dhParam = new DharmaParameter();
dhParam->DbType = DbType::Int32;
dhCmd->Parameters->Add(dhParam);

// Prepare the statement.
dhCmd->Prepare();

// Set the value for region id.
dhParam->Value = __box (318);

// Execute the command and create a DataReader object.
IDataReader * dhRdr = dhCmd->ExecuteReader();

```

### From Interactive SQL

From interactive SQL, simply issue the c-treeSQL `CALL` statement. The following example shows the `CALL` statement that invokes the `order_parts` stored procedure, using a literal value instead of a parameter marker.

### Invoking a Stored Procedure from Interactive SQL

```
CALL order_parts (318);
```

## Modifying and Deleting Stored Procedures

There is no “ALTER PROCEDURE” statement. To modify a procedure, you will need to drop and recreate it. To recreate the procedure, you need the original source of the `CREATE PROCEDURE` statement. Before you drop the procedure, decide which approach you will use to recreate it:

- If you kept the original procedure definition in an c-treeSQL script file, simply edit the file and resubmit it through interactive SQL.
- Query system tables to extract the source of the `CREATE PROCEDURE` statement to a file.

The c-treeSQL `DROP PROCEDURE` statement deletes stored procedures from the database. Exercise care in dropping procedures, since any procedure that calls the dropped procedure will fail.

## Debugging Stored Procedures

If there’s a Java compilation error, the c-treeSQL Server returns the error at create time and does not create the procedure.

**Note:** If the compilation of the procedure fails, the c-treeSQL Server returns only the first error to the calling application.

The following example invokes an ISQL script that attempts to create a procedure that fails and generates a Java compilation error.

### Java Compilation Error Example

```

C:\example_scripts>type type_mismatch.sql
CREATE PROCEDURE error_proc()
BEGIN
    Double dbl_val;
    Integer int_val = new Integer(1234);
    dbl_val = int_val;
END
C:\example_scripts>isql -s type_mismatch.sql testdb

```

```
error(-20141): error in compiling the stored procedure
:03: Incompatible type for =. Can't convert java.lang.Integer to java.lang.Double.
```

At run time, the c-treeSQL Server creates a log file, *sql\_server.log*.

You can write custom messages to the log file with the common methods **log()** and **err()**, which write a character-string message to *sql\_server.log*.

You can include the values of variables or return values of methods in the string using the standard Java concatenation operator (+) as shown in the following example.

### Logging Messages Example

```
SQLCursor select_syscalctable = new SQLCursor (
"SELECT fld FROM ADMIN.syscalctable ");
select_syscalctable.open();
select_syscalctable.fetch();
log ("Any records? Found returned " + select_syscalctable.found());
.
.
.
```

The log invocation in the previous example writes the following line to the *sql\_server.log* file:

```
Any records? Found returned true
```

## Transactions and Stored Procedures

Any updates done by a stored procedure become part of the transaction that called the procedure. The transaction behavior is similar to executing the sequence of c-treeSQL statements in the procedure directly by the calling application.

Stored procedures cannot contain **COMMIT** or **ROLLBACK** statements.

## Stored Procedure Security

- Users issuing the **CREATE PROCEDURE** statement must have the **DBA** privilege or **RESOURCE** privilege.
- The owner or users with the **DBA** privilege can execute or drop any stored procedure, and grant the **EXECUTE** privilege to other users.
- When a procedure is being executed on behalf of a user with **EXECUTE** privilege on that procedure, for the objects that are accessed by the procedure, the procedure owner's privileges are checked and not the user's. This enables a user to execute a procedure successfully even when he does not have the privileges to directly access the objects that are accessed by the procedure, so long as he has **EXECUTE** privilege on the procedure.

## Restrictions on Calling Java Methods in Stored Procedures

Java stored procedures cannot use the **System.exit()** method. Calling this method in Stored Procedure body results in a compilation error.

## 3.3 Using the c-treeSQL Java Classes

This section describes how you use the c-treeSQL Java classes to issue and process c-treeSQL statements in Java stored procedures.

To process c-treeSQL statements in a stored procedure, you need to know whether the SQL statement generates results (in other words, if the statement is a query) or not. `SELECT` statements, for example, generate results: they retrieve data from one or more database tables and return the results as rows in a table.

Whether a statement generates such an SQL result set dictates which c-treeSQL Java classes you use to issue it:

- To issue c-treeSQL statements that do not generate results (such as `INSERT`, `GRANT`, or `CREATE`), use either the `SQLStatement` class (for one-time execution) or the `SQLPStatement` class (for repeated execution).
- To issue c-treeSQL statements that generate results (`SELECT` and, in some cases, `CALL`), use the `SQLCursor` class to retrieve rows from a database or another procedure's result set.

In either case, if you want to return a result set to the application, use the `DhSQLResultSet` class to store rows of data in a procedure result set. You must use `DhSQLResultSet()` methods to transfer data from an SQL result set to the procedure result set for the calling application to process it. You can also use `DhSQLResultSet()` methods to store rows of data generated internally by the procedure.

In addition, c-treeSQL provides the `DhSQLException` class so procedures can process and generate Java exceptions through the `try`, `catch`, and `throw` constructs.

## Passing Values to SQL Statements

Stored procedures need to be able to pass and receive values from c-treeSQL statements they execute. They do this through the `setParam()` and `getValue()` methods.

### The `setParam` Method: Pass Input Values to SQL Statements

The `setParam()` method sets the value of a c-treeSQL statement's parameter marker to the specified value (a literal, a procedure variable, or a procedure input parameter).

The `setParam()` method takes two arguments:

```
setParam ( marker_num , value ) ;
```

- `marker_num` is an integer that specifies which parameter marker in the SQL statement is to receive the value (1 denotes the first parameter marker, 2 denotes the second, and so on).
- `value` is a literal or the name of a variable or input parameter that contains the value to be assigned to the parameter marker.

The following example shows an excerpt from a stored procedure that uses `setParam()` to assign values from two procedure variables to the parameter markers in an SQL `INSERT` statement. When the procedure executes, it substitutes the value of the `cust_number` procedure variable for the first parameter marker and the value of the `cust_name` variable for the second parameter marker.

### `setParam()` and c-treeSQL Statement Input Parameters

```
SQLStatement insert_cust = new SQLStatement (
    "INSERT INTO customer VALUES (?,?) ");

insert_cust.setParam (1, cust_number);
insert_cust.setParam (2, cust_name);
.
.
.
```

## The `getValue` Method: Pass Values from SQL Result Sets to Variables

The `getValue()` method of the `SQLCursor` class assigns a single value from an SQL result set (returned by an SQL query or another stored procedure) to a procedure variable or output parameter.

The format and arguments for `getValue()` are as follows:

```
getValue ( col_num , data_type ) ;
```

The `getValue()` interface returns an object, it should be explicitly typecasted to appropriate type.

- `col_num` is an integer that specifies which column of the result set is of interest. `getValue()` retrieves the value in the currently-fetched record of the column denoted by `col_num`. (1 denotes the first column of the result set, 2 denotes the second, and so on).
- `data_type` specifies the expected c-treeSQL type of the returned parameter (see "Implicit Data Type Conversion Between SQL and Java Types" for details on how c-treeSQL data types map to Java data types).

The following example shows an excerpt from a stored procedure that uses `getValue()` to assign values from two result-set columns to procedure variables. In this example, the result set is generated by a c-treeSQL `SELECT` statement.

## Using `getValue()` to Pass Values from Result Sets

```
StringBuffer ename = new StringBuffer (20) ;
java.math.BigDecimal esal = new java.math.BigDecimal () ;

SQLCursor empcursor = new SQLCursor (
    "SELECT name, sal FROM emp " ) ;

empcursor.open () ;
empcursor.fetch () ;
if (empcursor.found ())
{
    ename = (StringBuffer)empcursor.getValue (1, CHAR);
    esal = (BigDecimal)empcursor.getValue (2, DECIMAL);
}
.
.
.
```

In the `SELECT` statement in previous example, it was clear that the result set had two columns, `name` and `sal`. If the `SELECT` statement had used a wildcard in its select list (`SELECT * FROM EMP`) you have to know the structure of the `EMP` table in order to correctly specify the column numbers in the `getValue()` method.

## Passing Values to and From Stored Procedures: Input and Output Parameters

Applications need to pass and receive values from the stored procedures they call. They do this through input and output parameters declared in the procedure specification.

Applications can pass and receive values from stored procedures using input and output parameters declared in the stored procedure specification. When it processes the `CREATE PROCEDURE` statement, the c-treeSQL Server declares Java variables of the same name. This means the body of the stored procedure can refer to input and output parameters as if they were Java variables declared in the body of the stored procedure.

**Note:** Procedure result sets are another way for applications to receive output values from a stored procedure. Procedure result sets provide output in a row-oriented tabular format. See [“Returning a Procedure Result Set: the `RESULT` Clause and `DhSQLResultSet`”](#).

Parameter declarations include the parameter type (IN, OUT, or INOUT), the parameter name, and c-treeSQL data type (see [“Implicit Data Type Conversion Between SQL and Java Types”](#) for details of how c-treeSQL data types map to Java data types).

Declare input and output parameters in the specification section of a stored procedure, as shown in the following example.

### Example Input, Output, and Input-Output Parameters

```
CREATE PROCEDURE order_entry (
    IN cust_name    CHAR(20),
    IN item_num     INTEGER,
    IN quantity     INTEGER,
    OUT status_code INTEGER,
    INOUT order_num INTEGER
)
.
.
.
```

**Note:** The input parameter *cust\_name* in the procedure above should be of type CHAR in ANSI versions of c-treeSQL.

When the *order\_entry* stored procedure executes, the calling application passes values for the *cust\_name*, *item\_num*, *quantity*, and *order\_num* input parameters. The body of the procedure refers to them as Java variables. Similarly, Java code in the body of *order\_entry* processes and returns values in the *status\_code* and *order\_num* output parameters. The variable *order\_num* can be used for both passing input and receiving output value from the procedure.

**Note:** The procedure parameter names should always be in lower case. This is because the parameter names are treated as identifiers in the c-treeSQL statement and are converted to lowercase by the c-treeSQL Server. The parameter names used within the procedure body are not converted to lower case. Hence declaring the parameters with upper case may lead to compilation errors.

## Implicit Data Type Conversion Between SQL and Java Types

When the c-treeSQL Server creates a stored procedure, it converts the type of any input and output parameters (see [“The setParam Method: Pass Input Values to SQL Statements”](#)) from the c-treeSQL data types in the procedure specification to Java wrapper types.

The Java.lang package defines classes for all the primitive Java types that “wrap” values of the corresponding primitive type in an object. The c-treeSQL Server converts the c-treeSQL data types declared for input and output parameters to one of these wrapper types, described Mapping Between SQL and Java Data Types.

You must be sure to use wrapper types when declaring procedure variables to use as arguments to the [getValue\(\)](#), [setParam\(\)](#), and [set](#) methods. These methods take objects as arguments and will generate compilation errors if you pass a primitive type to them.

The following example shows two stored procedures. The first tries to use a variable declared as the Java int primitive type as an argument to the [SQLResultSet.set\(\)](#) method, and will generate a compilation error. The second correctly declares the variable using the integer wrapper class.

### Using Wrapper Types as Arguments to c-treeSQL Classes

```
CREATE PROCEDURE type_mismatch( )
```

```

RESULT (res_int INTEGER )
BEGIN
    // Create a variable as (primitive) type int
    // to test the working of set method of SQLResultSet
    int pvar_int = 4;

    // Transfer the value from the procedure variable to the result set.
    SQLResultSet.set(1,pvar_int); //Error!

    // Insert the row into the procedure result set.
    SQLResultSet.insert();
END
CREATE PROCEDURE type_okmatch( )
RESULT (res_int INTEGER )
BEGIN
    // Create a variable as (wrapper) type Integer
    Integer pvar_int = new Integer(4);

    // Transfer the value from the procedure variable to the result set.
    SQLResultSet.set(1,pvar_int); //Success!

    // Insert the row into the procedure result set.
    SQLResultSet.insert();
END

```

When the SQL engine submits the Java class it creates from the stored procedure to the Java compiler, the compiler checks for data-type consistency between the converted parameters and variables you declare in the body of the stored procedure.

To avoid type mismatch errors, use the data-type mappings shown in the following table for declaring parameters and result-set fields in the procedure specification and the Java variables in the procedure body.

SQL Type	Java Wrapper Type	SQL Type	Java Wrapper Type
CHAR	StringBuffer	INTEGER	Integer
VARCHAR	StringBuffer	BIGINT	Long
LONGVARCHAR	StringBuffer	DOUBLE PRECISION	Double
NUMERIC	java.math.BigDecimal	BINARY	byte[ ]
DECIMAL	java.math.BigDecimal	VARBINARY	byte[ ]
MONEY	java.math.BigDecimal	LONGVARBINARY	byte[ ]
BIT	Boolean	DATE	java.sql.Date
TINYINT	byte[1]	TIME	java.sql.Time
SMALLINT	Integer	TIMESTAMP	java.sql.Timestamp
INTEGER	Integer		

## Executing an SQL Statement

If a c-treeSQL statement does not generate a result set, stored procedures can execute it in one of two ways:

- Immediate execution, using methods of the *SQLStatement* class, executes a statement once.

- Prepared execution, using methods of the *SQLPStatement* class, prepares a statement so you can execute it multiple times in a procedure loop.

Both *SQLStatement* and *SQLPStatement* classes can be used to call stored procedures that do not have a RESULT clause (i.e., the JSPs that do not return any resultset). Note that values of OUT and INOUT arguments can be retrieved using the [getParam\(\)](#) method of *SQLStatement*, *SQLPStatement*, and *SQLCursor*.

The following table shows the SQL statements that do not generate result sets. You can execute these statements in a stored procedure using either the *SQLStatement* or *SQLPStatement* class.

### Executable SQL Statements

ALTER TABLE	DROP PROCEDURE
CALL (if no result set)	DROP TABLE
CREATE INDEX	DROP TRIGGER
CREATE SYNONYM	DROP VIEW
CREATE PROCEDURE	GRANT
CREATE TABLE	INSERT
CREATE TRIGGER	RENAME
CREATE VIEW	REVOKE
DELETE	UPDATE
DROP INDEX	UPDATE STATISTICS
DROP SYNONYM	

### Immediate Execution

Use immediate execution when a procedure needs to execute an SQL statement only once. The following example shows an instance of immediate execution.

### Immediate Execution Example

```
CREATE PROCEDURE insert_customer (
    IN cust_number INTEGER,
    IN cust_name CHAR(20)
)
BEGIN
    SQLStatement insert_cust = new SQLStatement (
        "INSERT INTO customer VALUES (?,?) ";
    insert_cust.setParam (1, cust_number);
    insert_cust.setParam (2, cust_name);
    insert_cust.execute ();
END
```

**Note:** The input parameter *cust\_name* in the procedure above should be of type CHAR in ANSI versions of c-treeSQL.

This example inserts a row in a table. The constructor for `SQLStatement()` takes the c-treeSQL `INSERT` statement as its only argument. In this example, the statement includes two parameter markers.

## Prepared Execution

Use prepared execution when you need to execute the same SQL statement repeatedly. Prepared execution avoids the overhead of creating multiple `SQLStatement` objects for a single statement.

The advantage of prepared execution comes when you have the same c-treeSQL statement executed from within a loop. Instead of creating an object during each pass through the loop, prepared execution creates an object once and only passes input parameters for each execution of the statement.

Once a stored procedure creates a `SQLStatement` object, it can execute it multiple times, supplying different values for each execution.

The following example extends the previous example to use prepared execution.

## Prepared Execution Example

```
CREATE PROCEDURE prepared_insert_customer ()
IMPORT
import java.math.*;
BEGIN
    SQLStatement p_insert_cust = new SQLStatement (
        "INSERT INTO customer VALUES (?,?) ");
    int i;
    String [] [] new_custs = {
        {"01", "ABC"},
        {"02", "PQR"},
        {"03", "XYZ"}
    };
    for (i=0; i<new_custs.length; i++)
    {
        p_insert_cust.setParam (1, Integer.valueOf (new_custs[i] [0]));
        p_insert_cust.setParam (2, new_custs[i] [1]);
        p_insert_cust.execute ();
    }
END
```

## Retrieving Data: the `SQLCursor` Class

Methods of the `SQLCursor` class let stored procedures retrieve rows of data.

When stored procedures create an object from the `SQLCursor` class, they pass as an argument a c-treeSQL statement that generates a result set. The c-treeSQL statement is either a `SELECT` or `CALL` statement:

- A `SELECT` statement queries the database and returns data that meets the criteria specified by the query expression in the `SELECT` statement.
- A `CALL` statement invokes another stored procedure that returns a result set specified by the `RESULT` clause of the `CREATE PROCEDURE` statement.

Either way, once the procedure creates an object from the `SQLCursor` class, the processing of results sets follows the same steps:

1. Open the cursor with the `SQLCursor.open()` method
2. Check whether there are any records in the result set with the `SQLCursor.found()` method

3. If there are records in the result set, loop through the result set:
  - Try to fetch a record with the **SQLCursor.fetch()** method
  - Check whether the fetch returned a record with the **SQLCursor.found()** method
  - If the fetch operation returned a record, assign values from the result-set record's fields to procedure variables or procedure output parameters with the **SQLCursor.getValue()** method
  - Process the data in some manner
  - If the fetch operation did not return a record, exit the loop
4. Close the cursor with the **SQLCursor.close()** method

The following example shows an example that uses **SQLCursor()** to process the result set returned by a c-treeSQL **SELECT** statement.

### Processing a Result Set from a SELECT Statement

```
CREATE PROCEDURE get_sal ()
IMPORT
import java.math.BigDecimal;
BEGIN
    StringBuffer ename = new StringBuffer (20) ;
    BigDecimal esal = new BigDecimal () ;

    SQLCursor empcursor = new SQLCursor (
        "SELECT name, sal FROM emp " ) ;

    empcursor.open () ;
    empcursor.fetch () ;
    while (empcursor.found ())
    {
        ename = (StringBuffer)empcursor.getValue (1, CHAR);
        esal = (BigDecimal)empcursor.getValue (2, DECIMAL);
        empcursor.fetch();
    }
    empcursor.close () ;
END
```

Stored procedures also use *SQLCursor* objects to process a result set returned by another stored procedure. Instead of a **SELECT** statement, the **SQLCursor()** constructor includes a **CALL** statement that invokes the desired procedure.

The following example shows an excerpt from a stored procedure that processes the result set returned by another procedure, **get\_customers()** customers which takes *dept\_no* as input and returns two columns *cust\_number* and *cust\_name* as its result set.

### Processing a Result Set from a CALL Statement

```
CREATE PROCEDURE get_customer()
IMPORT
import java.math.BigDecimal;
BEGIN
    SQLCursor cust_cursor = new SQLCursor ("CALL get_customers (?) ") ;
    Integer cust_number;
    StringBuffer cust_name;
    cust_cursor.setParam (1, new Integer(10));
    cust_cursor.open () ;
    cust_cursor.fetch();
    while (cust_cursor.found())
    {
        cust_number = (Integer)cust_cursor.getValue (1, INTEGER);
        cust_name = (Stringz)cust_cursor.getValue (2, CHAR);
        cust_cursor.fetch () ;
    }
END
```

```
    }  
    cust_cursor.close () ;  
END
```

### registerOutParam Method: registering the Type of OUT and INOUT Variables

Before executing a stored procedure call, you must explicitly call `registerOutParam()` to register the type of each OUT and INOUT parameter.

The `registerOutParam()` method of the `SQLCursor` class registers the type of OUT and INOUT parameters.

### Syntax

The format and arguments for the `registerOutParam()` method are as follows:

```
registerOutParam( param_num , data_type ) ;
```

*param\_num*

An integer that specifies which parameter of the called procedure is to be registered. (1 denotes the first parameter, 2 denotes the second, and so on).

*data\_type*

Specifies the required SQL type of the returned parameter (see "Implicit Data Type Conversion Between SQL and Java Types" details of how SQL data types map to Java data types).

The following example shows an excerpt from a stored procedure that uses `registerOutParam()` interface to register the OUT and INOUT parameters.

### registerOutParam() Interface Example

```
CREATE PROCEDURE get_sal (IN cust_num INTEGER,  
                        OUT dept_num INTEGER,  
                        INOUT salary DOUBLE PRECISION  
                        )  
BEGIN  
    Integer dept = new Integer(40);  
    Double incr = 0.25;  
  
    // select department number into dept using cust_num.  
    dept_num = dept;  
    salary = new Double(salary.doubleValue() + (salary.doubleValue) *  
                       incr.doubleValue());  
END  
  
CREATE PROCEDURE call_get_sal()  
BEGIN  
    Integer num = new Integer(54);  
    Double sal = new Double(25000.00);  
    SQLCursor cust_cur = new SQLCursor("call get_sal(?,?,?)");  
    cust_cur.setParam(1,num);  
    cust_cur.registerOutParam(2,INTEGER);  
    cust_cur.registerOutParam(3,DOUBLE);  
    cust_cur.setParam(3,sal);  
    cust_cur.open();  
  
    // process the data  
    cust_cur.close();  
END
```

If all the OUT and INOUT parameters are not registered, then sql engine returns an error: error(-20161): Not all OUT/INOUT parameters are registered.

## Returning a Procedure Result Set: the RESULT Clause and DhSQLResultSet

The `get_sal()` procedure in the previous example used the `SQLCursor.getValue()` method to store the values of a database record in individual variables. But the procedure did not do anything with those values, and they would be overwritten in the next iteration of the loop that fetches records.

The `DhSQLResultSet` class provides a way for a procedure to store rows of data in a procedure result set so the rows can be returned to the application that calls it. There can only be one procedure result set in a stored procedure.

A stored procedure must explicitly process a result set to return it to the calling application:

- Declare the procedure result set through the `RESULT` clause of the procedure specification
- Populate the procedure result set in the body of the procedure using the methods of the `DhSQLResultSet` class

When the c-treeSQL Server creates a Java class from a `CREATE PROCEDURE` statement that contains the `RESULT` clause, it implicitly instantiates an object of type `DhSQLResultSet()`, and calls it `SQLResultSet()`. Invoke methods of the `SQLResultSet()` instance to populate fields and rows of the procedure result set.

The following example extends Example 3-16: Processing a Result Set from a `SELECT` Statement to return a procedure result set. For each row of the c-treeSQL result set assigned to procedure variables, the procedure:

- Assigns the current values in the procedure variables to corresponding fields in the procedure result set with the `DhSQLResultSet.set()` method
- Inserts a row into the procedure result set with the `DhSQLResultSet.insert()` method

### Example Returning a Procedure Result Set From a Stored Procedure

```
CREATE PROCEDURE get_sal2 ()
RESULT (
    empname CHAR(20),
    empsal NUMERIC
)
IMPORT
import java.math.BigDecimal;
BEGIN
    StringBuffer ename = new StringBuffer (20) ;
    BigDecimal esal = new BigDecimal () ;
    SQLCursor empcursor = new SQLCursor (
        "SELECT name, sal FROM emp " ) ;

    empcursor.open () ;
    do
    {
        empcursor.fetch () ;
        if (empcursor.found ())
        {
            ename = (StringBuffer)empcursor.getValue (1, CHAR);
            esal = (BigDecimal)empcursor.getValue (2, DECIMAL);
            SQLResultSet.set (1, ename);
            SQLResultSet.set (2, esal);
            SQLResultSet.insert ();
        }
    } while (empcursor.found () ) ;
    empcursor.close () ;
```

END

**Note:** The *resultset* parameter *empname* in the procedure above should be of type `CHAR` in ANSI versions of c-treeSQL.

## Handling Null Values

Stored procedures need to routinely set and detect null values:

- Stored procedures may need to set the values of a c-treeSQL statement input parameters or procedure result fields to null.
- Stored procedures must check if the value of a field in a c-treeSQL result set is null before assigning it through the `SQLCursor.getValue()` method. (The c-treeSQL Server generates a runtime error if the result-set field specified in `getValue()` is null.)

### Setting SQL Statement Input Params & Procedure Result SetFields to Null

Both the `setParam()` method (see “The setParam Method: Pass Input Values to SQL Statements”) and `set()` method (see “Returning a Procedure Result Set: the RESULT Clause and DhSQLResultSet”) take objects as their value arguments. You can pass a null reference directly to the method or pass a variable which has been assigned the null value. The following example shows using both techniques to set a c-treeSQL input parameter to null.

#### Example Passing Null Values to `setParam()`

```
CREATE PROCEDURE test_nulls( )
BEGIN
    Integer pvar_int1      = new Integer(0);
    Integer pvar_int2      = new Integer(0);
    Integer pvar_int3;
    pvar_int3 = null;
    SQLStatement insert_t1 = new SQLStatement
    ( "INSERT INTO ADMIN.t1 (c1,c2, c3) values (?,?,?) ");
    insert_t1.setParam(1, new Integer(1)); // Set to non-null value
    insert_t1.setParam(2, null);           // Set directly to null
    insert_t1.setParam(3, pvar_int3);     // Set indirectly to null
    insert_t1.execute();
END
```

### Assigning Null Values from SQL Result Sets: `SQLCursor.wasNULL` Method

If the value of the *field* argument to the `SQLCursor.getValue()` method is null, the c-treeSQL Server returns a runtime error:

```
(error(-20144): Null value fetched.)
```

This means you must always check whether a value is null before attempting to assign a value in a c-treeSQL result set to a procedure variable or output parameter. The `SQLCursor` class provides the `wasNULL()` method for this purpose.

The `SQLCuroor.wasNULL()` method returns `TRUE` if a field in the result set is null. It takes a single integer argument that specifies which field of the current row of the result set to check.

The following example illustrates using `wasNULL()`.

## Example Result Sets for Null Values with wasNULL()

```

CREATE PROCEDURE test_nulls2( )
RESULT ( res_int1 INTEGER ,
        res_int2 INTEGER ,
        res_int3 INTEGER )
BEGIN
    Integer pvar_int1      = new Integer(0);
    Integer pvar_int2      = new Integer(0);
    Integer pvar_int3      = new Integer(0);
    SQLCursor select_t1 = new SQLCursor
    ( "SELECT c1, c2, c3 from t1" );

    select_t1.open();
    select_t1.fetch();
    while ( select_t1.found() )
    {
        // Assign values from the current row of the SQL result set
        // to the pvar_intx procedure variables. Must first check
        // whether the values fetched are null: if they are, must set
        // pvars explicitly to null.
        if ((select_t1.wasNULL(1)) == true)
            pvar_int1 = null;
        else
            pvar_int1 = (Integer)select_t1.getValue(1, INTEGER);
        if ((select_t1.wasNULL(2)) == true)
            pvar_int2 = null;
        else
            pvar_int2 = (Integer)select_t2.getValue(2, INTEGER);
        if ((select_t1.wasNULL(3)) == true)
            pvar_int3 = null;
        else
            pvar_int3 = (Integer)select_t13.getValue(3, INTEGER);
        // Transfer the value from the procedure variables to the
        // columns of the current row of the procedure result set.
        SQLResultSet.set(1,pvar_int1);
        SQLResultSet.set(2,pvar_int2);
        SQLResultSet.set(3,pvar_int3);
        // Insert the row into the procedure result set.
        SQLResultSet.insert();

        select_t1.fetch();
    }
    // Close the SQL result set.
    select_t1.close();
END

```

## Handling Errors

c-treeSQL stored procedures use standard Java try/catch constructs to process exceptions.

Any errors in a c-treeSQL statement execution result in the creation of an *DhSQLException* class object. When c-treeSQL detects an error in an SQL statement, it throws an exception. The stored procedure should use try/catch constructs to process such exceptions. The [getDiagnostics\(\)](#) method of the *DhSQLException* class object provides a mechanism to retrieve different details of the error.

The [getDiagnostics\(\)](#) method takes a single argument whose value specifies which error message detail it returns:

### getDiagnostics Method Arguments

Argument Value	Returns
<i>RETURNED_SQLSTATE</i>	The <i>SQLSTATE</i> returned by execution of the previous SQL statement.
<i>MESSAGE_TEXT</i>	The condition indicated by <i>RETURNED_SQLSTATE</i> .
<i>CLASS_ORIGIN</i>	Not currently used. Always returns null.
<i>SUBCLASS_ORIGIN</i>	Not currently used. Always returns null.

The error messages and the associated *SQLSTATE* and c-treeSQL error code values are documented in the *c-treeSQL Reference Manual*, Table B-2.

The following example shows an excerpt from a stored procedure that uses [DhSQLException.getDiagnostics\(\)](#).

### Example Exception Handling with DhSQLException.getDiagnostics()

```
CREATE PROCEDURE test_proc()
BEGIN
    try
    {
        SQLStatement insert_cust = new SQLStatement (
            "INSERT INTO customer VALUES (1,2) ");
    }
    catch (DhSQLException e)
    {
        String errstate = e.getDiagnostics (RETURNED_SQLSTATE) ;
        String errmesg = e.getDiagnostics (MESSAGE_TEXT) ;
    }
END
```

Stored procedures can also throw their own exceptions by instantiating a *DhSQLException* object and throwing the object when the procedure detects an error in execution. The conditions under which the procedure throws the exception object are completely dependent on the procedure.

The following example illustrates using the [DhSQLException\(\)](#) constructor to create an exception object called *excep*. It then throws the *excep* object under all conditions.

### Example Throwing Procedure-Specific Exceptions

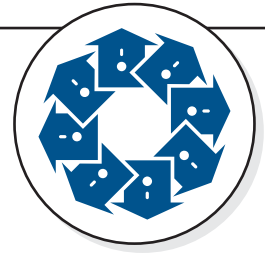
```
CREATE PROCEDURE sp1_02()
BEGIN
    // raising exception
    DhSQLException excep = new DhSQLException(666,new String("Entered the tst02 procedure"));
    if (true)
        throw excep;
END
```

## Calling Stored Procedures from Stored Procedures

Stored procedures and triggers can call other stored procedures. Nesting procedures lets you take advantage of existing procedures. Instead of rewriting the code, procedures can simply issue *CALL* statements to the existing procedures.

Another use for nesting procedures is to assemble result sets generated by queries on different databases into a single result set. With this technique, the stored procedure processes multiple `SELECT` statements through multiple instances of the `SQLCursor` class. For each of the instances, the procedure uses the `DhSQLResultSet` class to add rows to the result set returned by the procedure.





# Using Triggers

## 4.1 Introduction

Triggers are a special type of stored procedure used to maintain database integrity.

Like stored procedures, triggers also contain Java code (embedded in a `CREATE TRIGGER` statement) and use c-treeSQL Java classes. However, triggers are automatically invoked (“fired”) by certain c-treeSQL operations (an `INSERT`, `UPDATE`, or `DELETE` operation) on the trigger’s target table.

This chapter provides a general description of triggers and discusses in detail where trigger procedures differ from stored procedures. Unless otherwise noted, much of the material in Using Stored Procedures also applies to triggers.

## 4.2 Trigger Basics

Use the c-treeSQL `CREATE TRIGGER` statement to create a trigger. The syntax for the `CREATE TRIGGER` statement is:

```
CREATE TRIGGER [ owner_name. ] trigname
  { BEFORE | AFTER }
  { INSERT | DELETE | UPDATE [ OF (column_name [ , ... ] ) }
ON table_name
  [ REFERENCING { OLDROW [ , NEWROW ] | NEWROW [ , OLDROW ] } ]
  [ FOR EACH { ROW | STATEMENT } ]
  [ IMPORT java_import_clause ]

BEGIN
  java_snippet
END
```

### Structure of Triggers

Like a stored procedure, a trigger has a specification and a body.

The body of a trigger is the same as that of a stored procedure: `BEGIN` and `END` delimiters enclosing a Java snippet. The Java code in the snippet defines the triggered action that executes when the trigger is fired. As with stored procedures, when it processes a `CREATE TRIGGER` statement, c-treeSQL adds wrapper code to create a Java class and method that is invoked when the trigger is fired.

The trigger specification, however, is different from a stored procedure specification. It contains the following elements:

- The `CREATE` clause specifies the name of the trigger. c-treeSQL stores the `CREATE TRIGGER` statement in the database under *triname*. It also uses *triname* in the name of the Java class that c-treeSQL declares to wrap around the Java snippet. The class name uses the format *username\_triname\_TP*, where *username* is the user name of the database connection that issued the `CREATE TRIGGER` statement.
- The `BEFORE` or `AFTER` keywords specify the trigger action time: whether the triggered action implemented by `java_snippet` executes before or after the triggering `INSERT`, `UPDATE`, or `DELETE` statement.
- The `INSERT`, `DELETE`, or `UPDATE` keyword specifies the trigger event: which data modification command activates the trigger. If `UPDATE` is the trigger event, this clause can include an optional column list. Updates to any of the specified columns or use of a specified column in a search condition to update other values will activate the trigger. As long as a specified column is not used in either case then the trigger will not be activated. If an `UPDATE` trigger does not include the optional column list, an update statement specifying any of the table columns will activate the trigger.
- The `ON table_name` clause specifies the trigger table: the table for which the specified trigger event activates the trigger. The `ON` clause cannot specify a view or a remote table.
- The optional `REFERENCING` clause is allowed only if the trigger also specifies the `FOR EACH ROW` clause. It provides a mechanism for c-treeSQL to pass row values as input parameters to the stored procedure implemented by `java_snippet`. The code in `java_snippet` uses the `getValue()` method of the `NEWROW` and `OLDROW` objects to retrieve values of columns in rows affected by the trigger event and store them in procedure variables. See [“OLDROW and NEWROW Objects: Passing Values to Triggers”](#) for details.
- The `FOR EACH` clause specifies the frequency with which the triggered action implemented by `java_snippet` executes:
  - `FOR EACH ROW` means the triggered action executes once for each row being updated by the triggering statement. `CREATE TRIGGER` must include the `FOR EACH ROW` clause if it also includes a `REFERENCING` clause.
  - `FOR EACH STATEMENT` means the triggered action executes only once for the whole triggering statement. `FOR EACH STATEMENT` is the default.
- The `IMPORT` clause is the same as in stored procedures. It specifies standard Java classes to import.

The following example shows the elements of a trigger.

## Structure of a Trigger

Trigger action time

Trigger table

```
CREATE TRIGGER BUG_UPDATE_TRIGGER
    AFTER
    UPDATE OF (STATUS, PRIORITY) Trigger event
    ON BUG_IN
    REFERENCING OLDROW, NEWROW
    FOR EACH ROW
    IMPORT
        import java.sql*;      } Import clause
```

**Trigger  
 Specificati  
 on**

```
BEGIN
.
.
.
END
```

**Trigger  
 Body**

## Triggers vs. Stored Procedures vs. Constraints

Triggers are identical to stored procedures in many respects. There are three main differences:

- Triggers are automatic. When the trigger event (an `INSERT`, `UPDATE`, or `DELETE` statement) affects the specified table (and, optionally in `UPDATE` operations, the specified columns), the Java code contained in the body of the trigger executes. Stored procedures, on the other hand, must be explicitly invoked by an application or another procedure.
- Triggers cannot have output parameters or a result set. Since triggers are automatic, there is no calling application to process any output they may generate. The practical consequence of this is that the Java code in the trigger body cannot invoke methods of the `DhSQLResultSet` class.
- Triggers have limited input parameters. The only possible input parameters for triggers are values of columns in the rows affected by the trigger event. If the trigger includes the `REFERENCING` clause, c-treeSQL passes the values (either as they existed in the database or are specified in the `INSERT` or `UPDATE` statement) of each row affected. The Java code in the trigger body can use those values in its processing by invoking the `getValue()` method of the `OLDROW` and `NEWROW` objects (see [“OLDROW and NEWROW Objects: Passing Values to Triggers”](#)).

The automatic nature of triggers make them well-suited for enforcing referential integrity. In this regard, they are like constraints, since both triggers and constraints can help insure that a value stored in the foreign key of a table must either be null or be equal to some value in the matching unique or primary key of another table.

However, triggers differ from constraints in the following ways:

- Triggers are active, while constraints are passive. While constraints prevent updates that violate referential integrity, triggers perform explicit actions in addition to the update operation.
- Triggers can do much more than enforce referential integrity. Because they are passive, constraints are limited to preventing updates in a narrow set of conditions. Triggers are more flexible. The following section outlines some common uses for triggers.

## Typical Uses for Triggers

Typical uses for triggers include combinations of the following:

### Cascading deletes

A delete operation on one table causes additional rows to be deleted from other tables that are related to the first table by key values. This is an active way of enforcing referential integrity that a table constraint enforces passively.

### Cascading updates

An update operation on one table causes additional rows to be updated in other tables that are related to the first table by key values. These updates are commonly limited to the key fields themselves. This is an active way of enforcing referential integrity that a table constraint enforces passively.

### Summation updates

An update operation in one table causes a value in a row of another table to be updated by being increased or decreased.

### Automatic archiving

A delete operation on one table creates an identical row in an archive table that is not otherwise used by the database.

## 4.3 OLDROW and NEWROW Objects: Passing Values to Triggers

The `OLDROW` and `NEWROW` objects provide a mechanism for c-treeSQL to pass row values as input parameters to the stored procedure in a trigger that executes once for each affected row. If the `CREATE TRIGGER` statement contains the `REFERENCING` clause, the c-treeSQL Server implicitly instantiates an `OLDROW` or `NEWROW` object (or both, depending on the arguments to the `REFERENCING` clause) when it creates the Java class.

This allows the Java code in the snippet to use the `getValue()` method of those objects to retrieve values of columns in rows affected by the trigger event and store them in procedure variables:

- The `OLDROW` object contains values of a row as it exists in the database before an update or delete operation. It is instantiated when triggers specify an `UPDATE...REFERENCING OLDROW` or `DELETE...REFERENCING OLDROW` clause. It is meaningless and not available for insert operations.
- The `NEWROW` object contains values of a row as specified in an `INSERT` or `UPDATE` statement. It is instantiated when triggers specify an `UPDATE...REFERENCING NEWROW` or `INSERT...REFERENCING NEWROW` clause. It is meaningless and not available for delete operations.

`UPDATE` is the only triggering statement that allows both `NEWROW` and `OLDROW` in the `REFERENCING` clause.

Triggers use the `OLDROW.getValue()` and `NEWROW.getValue()` methods to assign a value from a row being modified to a procedure variable. The format and arguments for `getValue()` are the same as in other c-treeSQL Java classes:

```
getValue ( col_num , data type ) ;
```

- `col_num` is an integer that specifies which column affected row is of interest. `getValue()` retrieves the value in the column denoted by `col_num` (1 denotes the first column of the result set, 2 denotes the second, and so on).

- *data\_type* specifies the required c-treeSQL type of the returned column value (see "Implicit Data Type Conversion Between SQL and Java Types" for details on how c-treeSQL data types map to Java data types).

The following example shows an excerpt from a trigger that uses `getValue()` to assign values from both OLDROW and NEWROW objects.

### Using `getValue()` to Process Row Values Within Triggers

```
CREATE TRIGGER BUG_UPDATE_TRIGGER
AFTER UPDATE OF (STATUS, PRIORITY) ON BUG_INFO
REFERENCING OLDROW, NEWROW
FOR EACH ROW

IMPORT
    import java.sql.* ;
BEGIN
    try
    {
        // column number of STATUS is 10
        String old_status, new_status;

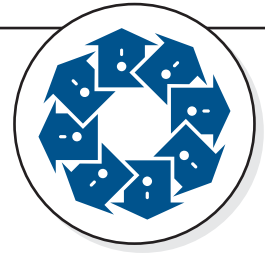
        old_status = (String)OLDROW.getValue(10,CHAR);
        new_status = (String)NEWROW.getValue(10,CHAR);

        if ((old_status.CompareTo("OPEN") == 0) &&
            (new_status.CompareTo("FIXED") == 0))
        {
            // If STATUS has changed from OPEN to FIXED
            // increment the bugs_fixed_cnt by 1 in the
            // row corresponding to current month
            // and current year
            SQLStatement update_stmt = new SQLStatement (
                " update BUG_STATUS set bugs_fixed_cnt = bugs_fixed_cnt + 1 "
                " where month = ? and year = ?");
            update_stmt.execute();
        }
    }
    catch(DhSQLException e)
    {throw e;}
END
```

### Restrictions on creating Triggers

Triggers can be created only on user tables. An attempt to create triggers on system tables (systrigger, systables, sysprocedures etc.) results in an error: (error **(20158)** : Can't create triggers on system tables).





# Using User Defined Scalar Functions

## 5.1 Introduction

Scalar functions are an integral part of the support provided by the c-treeSQL Server for query expression. The c-treeSQL Server provides several built-in scalar functions that transform data in different ways. Sometimes, however, there is a need for a custom-transformation of the data- a transformation that is not done by any of the provided functions. This problem is solved by the concept of a user-defined scalar function (UDF) - a scalar function that is defined by the user. c-treeSQL UDFs allow the user to define their own functions to transform data in some custom manner.

User Defined Scalar Functions are an extension to the existing built-in scalar functions and return a single value each time one is invoked. These functions can be used in queries in the same way that system defined scalar functions are used.

The user defines functions by creating Java Stored Functions, modules written in Java that are similar to the ones written for stored procedures and triggers. The java code snippet contained in the User Defined Scalar Function definition is processed by the c-treeSQL Server into a Java class definition and stored in the database in text and compiled form. User Defined Scalar Functions can be created, executed and dropped using ISQL, ODBC and JDBC.

## 5.2 Create Function

User Defined Scalar Functions are created using the `CREATE FUNCTION` command. User Defined Scalar Functions can be created by any user with resource privilege.

The syntax for the `CREATE FUNCTION` statement is:

```
CREATE FUNCTION [ owner_name.]function_name
  ( [parameter_decl , ...] )
  RETURNS (data_type)
  [ IMPORT
      java_import_clause ]
  BEGIN
    java_snippet
  END
parameter_decl ::
  [ IN ] parameter_name data_type
```

When creating stored procedures, the key words `BEGIN`, `RETURNS`, `IMPORT` and `END` should start at the first column of the line. The function name is limited to 64 characters

## 5.3 Description

The `CREATE FUNCTION` command creates a new User Defined Scalar Function in the current database. `function_name` is the name that is to be assigned to the function. Different owners can have functions with the same name in the same database.

Each *parameter\_decl* is of the form:

```
[ARG_TYPE] ARG_NAME DATA_TYPE
```

and is separated by a comma (',')

- *ARG\_TYPE* is optional. When specified it can only be IN.
- *ARG\_NAME* specifies the name of the argument to the function. The function can access the values through the corresponding Java variables.
- *DATA\_TYPE* is any valid supported c-treeSQL data type which specifies the type of the argument.

The return value of the function is specified with the `RETURNS` key word followed by the *data\_type* declaration.

The Java `IMPORT` statements which are needed for the correct functioning of the java snippet (the code between `BEGIN` and `END` -- see below) need to be placed in the section between the `IMPORT` and `BEGIN` keywords. This section need not be present if there are no import statements.

The function cannot be created without specifying all the import statements needed by the snippet. (For details refer to any Java manual or book)

The Function body, referred to as the Java snippet, needs to be placed between the `BEGIN` and `END` keywords. Any valid Java code other than class or function definitions can be present here. The snippet must have a return statement.

**Note:** Each of the keywords `IMPORT`, `BEGIN` and `END` need to be upper case and be present on a separate line. Once the `IMPORT` or `BEGIN` are specified, the statement is terminated by an `END` but not a semicolon (;).

### Creating a User Defined Scaler Function

```
ISQL> CREATE FUNCTION str_cat(IN org_string VARCHAR(20), IN string_to_concat VARCHAR(20))
RETURNS VARCHAR(40)
IMPORT
import java.math.*;
BEGIN
    String new_str = org_string + string_to_concat ;
    return new_str;
END
```

The above example creates a User Defined Scalar Function, *str\_cat*, that takes two input arguments and returns the concatenated string.

## 5.4 Invoking User Defined Scalar Functions

User Defined Scalar Function are a type of c-treeSQL expression that return a value based on the argument(s) supplied. User Defined Scalar Functions are invoked in exactly the same manner as built in scalar functions.

User Defined Scalar Functions can be used in the `SELECT` list or in the `WHERE` clause. They can be used as parameters of other scalar functions or in any expression. The parameter passed to a user defined scalar function can be a literal, field reference or any expression

## With Constants

### Example with Constants

```
SELECT str_cat('abcd','efgh') FROM syscalctable;
```

```
STR_CAT(ABCD,EFGH)
-----
abcdefghijkl
1 record selected
```

## With Constants and Column References

### Example with Constants and Column References

```
SELECT empfname, str_cat(empfname, emplname) FROM emp WHERE str_cat('mary', 'john') = 'maryjohn';
```

```
EMPFNAME          STR_CAT(EMPFNAME,EMPLNAME)
-----
Mary              MaryJohn
1 records selected
```

## With parameter reference (ODBC/JDBC)

### Example with Parameter References

```
SELECT str_cat(?, ?) FROM emp
SELECT * FROM emp WHERE str_cat(?, ?) = 'MaryJohn'
```

## 5.5 Drop Function

Deletes a User Defined Scalar Function.

### Syntax

```
DROP FUNCTION function_name
```

### Dropping a User Defined Scaler Function

```
ISQL> DROP FUNCTION str_cat;
```

## 5.6 User Defined Scalar Function Security

Users issuing the CREATE FUNCTION statement must have the DBA privilege or RESOURCE privilege.

The owner or users with the DBA privilege can execute or drop any User Defined Scalar Function, and grant the EXECUTE privilege to other users.

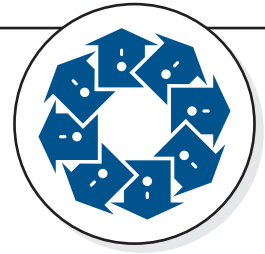
When a User Defined Scalar Function is being executed on behalf of a user with EXECUTE privilege on that User Defined Scalar Function, for the objects that are accessed by the User Defined Scalar Function, the User Defined Scalar Function owner's privileges are checked and not the user's. This enables a user to execute a User Defined Scalar Function successfully even when he does not have the privileges to directly access the objects that are accessed by the User Defined Scalar Function, so long as he has EXECUTE privilege on the User Defined Scalar Function.

## 5.7 Calling Scalar Functions from a User Defined Scalar Function

Keep in mind the c-treeSQL Server does not have a mechanism for calling a scalar function directly from a user-defined function. In order to use scalar functions, it is necessary to call the function from within a c-treeSQL statement. The c-treeSQL Server defines the special table SYSCALCTABLE which has only one row for use in situations where all the data is in the inputs.

### Example

```
CREATE FUNCTION myuser.TO_NATIVE_DATE(IN UNIX_TYPE_DATE INT)
RETURNS TIMESTAMP
IMPORT
import java.sql.*;
BEGIN
    StringBuffer query_str = new StringBuffer("SELECT TIMESTAMPADD(SQL_TSI_SECOND,");
    query_str.append(UNIX_TYPE_DATE).append(", TO_TIMESTAMP('01/01/1970 00:00:00')) FROM
SYSCALCTABLE");
    SQLCursor SelCursor = new SQLCursor(query_str.toString());
    SelCursor.open();
    SelCursor.fetch();
    return(Timestamp)SelCursor.getValue(1, TIMESTAMP);
END
```



# Java Class Reference

## 6.1 Introduction

This chapter provides reference material on the c-treeSQL Java classes and methods. The following table lists all the methods in the c-treeSQL Java classes and shows which classes declare them. Following sections are arranged alphabetically and describe each class and its methods in more detail. Some methods are common to more than one class.

### Methods in the c-treeSQL Java Classes

Method	SQLI-Statement	SQLP-Statement	SQLCursor	DhSQL-ResultSet	DhSQL-Exception	Purpose
setParam	X	X	X			Sets the value of an SQL statement's input parameter to the specified value (a literal, procedure variable, or procedure input parameter)
makeNULL	X	X	X			Sets the value of an SQL statement's input parameter to null
execute	X	X				Executes the SQL statement
rowCount	X	X	X			Returns the number of rows affected (selected, inserted, updated, or deleted) by the SQL statement
open			X			Opens the result set specified by the SELECT or CALL statement
close			X			Closes the result set specified by the SELECT or CALL statement
fetch			X			Fetches the next record in a result set
found			X			Checks whether a fetch operation returned a record

Method	SQLI-Statement	SQLP-Statement	SQLCursor	DhSQL-ResultSet	DhSQL-Exception	Purpose
wasNULL			X			Checks if the value in a fetched field is null
getValue			X			Returns the value of the specified field from the fetched row as an Object.
getParam	X	X	X			Returns the value of the specified OUT/INOUT parameter as an Object. (This method is called after executing a CALL statement).
registerOutParam	X	X	X			Registers the return type of the OUT and INOUT parameters.
set				X		Sets the field in the currently-active row of a procedure's result set to the specified value (a literal, procedure variable, or procedure input parameter)
makeNULL				X		Sets a field of the currently-active row in a procedure's result set to null
insert				X		Inserts the currently-active row into a procedure's result set
getDiagnostics					X	Returns the specified detail of an error message
log	X	X	X	X	X	Writes a message to the file sql_server.log. Inherited by all the c-treeSQL classes. See <a href="#">"Debugging Stored Procedures"</a> .
err	X	X	X	X	X	Writes a message to the file sql_server.log. Inherited by all the c-treeSQL classes. See <a href="#">"Debugging Stored Procedures"</a> .

## 6.2 DhSQLException

### Description

The *DhSQLException* class extends the general *java.lang.Exception* class to provide detail about errors in c-treeSQL statement execution. Any such errors raise an exception with an argument that is an *SQLException* class object. The [getDiagnostics\(\)](#) method retrieves details of the error.

## Constructors

```
public DhSQLException(int ecode, String errMsg)
```

## Parameters

*ecode*

The error number associated with the exception condition.

*errMsg*

The error message associated with the exception condition.

The following example illustrates the **DhSQLException()** constructor to create an exception object called *excep*. It then throws the *excep* object under all conditions.

## Example

```
CREATE PROCEDURE sp1_02()
BEGIN
  // raising exception
  DhSQLException excep = new DhSQLException(666,new String
    ("Entered the sp1_02 procedure"));
  if (true)
    throw excep;
END
```

## DhSQLException.getDiagnostics

Returns the requested detail about an exception.

## Format

```
public String getDiagnostics(int diagType)
```

## Returns

A string containing the information specified by the *diagType* parameter, as shown in the table below.

## Parameters

*diagType*

One of the values shown in the following table.

## Argument Values for DhSQLException.getDiagnostics

Argument Value	Returns
<i>RETURNED_SQLSTATE</i>	The <i>SQLSTATE</i> returned by execution of the previous SQL statement.
<i>MESSAGE_TEXT</i>	The condition indicated by <i>RETURNED_SQLSTATE</i> .
<i>CLASS_ORIGIN</i>	Not currently used. Always returns null.
<i>SUBCLASS_ORIGIN</i>	Not currently used. Always returns null.

## Throws

DhSQLException

## Example

```
CREATE PROCEDURE test_proc()
BEGIN
  String errstate;
  String errmesg;
  try
  {
    SQLStatement insert_cust = new SQLStatement ( "INSERT
    INTO customer VALUES (1,2) ");
    insert_cust.execute();
  }
  catch (DhSQLException e)
  {
    errstate = e.getDiagnostics (
    DhSQLException.RETURNED_SQLSTATE) ;
    errmesg = e.getDiagnostics (
    DhSQLException.MESSAGE_TEXT) ;
  }
END
```

## 6.3 DhSQLResultSet

### Description

Methods of the *DhSQLResultSet* class populate a result set that the stored procedure returns to the application that called the procedure.

The Java code in a stored procedure does not explicitly create *DhSQLResultSet* objects. Instead, when the SQL engine creates a Java class from a `CREATE PROCEDURE` statement that contains a `RESULT` clause, it implicitly instantiates an object of type *DhSQLResultSet*, and calls it *SQLResultSet*.

Procedures invoke methods of the `SQLResultSet()` instance to populate fields and rows of the result set.

### Constructors

No explicit constructor

### Parameters

None

### Throws

DhSQLException

### DhSQLResultSet.insert

Inserts the currently-active row into a procedure's result set.

### Format

```
public void insert()
```

## Returns

None

## Parameters

None

## Throws

DhSQLException

## Example

```
CREATE PROCEDURE get_sal2 ()
RESULT (
    empname CHAR(20),
    empsal   NUMERIC,
)
IMPORT
import java.math.BigDecimal;
BEGIN
    StringBuffer ename = new StringBuffer (20) ;
    BigDecimal esal = new BigDecimal () ;
    SQLCursor empcursor = new SQLCursor (
        "SELECT name, sal FROM emp " ) ;

    empcursor.open () ;
    do
    {
        empcursor.fetch () ;
        if (empcursor.found ())
        {
            ename = (String)empcursor.getValue (1, CHAR);
            esal = (BigDecimal)empcursor.getValue (2, NUMERIC);

            SQLResultSet.set (1, ename);
            SQLResultSet.set (2, esal);
            SQLResultSet.insert ();
        }
    } while (empcursor.found () ) ;
    empcursor.close () ;
END
```

**Note:** The resultset parameter *empname* in the procedure above should be of type CHAR in ANSI versions of c-treeSQL.

## DhSQLResultSet.makeNULL

Sets a field of the currently-active row in a procedure's result set to null. This method is redundant with using the [DhSQLResultSet.set\(\)](#) method to set a procedure result-set field to null.

## Format

```
public void makeNULL(int field)
```

## Returns

None

## Parameters

*field*

An integer that specifies which field of the result-set row to set to null (1 denotes the first field in the row, 2 denotes the second, and so on).

## Throws

DhSQLException

## Example

```
CREATE PROCEDURE test_makeNULL2 (  
    IN char_in CHAR(20))  
RESULT ( res_char CHAR(20) , res_vchar VARCHAR(30))  
BEGIN  
    SQLResultSet.set(1,char_in);  
    SQLResultSet.makeNULL(2);  
END
```

**Note:** The input parameter *char\_in* in the procedure above should be of type CHAR in ANSI versions of c-treeSQL.

## DhSQLResultSet.set

Sets the field in the currently-active row of a procedure's result set to the specified value (a literal, procedure variable, or procedure input parameter).

## Format

```
public void set(int field, Object val)
```

## Returns

None

## Parameters

*field*

An integer that specifies which field of the result-set row to set to the value specified by *val* (1 denotes the first field in the row, 2 denotes the second, and so on).

*val*

A literal or the name of a variable or input parameter that contains the value to be assigned to the field.

## Throws

DhSQLException

## Example

```
CREATE PROCEDURE get_sal2 ()  
RESULT (  
    empname CHAR(20),  
    empsal NUMERIC,  
)  
BEGIN
```

```

StringBuffer ename = new StringBuffer (20) ;
BigDecimal esal = new BigDecimal () ;
SQLCursor empcursor = new SQLCursor (
    "SELECT name, sal FROM emp " ) ;

empcursor.open () ;
do
{
    empcursor.fetch () ;
    if (empcursor.found ())
    {
        ename = (StringBuffer)empcursor.getValue (1, CHAR) ;
        esal = (BigDecimal)empcursor.getValue (2, NUMERIC) ;

        SQLResultSet.set (1, ename) ;
        SQLResultSet.set (2, esal) ;
        SQLResultSet.insert () ;
    }
} while (empcursor.found ()) ;
empcursor.close () ;
END

```

**Note:** The resultset parameter *empname* in the procedure above should be of type CHAR in ANSI versions of c-treeSQL.

## 6.4 SQLCursor

### Description

Methods of the *SQLCursor* class retrieve rows of data from a database or another stored procedure's result set.

### Constructors

*SQLCursor* (String statement)

### Parameters

*statement*

A c-treeSQL statement that generates a result set. Enclose the statement in double quotes. The c-treeSQL statement is either a *SELECT* or *CALL* statement:

- A *SELECT* statement queries the database and returns data that meets the criteria specified by the query expression in the *SELECT* statement.
- A *CALL* statement invokes another stored procedure that returns a result set specified by the *RESULT* clause of the *CREATE PROCEDURE* statement.

### Throws

*DhSQLException*

The following excerpt from a stored procedure instantiates an *SQLCursor* object called *cust\_cursor* that retrieves data from a database table.

### Example

```
SQLCursor empcursor = new SQLCursor ("SELECT name, sal FROM emp " ) ;
```

The following excerpt from a stored procedure instantiates an *SQLCursor* object called *cust\_cursor* that calls another stored procedure.

### Example with Nested Stored Procedure

```
SQLCursor cust_cursor = new SQLCursor ("CALL get_customers (?) ") ;
```

## SQLCursor.close

Closes the result set specified by the *SELECT* or *CALL* statement.

### Format

```
public void close()
```

### Returns

None

### Parameters

None

### Throws

DhSQLException

### Example

```
.  
. .  
. . .  
    if (cust_cursor.Found ())  
    {  
        cust_cursor.getValue (1, INTEGER);  
        cust_cursor.getValue (2, CHAR) ;  
    }  
    else  
        break;  
}  
  
cust_cursor.close () ;
```

## SQLCursor.fetch

Fetches the next record in a result set, if there is one.

### Format

```
public void fetch()
```

### Returns

None

## Parameters

None

## Throws

DhSQLException

## Example

```
for (;;)
{
    cust_cursor.fetch ();
    if (cust_cursor.found ())
    {
        cust_cursor.getValue (1, INTEGER);
        cust_cursor.getValue (2, CHAR) ;
    }
    else
        break;
}
```

## SQLCursor.found

Checks whether a fetch operation returned a record.

## Format

```
public boolean found ()
```

## Returns

True if the previous call to [fetch\(\)](#) returned a record, false otherwise.

## Parameters

None

## Throws

DhSQLException

## Example

```
for (;;)
{
    cust_cursor.fetch ();
    if (cust_cursor.found ())
    {
        cust_cursor.getValue (1, INTEGER);
        cust_cursor.getValue (2, CHAR) ;
    }
    else
        break;
}
```

## SQLCursor.getParam

When a procedure is called from another procedure, this method returns the value of the specified OUT/INOUT parameter of the called procedure as a Java object. This returned object must be typecasted to the appropriate type. This method is common to the *SQLCursor*, *SQLStatement*, and *SQLPStatement* classes.

### Format

```
public Object getParam(int field, short dType)
```

### Returns

Returns the Object specified by the field value.

### Parameters

*field*

An integer that specifies which argument value of the called procedure is to be returned (1 denotes the first parameter, 2 denotes the second, and so on).

If the specified parameter does not exist, the c-treeSQL Server returns an error:

```
(error(-20145): Invalid field reference.)
```

*dType*

The expected data type of the returning parameter.

### Throws

*DhSQLException*

### Example

```
CREATE PROCEDURE swap_proc(IN param1 INTEGER,
                          OUT param2 INTEGER,
                          INOUT param3 INTEGER
                          )
BEGIN
    param2 = param3;
    param3 = param1;
END

CREATE PROCEDURE call_swap_proc()
BEGIN
    Integer val1 = new Integer(10);
    Integer val2;
    Integer val3 = new Integer(20);
    SQLCursor tmp_cur = new SQLCursor("CALL swap_proc(?,?,?)");
    tmp_cur.registerOutParam(2, INTEGER);
    tmp_cur.registerOutParam(3, INTEGER);
    tmp_cur.setParam(1, val1);
    tmp_cur.setParam(3, val3);
    tmp_cur.open();
    val2 = (Integer)tmp_cur.getParam(2, INTEGER);
    val3 = (Integer)tmp_cur.getParam(3, INTEGER);
    // process the val2 and val3
    - - -
    tmp_cur.close();
END
```

## SQLCursor.getValue

Returns the value of the specified field of the fetched row of a cursor or result set as a Java object. This returned object must be typecasted to an appropriate SQL type.

### Format

```
public Object getValue(int field, short dType)
```

### Returns

Returns the Object specified by the field value.

### Parameters

#### *field*

An integer that specifies which field of the fetched record is of interest. **getValue()** retrieves the value in the currently-fetched record of the column denoted by field (1 denotes the first column of the result set, 2 denotes the second, and so on).

The value in the column denoted by field cannot be null, or the c-treeSQL Server returns an error:

```
(error(-20144): Null value fetched.)
```

This means you must always check whether a value is null before attempting to assign a value in an SQL result set to a procedure variable or output parameter. The *SQLCursor* class provides the **wasNULL()** method for this purpose. See Section "SQLCursor.wasNULL" for details.

#### *dType*

The expected data type of the return Object being returned.

### Throws

DhSQLException

### Example

```
CREATE PROCEDURE test_nulls2()
RESULT (col1 INTEGER,
       col2 INTEGER,
       col3 INTEGER)
BEGIN
  Integer pvar_int1      = new Integer(0);
  Integer pvar_int2      = new Integer(0);
  Integer pvar_int3      = new Integer(0);
  SQLCursor select_t1 = new SQLCursor
  ( "SELECT c1, c2, c3 from t1" );

  select_t1.open();
  select_t1.fetch();
  while ( select_t1.found() )
  {
    // Assign values from the current row of the SQL result set
    // to the pvar_intx procedure variables. Must first check
    // whether the values fetched are null: if they are, must set
    // pvars explicitly to null.
    if ((select_t1.wasNULL(1)) == true)
      pvar_int1 = null;
    else
      pvar_int1 = (Integer)select_t1.getValue(1, INTEGER);
    if ((select_t1.wasNULL(2)) == true)
      pvar_int2 = null;
    else
      pvar_int2 = (Integer)select_t1.getValue(2, INTEGER);
```

```
        if ((select_t1.wasNULL(3)) == true)
            pvar_int3 = null;
        else
            pSQLResultSet.set(1,pvar_int1);
            SQLResultSet.set(2,pvar_int2);
            SQLResultSet.set(3,pvar_int3);
            SQLResultSet.insert();
            select_t1.fetch();
        }
        select_t1.close();
END
```

## SQLCursor.makeNULL

Sets the value of a c-treeSQL statement's input parameter to null. This method is common to the *SQLCursor*, *SQLStatement*, and *SQLPStatement* classes. This method is redundant with using the **setParam()** method to set a statement's input parameter to null.

### Format

```
public void makeNULL(int f)
```

### Returns

None

### Parameters

*f*

An integer that specifies which input parameter of the c-treeSQL statement string to set to null (1 denotes the first input parameter in the statement, 2 denotes the second, and so on).

### Throws

DhSQLException

### Example

```
CREATE PROCEDURE sc_makeNULL()
BEGIN
    SQLCursor select_btypes = new SQLCursor (
        "SELECT small_fld from sfns where small_fld = ? ");

    select_btypes.makeNULL(1);
    select_btypes.open();
    select_btypes.fetch();
    select_btypes.close();
END
```

## SQLCursor.open

Opens the result set specified by the *SELECT* or *CALL* statement.

### Format

```
public void open()
```

## Returns

None

## Parameters

None

## Throws

DhSQLException

## Example

```
SQLCursor empcursor = new SQLCursor (
    "SELECT name, sal FROM emp " );
empcursor.open ( ) ;
```

## SQLCursor.registerOutParam

Registers the expected type of OUT and INOUT parameters. This method is common for *SQLCursor*, *SQLStatement* and *SQLPStatement* classes.

### Format

```
public void registerOutParam(int pIndex, short dType)
```

### Returns

None

### Parameters

*pindex*

An integer that specifies which OUT/INOUT parameter is to be registered (1 denotes the first parameter, 2 denotes the second, and so on).

*dtype*

The expected type of the OUT/INOUT parameter.

**Note:** *SQLCursor* class has another form of **registerOutParam()**, which takes three arguments. The syntax of this method is:

```
public void registerOutParam(int pIndex, short dType, short scale)
```

*pIndex* and *dType* are the same as the two arguments in **SQLCursor.registerOutParam()** method. The scale specifies the number of digits after the decimal point.

This method is not implemented in c-treeSQL and invocation of this method results in error:

```
"Scale for registerOutParam not implemented".
```

### Throws

DhSQLException

### Example

```
CREATE PROCEDURE register_proc()
BEGIN
  // cust_proc is a procedure with an IN, OUT and an INOUT arguments of type
  // integer, string and numeric respectively.
  SQLCursor cust_cur = new SQLCursor("call cust_proc(?,?,?)");
  cust_cur.registerOutParam(2, CHAR);
  cust_cur.registerOutParam(3, NUMERIC);
  cust_cur.open();
  // Process results
  cust_cur.close();
END
```

## SQLCursor.rowCount

Returns the number of rows affected (selected, inserted, updated, or deleted) by the c-treeSQL statement. This method is common to the *SQLCursor*, *SQLStatement*, and *SQLPStatement* classes.

### Format

```
public int rowCount()
```

### Returns

An integer indicating the number of rows.

### Parameters

None

### Throws

DhSQLException

The following example uses the **rowCount()** method of the *SQLStatement* class. It nests the method invocation within **SQLResultSet.set()** to store the number of rows affected (1, in this case) in the procedure's result set.

### Example

```
CREATE PROCEDURE sis_rowCount()
RESULT ( ins_recs BIGINT )
BEGIN
  SQLStatement insert_test103 = new SQLStatement (
    "INSERT INTO test103 (fld1) values (17)");

  insert_test103.execute();
  SQLResultSet.set(1,new Long(insert_test103.rowCount()));
  SQLResultSet.insert();
END
```

## SQLCursor.setParam

Sets the value of a c-treeSQL statement's input parameter to the specified value (a literal, procedure variable, or procedure input parameter). This method is common to the *SQLCursor*, *SQLStatement*, and *SQLPStatement* classes.

## Format

```
public void setParam(int f, Object val)
```

## Returns

None

## Parameters

*f*

An integer that specifies which parameter marker in the c-treeSQL statement is to receive the value (1 denotes the first parameter marker, 2 denotes the second, and so on).

*val*

A literal or the name of a variable or input parameter that contains the value to be assigned to the parameter marker.

## Throws

DhSQLException

## Example

```
CREATE PROCEDURE sps_setParam()
BEGIN
    // Assign local variables to be used as SQL input parameter references
    Integer ins_fld_ref    = new Integer(1);
    Integer ins_small_fld = new Integer(3200);
    Integer ins_int_fld   = new Integer(21474);
    Double  ins_doub_fld  = new Double(1.797E+30);
    StringBuffer ins_char_fld = new StringBuffer("Athula");
    StringBuffer ins_vchar_fld = new StringBuffer("Scientist");
    Float   ins_real_fld  = new Float(17);

    SQLStatement insert_sfns1 = new SQLStatement ("INSERT INTO sfns"
        + "(fld_ref,small_fld,int_fld,doub_fld,char_fld,vchar_fld)"
        + "values (?, ?, ?, ?, ?, ?)");

    insert_sfns1.setParam(1,ins_fld_ref);
    insert_sfns1.setParam(2,ins_small_fld);
    insert_sfns1.setParam(3,ins_int_fld);
    insert_sfns1.setParam(4,ins_doub_fld);
    insert_sfns1.setParam(5,ins_char_fld);
    insert_sfns1.setParam(6,ins_vchar_fld);
    insert_sfns1.execute();
END
```

## SQLCursor.wasNULL

Checks if the value in a fetched field is null.

## Format

```
public boolean wasNULL(int field)
```

## Returns

True if the field is null, false otherwise.

## Parameters

### *field*

An integer that specifies which field of the fetched record is of interest (1 denotes the first column of the result set, 2 denotes the second, and so on). **wasNULL()** checks whether the value in the currently-fetched record of the column denoted by *field* is null.

## Throws

DhSQLException

## Example

```
CREATE PROCEDURE test_wasNULL()
BEGIN
    int small_sp = new Integer(0);
    SQLCursor select_btypes = new SQLCursor ("SELECT small_fld from sfns");
    select_btypes.open();
    select_btypes.fetch();
    if ((select_btypes.wasNULL(1)) == true)
        small_sp = null;
    else
        select_btypes.getValue(1, INTEGER);
    select_btypes.close();
END
```

## 6.5 SQLStatement

### Description

Methods of the *SQLStatement* class provide for immediate (one-time) execution of c-treeSQL statements that do not generate a result set.

**Note:** *SQLStatement* can be used to call stored procedures that do not have result sets.

### Constructors

SQLStatement (String statement)

### Parameters

#### *statement*

A c-treeSQL statement that does not generate a result set. Enclose the statement in double quotes.

### Throws

DhSQLException

### Example

```
CREATE PROCEDURE insert_customer (
    IN cust_number INTEGER,
    IN cust_name CHAR(20)
)
BEGIN
    SQLStatement insert_cust = new SQLStatement
        ("INSERT INTO customer VALUES (?,?) ");
    insert_cust.execute();
END
```

END

**Note:** The input parameter *cust\_name* in the procedure above should be of type CHAR in ANSI versions of c-treeSQL.

## SQLStatement.execute

Executes the c-treeSQL statement. This method is common to the *SQLStatement* and *SQLPStatement* classes.

### Format

```
public void execute()
```

### Returns

None

### Parameters

None

### Throws

DhSQLException

### Example

```
CREATE PROCEDURE insert_customer (  
    IN cust_number INTEGER,  
    IN cust_name CHAR(20)  
)  
BEGIN  
    SQLStatement insert_cust = new SQLStatement (  
        "INSERT INTO customer VALUES (?,?) ");  
    insert_cust.setParam (1, cust_number);  
    insert_cust.setParam (2, cust_name);  
    insert_cust.execute ();  
END
```

## SQLStatement.getParam

When a procedure is called from another procedure, this method returns the value of the specified OUT/INOUT parameter of called procedure as a Java object. This returned object must be typecasted to the appropriate type.

### Format

```
public Object getParam(int field, short dType)
```

### Returns

Returns the Object specified by the field value.

## Parameters

### *field*

An integer that specifies which argument value of the called procedure is to be returned (1 denotes the first parameter, 2 denotes the second, and so on).

If the specified parameter does not exist, the c-treeSQL Server returns an error:

```
(error(-20145): Invalid field reference.)
```

### *dType*

The expected data type of the returning parameter.

## Throws

DhSQLException

## Example

```
CREATE PROCEDURE swap_proc(IN param1 INTEGER,
                           OUT param2 INTEGER,
                           INOUT param3 INTEGER
                          )
BEGIN
    param2 = param3;
    param3 = param1;
END

CREATE PROCEDURE call_swap_proc()
BEGIN
    Integer val1 = new Integer(10);
    Integer val2;
    Integer val3 = new Integer(20);
    SQLStatement istmt = new SQLStatement("CALL swap_proc(?,?,?)");
    istmt.registerOutParam(2, INTEGER);
    istmt.registerOutParam(3, INTEGER);
    istmt.setParam(1, val1);
    istmt.setParam(3, val3);
    istmt.execute();
    val2 = (Integer)istmt.getParam(2, INTEGER);
    val3 = (Integer)istmt.getParam(3, INTEGER);
    // process the val2 and val3
END
```

## SQLStatement.makeNULL

Sets the value of a c-treeSQL statement's input parameter to null. This method is common to the *SQLCursor*, *SQLStatement*, and *SQLPStatement* classes. This method is redundant with using the [setParam\(\)](#) method to set a statement's input parameter to null.

## Format

```
public void makeNULL(int f)
```

## Returns

None

## Parameters

### *f*

An integer that specifies which input parameter of the c-treeSQL statement string to set to null (1 denotes the first input parameter in the statement, 2 denotes the second, and so on).

### Throws

DhSQLException

### Example

```
CREATE PROCEDURE sis_makeNULL()
BEGIN
    SQLStatement insert_sfns1 = new SQLStatement ("INSERT INTO sfns"
    + "(fld_ref,small_fld,int_fld,doub_fld,char_fld,vchar_fld)"
    + "values (?, ?, ?, ?, ?, ?)");
    insert_sfns1.setParam(1,new Integer(66));
    insert_sfns1.makeNULL(2);
    insert_sfns1.makeNULL(3);
    insert_sfns1.makeNULL(4);
    insert_sfns1.makeNULL(5);
    insert_sfns1.makeNULL(6);
    insert_sfns1.execute();
END
```

## SQLStatement.registerOutParam

This method registers the expected type of OUT and INOUT parameters. This method is common for *SQLCursor*, *SQLStatement* and *SQLPStatement* classes.

### Format

```
public void registerOutParam(int pIndex, short dType)
```

### Returns

None

### Parameters

*pIndex*

An integer that specifies which OUT/INOUT parameter is to be registered (1 denotes the first parameter, 2 denotes the second, and so on).

*dType*

The expected type of the OUT/INOUT parameter.

### Throws

DhSQLException

### Example

```
CREATE PROCEDURE register_proc()
BEGIN
    // cust_proc is a procedure with an IN, OUT and an INOUT arguments of type
    // integer, string and numeric respectively.
    SQLStatement istmt = new SQLStatement("call cust_proc(?,?,?)");
    istmt.registerOutParam(2, CHAR);
    istmt.registerOutParam(3, NUMERIC);
    istmt.execute();
END
```

```
// Process results  
END
```

## SQLStatement.rowCount

Returns the number of rows affected (selected, inserted, updated, or deleted) by the c-treeSQL statement. This method is common to the *SQLCursor*, *SQLStatement*, and *SQLPStatement* classes.

### Format

```
public int rowCount()
```

### Returns

An integer indicating the number of rows.

### Parameters

None

### Throws

DhSQLException

The following example uses the **rowCount()** method of the *SQLStatement* class. It nests the method invocation within **SQLResultSet.set()** to store the number of rows affected (1, in this case) in the procedure's result set.

### Example

```
CREATE PROCEDURE sis_rowCount()  
RESULT ( ins_recs BIGINT )  
BEGIN  
    SQLStatement insert_test103 = new SQLStatement (  
        "INSERT INTO test103 (fld1) values (17)");  
  
    insert_test103.execute();  
    SQLResultSet.set(1,new Long(insert_test103.rowCount()));  
    SQLResultSet.insert();  
END
```

## SQLStatement.setParam

Sets the value of a c-treeSQL statement's input parameter to the specified value (a literal, procedure variable, or procedure input parameter). This method is common to the *SQLCursor*, *SQLStatement*, and *SQLPStatement* classes.

### Format

```
public void setParam(int f, Object val)
```

### Returns

None

## Parameters

- *f* An integer that specifies which parameter marker in the c-treeSQL statement is to receive the value (1 denotes the first parameter marker, 2 denotes the second, and so on).
- *val* A literal or the name of a variable or input parameter that contains the value to be assigned to the parameter marker.

## Throws

DhSQLException

## Example

```
CREATE PROCEDURE sps_setParam()
BEGIN
    // Assign local variables to be used as SQL input parameter references
    Integer ins_fld_ref = new Integer(1);
    Integer ins_small_fld = new Integer(3200);
    Integer ins_int_fld = new Integer(21474);
    Double ins_doub_fld = new Double(1.797E+30);
    StringBuffer ins_char_fld = new StringBuffer("Athula");
    StringBuffer ins_vchar_fld = new StringBuffer("Scientist");
    Float ins_real_fld = new Float(17);

    PreparedStatement insert_sfns1 = new PreparedStatement ("INSERT INTO sfns"
+ "(fld_ref,small_fld,int_fld,doub_fld,char_fld,vchar_fld)"
+ "values (?,?,?,?,?,?,?)");

    insert_sfns1.setParam(1,ins_fld_ref);
    insert_sfns1.setParam(2,ins_small_fld);
    insert_sfns1.setParam(3,ins_int_fld);
    insert_sfns1.setParam(4,ins_doub_fld);
    insert_sfns1.setParam(5,ins_char_fld);
    insert_sfns1.setParam(6,ins_vchar_fld);
    insert_sfns1.execute();
END
```

## 6.6 SQLPStatement

### Description

Methods of the *SQLPStatement* class provide for prepared (repeated) execution of c-treeSQL statements that do not generate a result set.

**Note:** *SQLPStatement* can be used to call stored procedures that do not have result sets.

### Constructors

*SQLPStatement* (String statement)

### Parameters

*statement*

A c-treeSQL statement that does not generate a result set. Enclose the statement in double quotes.

### Throws

DhSQLException

## Example

```
SQLPStatement pstmt = new SQLPStatement (
    "INSERT INTO T1 VALUES (?, ?) " );
.
.
.
```

## SQLPStatement.execute

Executes the c-treeSQL statement. This method is common to the *SQLStatement* and *SQLPStatement* classes.

### Format

```
public void execute()
```

### Returns

None

### Parameters

None

### Throws

DhSQLException

## Example

```
CREATE PROCEDURE test_pstmt ()
BEGIN
    SQLPStatement pstmt = new SQLPStatement ( "INSERT INTO T1 VALUES (?, ?) " );
    pstmt.setParam (1, new Integer(10));
    pstmt.setParam (2, new Integer(10));
    pstmt.execute ();
    pstmt.setParam (1, new Integer(20));
    pstmt.setParam (2, new Integer(20));
    pstmt.execute ();
END
```

## SQLPStatement.getParam

When a procedure is called from another procedure, this method returns the value of the specified OUT/INOUT parameter of the called procedure as a Java object. This returned object must be typecasted to the appropriate type.

### Format

```
public Object getParam(int field, short dType)
```

### Returns

Returns the Object specified by the field value.

## Parameters

### *field*

An integer that specifies which argument value of the called procedure is to be returned (1 denotes the first parameter, 2 denotes the second, and so on).

If the specified parameter does not exist, the c-treeSQL Server returns an error:

```
(error(-20145): Invalid field reference.)
```

### *dType*

The expected data type of the returning parameter.

## Throws

DhSQLException

## Example

```
CREATE PROCEDURE swap_proc(IN param1 INTEGER,
                           OUT param2 INTEGER,
                           INOUT param3 INTEGER)
BEGIN
    param2 = param3;
    param3 = param1;
END

CREATE PROCEDURE call_swap_proc()
BEGIN
    Integer val1 = new Integer(10);
    Integer val2;
    Integer val3 = new Integer(20);
    SQLPStatement pstmt = new SQLPStatement("CALL swap_proc(?,?,?)");
    pstmt.registerOutParam(2, INTEGER);
    pstmt.registerOutParam(3, INTEGER);
    pstmt.setParam(1, val1);
    pstmt.setParam(3, val3);
    pstmt.execute();
    val2 = (Integer)pstmt.getParam(2, INTEGER);
    val3 = (Integer)pstmt.getParam(3, INTEGER);
    // process the val2 and val3
END
```

## SQLPStatement.makeNULL

Sets the value of a c-treeSQL statement's input parameter to null. This method is common to the *SQLCursor*, *SQLStatement*, and *SQLPStatement* classes. This method is redundant with using the [setParam\(\)](#) method to set a statement's input parameter to null.

### Format

```
public void makeNULL(int f)
```

### Returns

None

### Parameters

*f*

An integer that specifies which input parameter of the c-treeSQL statement string to set to null (1 denotes the first input parameter in the statement, 2 denotes the second, and so on).

## Throws

DhSQLException

## Example

```
CREATE PROCEDURE sps_makeNULL()
BEGIN
    SQLPStatement insert_sfns1 = new SQLPStatement ("INSERT INTO sfns"
        + " (fld_ref,small_fld,int_fld,doub_fld,char_fld,vchar_fld)"
        + " values (?, ?, ?, ?, ?, ?) " );
    insert_sfns1.setParam(1,new Integer(666));
    insert_sfns1.makeNULL(2);
    insert_sfns1.makeNULL(3);
    insert_sfns1.makeNULL(4);
    insert_sfns1.makeNULL(5);
    insert_sfns1.makeNULL(6);
    insert_sfns1.execute();
END
```

## SQLPStatement.registerOutParam

This method registers the expected Type of OUT and INOUT parameters. This method is common for *SQLCursor*, *SQLIStatement* and *SQLPStatement* classes.

## Format

```
public void registerOutParam(int pIndex, short dType)
```

## Returns

None

## Parameters

*pIndex*

An integer that specifies which OUT/INOUT parameter is to be registered (1 denotes the first parameter, 2 denotes the second, and so on).

*dType*

The expected type of the OUT/INOUT parameter.

## Throws

DhSQLException

## Example

```
CREATE PROCEDURE register_proc()
BEGIN
    // cust_proc is a procedure with an IN, OUT and an INOUT arguments of types
    // integer, string and numeric respectively.
    SQLPStatement pstmt = new SQLPStatement("call cust_proc(?,?,?)");
    pstmt.registerOutParam(2, CHAR);
    pstmt.registerOutParam(3, NUMERIC);
    pstmt.execute();
END
```

```
// Process results
END
```

## SQLPStatement.rowCount

Returns the number of rows affected (selected, inserted, updated, or deleted) by the SQL statement. This method is common to the *SQLCursor*, *SQLStatement*, and *SQLPStatement* classes.

### Format

```
public int rowCount()
```

### Returns

An integer indicating the number of rows.

### Parameters

None

### Throws

DhSQLException

This example uses the **rowCount()** method of the *SQLStatement* class. It nests the method invocation within **SQLResultSet.set()** to store the number of rows affected (1, in this case) in the procedure's result set.

### Example

```
CREATE PROCEDURE sis_rowCount()
RESULT ( ins_recs BIGINT )
BEGIN
    SQLStatement insert_test103 = new SQLStatement (
        "INSERT INTO test103 (fld1) values (17)");

    insert_test103.execute();
    SQLResultSet.set(1,new Long(insert_test103.rowCount()));
    SQLResultSet.insert();
END
```

## SQLPStatement.setParam

Sets the value of a c-treeSQL statement's input parameter to the specified value (a literal, procedure variable, or procedure input parameter). This method is common to the *SQLCursor*, *SQLStatement*, and *SQLPStatement* classes.

### Format

```
public void setParam(int f, Object val)
```

### Returns

None

## Parameters

*f*

An integer that specifies which parameter marker in the c-treeSQL statement is to receive the value (1 denotes the first parameter marker, 2 denotes the second, and so on).

*val*

A literal or the name of a variable or input parameter that contains the value to be assigned to the parameter marker.

## Throws

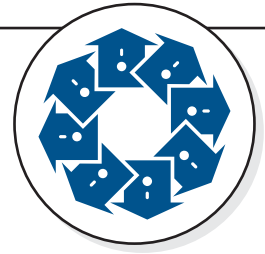
DhSQLException

## Example

```
CREATE PROCEDURE sps_setParam()
BEGIN
    // Assign local variables to be used as SQL input parameter references
    Integer ins_fld_ref = new Integer(1);
    Integer ins_small_fld = new Integer(3200);
    Integer ins_int_fld = new Integer(21474);
    Double ins_doub_fld = new Double(1.797E+30);
    StringBuffer ins_char_fld = new StringBuffer("Athula");
    StringBuffer ins_vchar_fld = new StringBuffer("Scientist");
    Float ins_real_fld = new Float(17);

    PreparedStatement insert_sfns1 = new PreparedStatement ("INSERT INTO sfns"
        + "(fld_ref,small_fld,int_fld,doub_fld,char_fld,vchar_fld)"
        + "values (?,?,?,?,,?) " );

    insert_sfns1.setParam(1,ins_fld_ref);
    insert_sfns1.setParam(2,ins_small_fld);
    insert_sfns1.setParam(3,ins_int_fld);
    insert_sfns1.setParam(4,ins_doub_fld);
    insert_sfns1.setParam(5,ins_char_fld);
    insert_sfns1.setParam(6,ins_vchar_fld);
    insert_sfns1.execute();
END
```



## Glossary

### **.NET Data Provider**

A .NET Data Provider is a bridge used for connecting ADO.NET applications to a database, executing commands and retrieving results.

### **add [an ODBC data source]**

Make a data source available to ODBC through the Add operation of the ODBC Administrator utility. Adding a data source tells ODBC where a specific database resides and which ODBC driver to use to access it. Adding a data source also invokes a setup dialog box for the particular driver so you can provide other details the driver needs to connect to the database.

### **ADMIN**

The default owner name for all system tables in a c-treeSQL database. Users must qualify references to system tables as ADMIN.tablename.

### **alias**

A temporary name for a table or column specified in the FROM clause of an SQL query expression. Also called correlation name. Derived tables and search conditions that join a table with itself must specify an alias. Once a query specifies an alias, references to the table or column must use the alias and not the underlying table or column name.

### **applet**

A special kind of Java program whose compiled class files a Java-enabled browser can download from the Internet and run.

### **ASCII**

(American Standard Code for Information Interchange) A 7-bit character set that provides 128 character combinations.

### **bytecode**

Machine-independent code generated by the Java compiler and executed by the Java interpreter.

### **cardinality**

Number of rows in a result table.

### **Cartesian product**

Also called cross-product. In a query expression, the result table generated when a FROM clause lists more than one table but specifies no join conditions. In such a case, the result table is formed by concatenating every row of every table with all other rows in all tables. Typically, Cartesian products are not useful and are slow to process.

### **client**

Generally, in client/server systems, the part of the system that sends requests to servers and processes the results of those requests.

**client character set**

A client character set is a character set used by a client application for representing SQL statements and data.

**collation**

The rules used to control how character strings in a character set compare with each other. Each character set specifies a collating sequence that defines relative values of each character for comparing, merging and sorting character strings.

**column alias**

An alias specified for a column. See alias.

**constraint**

Part of an SQL table definition that restricts the values that can be stored in a table. When you insert, delete, or update column values, the constraint checks the new values against the conditions specified by the constraint. If the value violates the constraint, it generates an error. Along with triggers, constraints enforce referential integrity by insuring that a value stored in the foreign key of a table must either be null or be equal to some value in the matching unique or primary key of another table.

**correlation name**

Another term for alias.

**cross product**

Another term for Cartesian product.

**data dictionary**

Another term for system catalog.

**data source**

See ODBC data source.

**derived table**

A virtual table specified as a query expression in the FROM clause of another query expression.

**driver manager**

See JDBC driver manager and ODBC driver manager.

**form of use**

The storage format for characters in a character set. Some character sets, such as ASCII, require one byte (octet) for each character. Others, such as Unicode, use two bytes, and are called multi-octet character sets.

**input parameter**

In a stored procedure specification, an argument that an application must pass when it calls the stored procedure. In an SQL statement, a parameter marker in the statement string that acts as a placeholder for a value that will be substituted when the statement executes.

**interface**

In Java, a definition of a set of methods that one or more objects will implement. Interfaces declare only methods and constants, not variables. Interfaces provide multiple-inheritance capabilities.

**Java snippet**

See snippet.

### **JDBC**

Java Database Connectivity: a part of the Java language that allows applications to embed standard SQL statements and access any database that implements a JDBC driver.

### **JDBC driver**

Database-specific software that receives calls from the JDBC driver manager, translates them into a form that the database can process, and returns data to the application.

### **JDBC driver manager**

A Java class that implements methods to route calls from a JDBC application to the appropriate JDBC driver for a particular JDBC URL.

### **join**

A relational operation that combines data from two tables.

### **metadata**

Data that details the structure of tables and indexes in c-tree Plus. The SQL engine stores metadata in the system catalog.

### **octet**

A group of 8 bits. Synonymous with byte, and often used in descriptions of character-set encoding format.

### **ODBC application**

Any program that calls ODBC functions and uses them to issue SQL statements. Many vendors have added ODBC capabilities to their existing Windows-based tools.

### **ODBC data source**

In ODBC terminology, a specific combination of a database system, the operating system it uses, and any network software required to access it. Before applications can access a database through ODBC, you use the ODBC Administrator to add a data source -- register information about the database and an ODBC driver that can connect to it -- for that database. More than one data source name can refer to the same database, and deleting a data source does not delete the associated database.

### **ODBC driver**

Vendor-supplied software that processes ODBC function calls for a specific data source. The driver connects to the data source, translates the standard SQL statements into syntax the data source can process, and returns data to the application. c-treeSQL includes an ODBC driver that provides access to c-tree Plus underlying the c-treeSQL Server.

### **ODBC driver manager**

A Microsoft-supplied program that routes calls from an application to the appropriate ODBC driver for a data source.

### **output parameter**

In a stored procedure specification, an argument in which the stored procedure returns a value after it executes.

### **package**

A group of related Java classes and interfaces, like a class library in C++. The Java development environment includes many packages of classes that procedures can import. The Java runtime system automatically imports the java.lang package. Stored procedures must explicitly import other classes by specifying them in the IMPORT clause of a CREATE PROCEDURE statement.

**parameter marker**

A question mark (?) in a procedure call or SQL statement string that acts as a placeholder for an input or output parameter supplied at runtime when the procedure executes. The CALL statement (or corresponding ODBC or JDBC escape clause) use parameter markers to pass parameters to stored procedures, and the PreparedStatement, SQLStatement, and SQLCursor objects use them within procedures.

**primary key**

A subset of the fields in a table, characterized by the constraint that no two records in a table may have the same primary key value, and that no fields of the primary key may have a null value. Primary keys are specified in a CREATE TABLE statement.

**procedure body**

In a stored procedure, the Java code between the BEGIN and END keywords of a CREATE PROCEDURE statement.

**procedure result set**

In a stored procedure, a set of data rows returned to the calling application. The number and data types of columns in the procedure result set are specified in the RESULT clause of the CREATE PROCEDURE statement. The procedure can transfer data from an SQL result set to the procedure result set or it can store data generated internally. A stored procedure can have only one procedure result set.

**procedure specification**

In a CREATE PROCEDURE statement, the clauses preceding the procedure body that specify the procedure name, any input and output parameters, any result set columns, and any Java packages to import.

**procedure variable**

A Java variable declared within the body of a stored procedure, as compared to a procedure input parameter or output parameter, which are declared outside the procedure body and are visible to the application that calls the stored procedure.

**query expression**

The fundamental element in SQL syntax. Query expressions specify a result table derived from some combination of rows from the tables or views identified in the FROM clause of the expression. Query expressions are the basis of SELECT, CREATE VIEW, and INSERT statements, and can be used in some expressions and search conditions.

**referential integrity**

The condition where the value stored in a database table's foreign key must either be null or be equal to some value in another table's the matching unique or primary key. SQL provides two mechanisms to enforce referential integrity: constraints specified as part of CREATE TABLE statements prevent updates that violate referential integrity, and triggers specified in CREATE TRIGGER statements execute a stored procedure to enforce referential integrity.

**repertoire**

The set of characters allowed in a character set.

**result set**

In a stored procedure, either an SQL result set or a procedure result set. More generally, another term for result table.

**result table**

A virtual table of values derived from columns and rows of one or more tables that meet conditions specified by an SQL query expression.

#### **row identifier**

Another term for tuple identifier.

#### **search condition**

The SQL syntax element that specifies a condition that is true or false about a given row or group of rows. Query expressions and UPDATE statements can specify a search condition. The search condition restricts the number of rows in the result table for the query expression or UPDATE statement. Search conditions contain one or more predicates. Search conditions follow the WHERE or HAVING keywords in SQL statements.

#### **selectivity**

The fraction of a table's rows returned by a query.

#### **server**

Generally, in client/server systems, the part of the system that receives requests from clients and responds with results to those requests.

#### **snippet**

In a stored procedure, the sequence of Java statements between the BEGIN and END keywords in the CREATE PROCEDURE (or CREATE TRIGGER) statement. The Java statements become a method in a class the SQL engine creates and submits to the Java compiler.

#### **SQL diagnostics area**

A data structure that contains information about the execution status (success, error or warning conditions) of the most recent SQL statement. The SQL-92 standard specified the diagnostics area as a standardized alternative to widely varying implementations of the SQLCA. c-treeSQL supports both the SQLCA and the SQL diagnostics area. The SQL GET DIAGNOSTICS statement returns information about the diagnostics area to an application, including the value of the SQLSTATE status parameter.

#### **SQL engine**

The core component of the c-treeSQL environment. The SQL engine receives requests from applications, processes them, and returns results.

#### **SQLCA**

SQL Communications area: A data structure that contains information about the execution status (success, error or warning conditions) of the most recent SQL statement. The SQLCA includes an SQLCODE field. The SQLCA provides the same information as the SQL diagnostics area, but is not compliant with the SQL-92 standard. c-treeSQL supports both the SQLCA and the SQL diagnostics area.

#### **SQLCODE**

An integer status parameter whose value indicates the condition status returned by the most recent SQL statement. An SQLCODE value of zero means success, a positive value means warning, and a negative value means an error status. SQLCODE is superseded by SQLSTATE in the SQL-92 standard. Applications declare either SQLSTATE or SQLCODE, or both. SQL returns the status to SQLSTATE or SQLCODE after execution of each SQL statement.

#### **SQL result set**

In a stored procedure, the set of data rows generated by an SQL statement (SELECT and, in some cases, CALL).

#### **SQLSTATE**

A 5-character status parameter whose value indicates the condition status returned by the most recent SQL statement. SQLSTATE is specified by the SQL-92 standard as a replacement for the SQLCODE status parameter (which was part of SQL-89). SQLSTATE defines many more specific error conditions than SQLCODE, which allows applications to implement more portable error handling. Applications declare either SQLSTATE or SQLCODE, or both. SQL returns the status to SQLSTATE or SQLCODE after execution of each SQL statement.

**stored procedure**

A snippet of Java source code embedded in an SQL CREATE PROCEDURE statement. The source code can use all standard Java features as well as use c-treeSQL-supplied Java classes for processing any number of SQL statements.

**system catalog**

Tables created by the SQL engine that store information about tables, columns, and indexes that make up the database. The SQL engine creates and manages the system catalog.

**system tables**

Another term for system catalog.

**tid**

Another term for tuple identifier.

**transaction**

A group of operations whose changes can be made permanent or undone only as a unit to protect against data corruption.

**trigger**

A special type of stored procedure that helps insure referential integrity for a database. Like stored procedures, triggers also contain Java source code (embedded in a CREATE TRIGGER statement) and use c-treeSQL Java classes. However, triggers are automatically invoked (“fired”) by certain SQL operations (an insert, update, or delete operation) on the trigger’s target table.

**trigger action time**

The BEFORE or AFTER keywords in a CREATE TRIGGER statement. The trigger action time specifies whether the actions implemented by the trigger execute before or after the triggering INSERT, UPDATE, or DELETE statement.

**trigger event**

The statement that causes a trigger to execute. Trigger events can be SQL INSERT, UPDATE, or DELETE statements that affect the table for which a trigger is defined.

**triggered action**

The Java code within the BEGIN END clause of a CREATE TRIGGER statement. The code implements actions to be completed when a triggering statement specifies the target table.

**tuple identifier**

A unique identifier for a tuple (row) in a table. Storage managers return a tuple identifier for the tuple that was inserted after an insert operation. The SQL engine passes a tuple identifier to the delete, update, and fetch stubs to indicate which tuple is affected. The SQL scalar function ROWID and related functions return tuple identifiers to applications.

**Unicode**

A superset of the ASCII character set that uses two bytes for each character rather than ASCII’s 7-bit representation. Able to handle 65,536 character combinations instead of ASCII’s 128, Unicode includes

alphabets for many of the world's languages. The first 128 codes of Unicode are identical to ASCII, with a second-byte value of zero.

### **URL**

In general, a Universal Resource Locator used to specify protocols and locations of items on the Internet. In JDBC, a database connection string in the form jdbc:subprotocol:subname. The c-treeSQL JDBC Driver format for database URLs is jdbc:ctree:T:host\_name:db\_name.

### **view**

A virtual table that recreates the result table specified by a SELECT statement. No data is stored in a view, but other queries can refer to it as if it were a table containing data corresponding to the result table it specifies.

### **virtual machine**

The Java specification for a hardware-independent and portable language environment. Java language compilers generate code that can execute on a virtual machine. Implementations of the Java virtual machine for specific hardware and software platforms allow the same compiled code to execute without modification.

### **virtual table**

A table of values that is not physically stored in a database, but instead derived from columns and rows of other tables. SQL generates virtual tables in its processing of query expressions: the FROM, WHERE, GROUP BY and HAVING clauses each generate a virtual table based on their input.



# Index

## A

- Additional Resources ..... 11, 22, 29, 39
- Advantages of Stored Procedures ..... 1
- Assigning Null Values from SQL Result Sets
  - SQLCursor.wasNULL Method ..... 58
- Automatic archiving..... 66

## C

- Calling Scalar Functions from a User Defined Scalar Function ..... 72
- Calling stored procedures ..... 3
  - from stored procedures ..... 60
- Calling Stored Procedures ..... 3
- Calling Stored Procedures from Stored Procedures ..... 60
- Cascading deletes ..... 66
- Cascading updates ..... 66
- Class
  - SQLCursor ..... 54
- Class reference, Java ..... 73
- Create Function..... 69
- Creating stored procedures ..... 2
- Creating Stored Procedures ..... 2
- c-treeSQL Java class methods ..... 73
- c-treeSQL Java classes ..... 41, 48

## D

- Debugging stored procedures..... 47
- Debugging Stored Procedures..... 47, 74
- Define ..... 7, 15, 25, 32
- Deleting stored procedures ..... 47
- Description ..... 70
- DhSQLException..... 74
- DhSQLException.getDiagnostics..... 75
- DhSQLResultSet..... 76
- DhSQLResultSet procedure ..... 57
- DhSQLResultSet.insert..... 76
- DhSQLResultSet.makeNULL..... 77
- DhSQLResultSet.set ..... 78
- Done..... 10, 21, 28, 38
- Drop Function..... 71

## E

- Environment variables ..... 44
- Error handling..... 59
- Executing an SQL Statement..... 52
- Executing SQL statements ..... 52

## F

- FAIRCOTM TYPOGRAPHICAL CONVENTIONS ..... vi
- Functions
  - User Defined Scaler Functions ..... 69

## G

- getValue method ..... 50
- Glossary ..... 99

## H

- Handling errors..... 59
- Handling Errors ..... 59
- Handling Null Values..... 58
- How c-treeSQL Interacts with Java..... 2

## I

- Immediate execution ..... 53
- Immediate Execution..... 53
- Implicit Data Type Conversion Between SQL and Java Types ..... 50, 51, 56, 67
- Implicit data type conversions ..... 51
- Init..... 6, 14, 24, 31
- Introduction..... 41, 63, 69, 73
- Introduction to Stored Procedures and Triggers and User Defined Functions..... 1
- Introductory Tutorial ..... 5
- Invoking stored procedures ..... 45
- Invoking Stored Procedures..... 45
- Invoking User Defined Scalar Functions ..... 70

## J

- Java class reference ..... 73
- Java Class Reference ..... 41, 73
- Java classes..... 41, 48, 73
- Java c-treeSQL interaction..... 2
- Java data type conversions..... 51
- Java environment variables ..... 44
- Java snippet ..... 42

## M

- Manage ..... 8, 18, 26, 35
- Methods..... 73
  - getValue ..... 50
  - setParam ..... 49
- Modifying and Deleting Stored Procedures ..... 47
- Modifying stored procedures..... 47

## N

- NEWROW ..... 66
- Null values..... 58

## O

- OLDROW ..... 66
- OLDROW and NEWROW Objects
  - Passing Values to Triggers ..... 64, 65, 66
- Overview ..... 1

## P

- Passing input values to SQL statements ..... 49
- Passing values from SQL result sets ..... 50
- Passing values to and from stored procedures..... 50
- Passing Values to and From Stored Procedures
  - Input and Output Parameters..... 50

Passing Values to SQL Statements.....	49
Prepared execution.....	54
Prepared Execution.....	54
Procedures	
DhSQLReslutSet.....	57
<b>Q</b>	
Quick Tour.....	5
<b>R</b>	
Record/Row Locking.....	23
registerOutParam Method	
registering the Type of OUT and INOUT Variables	
.....	56
Relationships.....	12
Restrictions on Calling Java Methods in Stored	
Procedures.....	48
Restrictions on creating Triggers.....	67
RESULT clause.....	57
Retrieving data.....	54
Retrieving Data	
the SQLCursor Class.....	54
Returning a Procedure Result Set	
the RESULT Clause and DhSQLResultSet 50, 57,	
58	
Returning a result set.....	57
<b>S</b>	
Security.....	48
setParam method.....	49
Setting SQL Statement Input Params & Procedure	
Result SetFields to Null.....	58
Setting Up Your Environment to Write Stored	
Procedures.....	44
Snippet, Java.....	42
SQL data type conversions.....	51
SQL statement execution.....	52
SQL statements, immediate execution.....	53
SQL statements, prepared execution.....	54
SQLCursor.....	79
SQLCursor class.....	54
SQLCursor.close.....	80
SQLCursor.fetch.....	80
SQLCursor.found.....	81
SQLCursor.getParam.....	82
SQLCursor.getValue.....	83
SQLCursor.makeNULL.....	84
SQLCursor.open.....	84
SQLCursor.registerOutParam.....	85
SQLCursor.rowCount.....	86
SQLCursor.setParam.....	86
SQLCursor.wasNULL.....	83, 87
SQLStatement.....	88
SQLStatement.execute.....	89
SQLStatement.getParam.....	89
SQLStatement.makeNULL.....	90
SQLStatement.registerOutParam.....	91

SQLStatement.rowCount.....	92
SQLStatement.setParam.....	92
SQLPStatement.....	93
SQLPStatement.execute.....	94
SQLPStatement.getParam.....	94
SQLPStatement.makeNULL.....	95
SQLPStatement.registerOutParam.....	96
SQLPStatement.rowCount.....	97
SQLPStatement.setParam.....	97
Stored Procedure Basics.....	42
Stored Procedure Security.....	48
Stored procedures	
basic coding.....	42
benefits.....	1
calling.....	3
calling from stored procedures.....	60
creating.....	2
debugging.....	47
deleting.....	47
environment.....	44
environment variables.....	44
invoking.....	45
modifying.....	47
overview.....	1
passing values.....	50
security.....	48
structure.....	42
transactions.....	48
using.....	41
versus constraints.....	65
versus triggers.....	65
writing.....	45
Structure of stored procedures.....	42
Structure of Stored Procedures.....	42
Structure of triggers.....	63, 70
Structure of Triggers.....	63
Summation updates.....	66
<b>T</b>	
The getValue Method	
Pass Values from SQL Result Sets to Variables50	
The setParam Method	
Pass Input Values to SQL Statements... 49, 51, 58	
Transaction Processing.....	30
Transactions.....	48
Transactions and Stored Procedures.....	48
Trigger	
basics.....	63, 69
overview.....	1
passing values.....	66
structures.....	63, 70
uses.....	66
versus constraints.....	65
versus stored procedures.....	65
Trigger Basics.....	63
Triggers vs. Stored Procedures vs. Constraints....	65

Typical Uses for Triggers ..... 66

## **U**

User Defined Scalar Function Security ..... 71

User Defined Scaler Functions ..... 69

Using stored procedures ..... 41

Using Stored Procedures ..... 41, 63

Using the c-treeSQL Java Classes ..... 48

Using Triggers ..... 63

Using User Defined Scalar Functions ..... 69

## **W**

What Is a Java Snippet? ..... 42

With Constants ..... 71

With Constants and Column References ..... 71

With parameter reference (ODBC/JDBC) ..... 71

Writing stored procedures ..... 45

Writing Stored Procedures ..... 45