

Using Procedures, Triggers, and Batches

About this chapter

Procedures and triggers store procedural SQL statements in the database for use by all applications. They enhance the security, efficiency, and standardization of databases. User-defined functions are one kind of procedure that return a value to the calling environment for use in queries and other SQL statements. Batches are sets of SQL statements submitted to the database server as a group. Many features available in procedures and triggers, such as control statements, are also available in batches.

✍ For many purposes, server-side JDBC provides a more flexible way to build logic into the database than SQL stored procedures. For information on JDBC, see "Data Access Using JDBC" on page 503.

Contents

Topic	Page
Procedure and trigger overview	222
Benefits of procedures and triggers	223
Introduction to procedures	224
Introduction to user-defined functions	229
Introduction to triggers	232
Introduction to batches	237
Control statements	239
The structure of procedures and triggers	242
Returning results from procedures	246
Using cursors in procedures and triggers	251
Errors and warnings in procedures and triggers	256
Using the EXECUTE IMMEDIATE statement in procedures	265
Transactions and savepoints in procedures and triggers	266
Some tips for writing procedures	267
Statements allowed in batches	269
Calling external libraries from procedures	271

Procedure and trigger overview

Procedures and triggers store procedural SQL statements in a database for use by all applications.

Procedures and triggers can include control statements that allow repetition (LOOP statement) and conditional execution (IF statement and CASE statement) of SQL statements.

Procedures are invoked with a CALL statement, and use parameters to accept values and return values to the calling environment. Procedures can also return result sets to the caller. Procedures can call other procedures and fire triggers.

Triggers are associated with specific database tables. They are invoked automatically (**fired**) whenever rows of the associated table are inserted, updated or deleted. Triggers do not have parameters and cannot be invoked by a CALL statement. Triggers can call procedures and fire other triggers.

User-defined functions are one kind of stored procedure that returns a single value to the calling environment. User-defined functions do not modify parameters passed to them. They broaden the scope of functions available to queries and other SQL statements.

Benefits of procedures and triggers

	<p>Procedures and triggers are defined in the database, separate from any one database application. This separation provides a number of advantages.</p>
Standardization	<p>Procedures and triggers allow standardization of any actions that are performed by more than one application program. The action is coded once and stored in the database. The applications need only CALL the procedure or fire the trigger to achieve the desired result. If the implementation of the action evolves over time, any changes are made in only one place, and all applications that use the action automatically acquire the new functionality.</p>
Efficiency	<p>When used in a database implemented on a network server, procedures and triggers are executed on the database server machine. They can access the data in the database without requiring network communication. This means that they execute faster and with less impact on network performance than if they had been implemented in an application on one of the client machines.</p> <p>When a procedure or trigger is created, it is checked for correct syntax and then stored in the system tables. The first time it is required by any application, it is retrieved from the system tables and compiled into the virtual memory of the server, and executed from there. Subsequent executions of the same procedure or trigger will result in immediate execution, since the compiled copy is retained. A procedure or trigger can be used concurrently by several applications and recursively by one application. Only one copy is compiled and kept in virtual memory.</p>
Security	<p>Procedures, including user-defined functions, execute with the permissions of the procedure owner but can be called by any user that has been granted permission to do so.</p> <p>Triggers execute under the table permissions of the owner of the associated table but are fired by any user with permission to insert, update or delete rows in the table. This means that a procedure or trigger can (and usually does) have different permissions than the user ID that invoked it. Procedures and triggers provide security by allowing users limited access to data in tables that they cannot directly examine or modify.</p>

Introduction to procedures

In order to use procedures, you need to understand how to do the following:

- ◆ Call procedures from a database application
- ◆ Create procedures
- ◆ Drop, or remove, procedures
- ◆ Control who has permission to use procedures

This section discusses each of these aspects of using procedures, and also describes some of the different uses of procedures.

Creating procedures

Procedures are created using the CREATE PROCEDURE statement. You must have RESOURCE authority in order to create a procedure.

Where you enter the statement depends on the tool you are using:

- ◆ You can create the example procedure **new_dept** by connecting to the sample database from Interactive SQL as user ID **DBA**, using password **SQL**, and typing the statement in the command window.
- ◆ You can create the example procedure by connecting to the sample database from Sybase Central, opening the Procedures folder, and clicking Add Procedure/Function Wizard. The Wizard walks you through the process. Alternatively, click Add Procedure/Function Template, which places you immediately in the last window of the Wizard, the Procedure window, in which you enter the code for the procedure.
- ◆ If you are using a tool other than Interactive SQL or Sybase Central, follow the instructions for your tool. You may need to change the command delimiter away from the semicolon before entering the CREATE PROCEDURE statement.

The following simple example creates a procedure that carries out an INSERT into the department table of the sample database, creating a new department.

```
CREATE PROCEDURE new_dept ( IN id INT,  
    IN name CHAR(35),  
    IN head_id INT )  
BEGIN  
INSERT  
INTO dba.department ( dept_id,  
    dept_name,
```

```
dept_head_id )  
VALUES ( id, name, head_id );  
END
```

☞ For a complete description of the CREATE PROCEDURE syntax, see "CREATE PROCEDURE statement" on page 403 of the book *Adaptive Server Anywhere Reference Manual*.

The body of a procedure is a **compound statement**. The compound statement starts with a BEGIN statement and concludes with an END statement. In the case of **new_dept**, the compound statement is a single INSERT bracketed by BEGIN and END statements.

☞ For more information, see "Using compound statements" on page 239.

Parameters to procedures are marked as one of IN, OUT, or INOUT. All parameters to the **new_dept** procedure are IN parameters, as they are not changed by the procedure.

Calling procedures

A procedure is invoked with a CALL statement. Procedures can be called by an application program, or they can be called by other procedures and triggers.

☞ For more information, see "CALL statement" on page 367 of the book *Adaptive Server Anywhere Reference Manual*.

The following statement calls the **new_dept** procedure to insert an Eastern Sales department:

```
CALL new_dept( 210, 'Eastern Sales', 902 );
```

After this call, you may wish to check the department table to see that the new department has been added.

The **new_dept** procedure can be called by all users who have been granted EXECUTE permission for the procedure, even if they have no permissions on the **department** table.

Dropping procedures

Once a procedure is created, it remains in the database until it is explicitly removed. Only the owner of the procedure or a user with DBA authority can drop the procedure from the database.

The following statement removes the procedure **new_dept** from the database:

```
DROP PROCEDURE new_dept
```

Permissions to execute procedures


A procedure is owned by the user who created it, and that user can execute it without permission. Permission to execute it can be granted to other users using the GRANT EXECUTE command.

For example, the owner of the procedure **new_dept** could allow **another_user** to execute **new_dept** with the statement:

```
GRANT EXECUTE ON new_dept TO another_user
```

The following statement revokes permission to execute the procedure:

```
REVOKE EXECUTE ON new_dept FROM another_user
```

 For more information on managing user permissions on procedures, see "Granting permissions on procedures" on page 583.

Returning procedure results in parameters

Procedures can return results to the calling environment in one of the following ways:

- ◆ Individual values are returned as OUT or INOUT parameters.
- ◆ Result sets can be returned.
- ◆ A single result can be returned using a RETURN statement.

This section describes how to return results from procedures as parameters.

The following procedure on the sample database returns the average salary of employees as an OUT parameter.

```
CREATE PROCEDURE AverageSalary( OUT avgsal  
    NUMERIC (20,3) )  
BEGIN  
    SELECT AVG( salary )  
    INTO avgsal  
    FROM employee;  
END
```

To run this procedure and display its output from Interactive SQL, carry out the following steps:

- 1 Connect to the sample database from Interactive SQL as user ID **DBA** using password **SQL**.
- 2 Create the procedure.

- 3 Create a variable to hold the procedure output. In this case, the output variable is numeric, with three decimal places, so create a variable as follows:

```
CREATE VARIABLE Average NUMERIC(20,3)
```

- 4 Call the procedure, using the created variable to hold the result:

```
CALL AverageSalary(Average)
```

The Interactive SQL statistics window displays the message "Procedure completed" if the procedure was created and run properly.

- 5 Look at the value of the output variable Average. The Interactive SQL Data window displays the value 49988.623 for this variable; the average employee salary.

Returning procedure results in result sets

In addition to returning results to the calling environment in individual parameters, procedures can return information in result sets. A result set is typically the result of a query. The following procedure returns a result set containing the salary for each employee in a given department:

```
CREATE PROCEDURE SalaryList ( IN department_id INT)
RESULT ( "Employee ID" INT, "Salary" NUMERIC(20,3) )
BEGIN
    SELECT emp_id, salary
    FROM employee
    WHERE employee.dept_id = department_id;
END
```

If this procedure is called from Interactive SQL, the names in the RESULT clause are matched to the results of the query and used as column headings in the displayed results.

To test this procedure from Interactive SQL, you can CALL it, specifying one of the departments of the company. The results are displayed in the Interactive SQL Data window. For example:

❖ To list the salaries of employees in the R & D department (department ID 100):

- ◆ Type the following:

```
CALL SalaryList (100)
```


Employee ID	Salary
102	45700.000
105	62000.000
160	57490.000
243	72995.000
247	48023.690

To execute a `CALL` of a procedure that returns a result set, Interactive SQL opens a cursor.

The cursor is left open after the `CALL` in case a second result set is returned. The Interactive SQL statistics window displays the plan of the `SELECT` query in the procedure and then displays the line:

Procedure is executing. Use `RESUME` to continue.

You need to execute the `RESUME` statement or the Interactive SQL `CLEAR` command from the Interactive SQL Command window before you can alter or drop the procedure.

 For more information about using cursors in procedures, see "Using cursors in procedures and triggers" on page 251.

Introduction to user-defined functions

User-defined functions are a class of procedures that return a single value to the calling environment. This section introduces creating, using, and dropping user-defined functions

Creating user-defined functions

User-defined functions are created using the CREATE FUNCTION statement. You must have RESOURCE authority in order to create a user-defined function.

The following simple example creates a function that concatenates two strings, together with a space, to form a full name from a first name and a last name.

You can create the example function **fullname** by connecting to the sample database from Interactive SQL as user ID **DBA**, using password **SQL**, and typing the statement in the command window.

If you are using a tool other than Interactive SQL or Sybase Central, you may need to change the command delimiter away from the semicolon before entering the CREATE FUNCTION statement.

```
CREATE FUNCTION fullname (firstname CHAR(30),
                          lastname CHAR(30))
RETURNS CHAR(61)
BEGIN
    DECLARE name CHAR(61);
    SET name = firstname || ' ' || lastname;
    RETURN ( name );
END
```

☞ For a complete description of the CREATE FUNCTION syntax, see "CREATE FUNCTION statement" on page 397 of the book *Adaptive Server Anywhere Reference Manual*.

The CREATE FUNCTION syntax differs slightly from that of the CREATE PROCEDURE statement. The following are distinctive differences:

- ◆ No IN, OUT, or INOUT keywords are required, as all parameters are IN parameters.
- ◆ The RETURNS clause is required to specify the data type being returned.
- ◆ The RETURN statement is required to specify the value being returned.

Calling user-defined functions

A user-defined function can be used, subject to permissions, in any place that a built-in non-aggregate function is used.

The following statement in Interactive SQL returns a full name from two columns containing a first and last name:

```
SELECT fullname (emp_fname, emp_lname)
FROM employee;
```

fullname (emp_fname, emp_lname)

Fran Whitney

Matthew Cobb

Philip Chin

...

The following statement in Interactive SQL returns a full name from a supplied first and last name:

```
SELECT fullname ('Jane', 'Smith');
```

fullname ('Jane','Smith')

Jane Smith

The **fullname** function can be used by any user who has been granted EXECUTE permission for the function.

Dropping user-defined functions

Once a user-defined function is created, it remains in the database until it is explicitly removed. Only the owner of the function or a user with DBA authority can drop a function from the database.

The following statement removes the function **fullname** from the database:

```
DROP FUNCTION fullname
```

Permissions to execute user-defined functions


A user-defined function is owned by the user who created it, and that user can execute it without permission. Permission to execute it can be granted to other users using the GRANT EXECUTE command.

For example, the creator of the function **fullname** could allow **another_user** to use **fullname** with the statement:

```
GRANT EXECUTE ON fullname TO another_user
```

The following statement revokes permission to use the function:

```
REVOKE EXECUTE ON fullname FROM another_user
```

 For more information on managing user permissions on functions, see "Granting permissions on procedures" on page 583.

Introduction to triggers

Triggers are used whenever referential integrity and other declarative constraints are not sufficient.

☞ For information on referential integrity, see "Ensuring Data Integrity" on page 347 and "CREATE TABLE statement" on page 415 of the book *Adaptive Server Anywhere Reference Manual*.

You may want to enforce a more complex form of referential integrity involving more detailed checking, or you may want to enforce checking on new data but allow legacy data to violate constraints. Another use for triggers is in logging the activity on database tables, independent of the applications using the database.

Trigger execution permissions

Triggers execute with the permissions of the owner of the associated table, not the user ID whose actions cause the trigger to fire. A trigger can modify rows in a table that a user could not modify directly.

Triggers can be defined on one or more of the following triggering actions:

Action	Description
INSERT	The trigger is invoked whenever a new row is inserted into the table associated with the trigger
DELETE	The trigger is invoked whenever a row of the associated table is deleted.
UPDATE	The trigger is invoked whenever a row of the associated table is updated.
UPDATE OF column-list	The trigger is invoked whenever a row of the associated table is updated such that a column in the <i>column-list</i> has been modified

Triggers can be defined as row-level triggers or statement-level triggers. Row-level triggers can execute BEFORE or AFTER each row modified by the triggering insert, update, or delete operation is changed. Statement-level triggers execute after the entire operation is performed.

Flexibility in trigger execution time is particularly useful for triggers that rely on referential integrity actions such as cascaded updates or deletes being carried out (or not) as they execute.

If an error occurs while a trigger is executing, the operation that fired the trigger fails. INSERT, UPDATE, and DELETE are **atomic** operations (see "Atomic compound statements" on page 241). When they fail, all effects of the statement (including the effects of triggers and any procedures called by triggers) are undone.

Creating triggers

You create triggers using the CREATE TRIGGER statement. You must have RESOURCE authority in order to create a trigger and you must have ALTER permissions on the table associated with the trigger. For information about ALTER permissions, see "Granting permissions on tables and views" on page 582. For information about RESOURCE permissions, see "Granting DBA and RESOURCE authority" on page 581.

The body of a trigger consists of a compound statement (see "Using compound statements" on page 239): a set of semicolon-delimited SQL statements bracketed by a BEGIN and an END statement.

COMMIT and ROLLBACK and some ROLLBACK TO SAVEPOINT statements are not permitted within a trigger (see "Transactions and savepoints in procedures and triggers" on page 266).

A row-level INSERT trigger example

The following trigger is an example of a row-level INSERT trigger. It checks that the birthdate entered for a new employee is reasonable:

```
CREATE TRIGGER check_birth_date
  AFTER INSERT ON Employee
  REFERENCING NEW AS new_employee
  FOR EACH ROW
  BEGIN
    DECLARE err_user_error EXCEPTION
    FOR SQLSTATE '99999';
    IF new_employee.birth_date > 'June 6, 1994' THEN
      SIGNAL err_user_error;
    END IF;
  END
```

This trigger is fired just after any row is inserted into the **employee** table. It detects and disallows any new rows that correspond to birth dates later than June 6, 1994.

The phrase REFERENCING NEW AS **new_employee** allows statements in the trigger code to refer to the data in the new row using the alias **new_employee**.

Signaling an error causes the triggering statement as well as any previous effects of the trigger to be undone.

For an INSERT statement that adds many rows to the employee table, the **check_birth_date** trigger is fired once for each new row. If the trigger fails for any of the rows, all effects of the INSERT statement are rolled back.

You can specify that the trigger fire before the row is inserted rather than after by changing the first line of the example to:

```
CREATE TRIGGER mytrigger BEFORE INSERT ON Employee
```

The REFERENCING NEW clause refers to the inserted values of the row; it is independent of the timing (BEFORE or AFTER) of the trigger.

The following CREATE TRIGGER statement defines a row-level DELETE trigger:

```
CREATE TRIGGER mytrigger BEFORE DELETE ON employee
REFERENCING OLD AS oldtable
FOR EACH ROW
BEGIN
    ...
END
```

The REFERENCING OLD clause enables the delete trigger code to refer to the values in the row being deleted using the alias **oldtable**.

You can specify that the trigger fire after the row is deleted rather than before, by changing the first line of the example to:

```
CREATE TRIGGER mytrigger AFTER DELETE ON employee
```

The REFERENCING OLD clause is independent of the timing (BEFORE or AFTER) of the trigger.

The following CREATE TRIGGER statement is appropriate for statement-level UPDATE triggers:

```
CREATE TRIGGER mytrigger AFTER UPDATE ON employee
REFERENCING NEW AS table_after_update
                OLD AS table_before_update
FOR EACH STATEMENT
BEGIN
    ...
END
```

The REFERENCING NEW and REFERENCING OLD clause allows the UPDATE trigger code to refer to both the old and new values of the rows being updated. Columns in the new row are referred to with the table alias **table_after_update** and columns in the old row are referred to with the table alias **table_before_update**.

The REFERENCING NEW and REFERENCING OLD clause has a slightly different meaning for statement-level and row-level triggers. For statement-level triggers the REFERENCING OLD or NEW aliases are table aliases, while in row-level triggers they refer to the row being altered.

A row-level
DELETE trigger
example

A statement-level
UPDATE trigger
example

Executing triggers

Triggers are executed automatically whenever an INSERT, UPDATE, or DELETE operation is performed on the table named in the trigger. A row-level trigger is fired once for each row that is affected, while a statement-level trigger is fired once for the entire statement.

When an INSERT, UPDATE, or DELETE fires a trigger, the order of operation is as follows:

- 1 Any BEFORE triggers are fired.
- 2 Any referential actions are performed.
- 3 The operation itself is performed.
- 4 Any AFTER triggers are fired.

If any of the steps encounters an error that is not handled within a procedure or trigger, the preceding steps are undone, the subsequent steps are not performed, and the operation that fired the trigger fails.

Dropping triggers

Once a trigger is created, it remains in the database until it is explicitly removed. You must have ALTER permissions on the table associated with the trigger.

The following statement removes the trigger **mytrigger** from the database:

```
DROP TRIGGER mytrigger
```

Trigger execution permissions

You cannot grant permissions to execute a trigger, as triggers are not executed by users: they are fired by Adaptive Server Anywhere in response to actions on the database. Nevertheless, a trigger does have permissions associated with it as it executes, defining its right to carry out certain actions.

Triggers execute using the permissions of the owner of the table on which they are defined, not the permissions of the user that caused the trigger to fire, and not the permissions of the user that created the trigger.

When a trigger refers to a table, it uses the group memberships of the table creator to locate tables with no explicit owner name specified. For example, if a trigger on **user_1.Table_A** references **Table_B** and does not specify the owner of **Table_B**, then either **Table_B** must have been created by **user_1** or **user_1** must be a member of a group (directly or indirectly) that is the owner of **Table_B**. If neither condition is met, a **table not found** message results when the trigger is fired.

Also, **user_1** must have permission to carry out the operations specified in the trigger.

Introduction to batches

A simple batch consists of a set of SQL statements, separated by semicolons. For example, the following set of statements form a batch, which creates an Eastern Sales department and transfers all sales reps from Massachusetts to that department.

```
INSERT
INTO department ( dept_id, dept_name )
VALUES ( 220, 'Eastern Sales' ) ;

UPDATE employee
SET dept_id = 220
WHERE dept_id = 200
AND state = 'MA' ;

COMMIT ;
```

You can include this set of statements in an application and execute them together.

Interactive SQL and batches

A list of semicolon-separated statements, such as the above, is parsed by Interactive SQL before it is sent to the server. In this case, Interactive SQL sends each statement individually to the server, not as a batch. Unless you have such parsing code in your application, the statements would be sent and treated as a batch. Putting a BEGIN and END around a set of statements causes Interactive SQL to treat them as a batch.

Many statements used in procedures and triggers can also be used in batches. You can use control statements (CASE, IF, LOOP, and so on), including compound statements (BEGIN and END), in batches. Compound statements can include declarations of variables, exceptions, temporary tables, or cursors inside the compound statement.

The following batch creates a table only if a table of that name does not already exist:

```
BEGIN
  IF NOT EXISTS (
    SELECT * FROM SYSTABLE
    WHERE table_name = 't1' ) THEN
    CREATE TABLE t1 (
      firstcol INT PRIMARY KEY,
      secondcol CHAR( 30 )
    ) ;
  ELSE
    MESSAGE 'Table t1 already exists' ;
```

```
END IF  
END
```

If you run this batch twice from Interactive SQL, it creates the table the first time you run it, and prints the message on the server message window the next time you run it.

Control statements

There are a number of control statements for logical flow and decision making in the body of the procedure or trigger, or in a batch. The following is a list of control statements available.

Control statement	Syntax
Compound statements	BEGIN [ATOMIC] statement-list END
Conditional execution: IF	IF condition THEN statement-list ELSEIF condition THEN statement-list ELSE statement-list END IF
Conditional execution: CASE	CASE expression WHEN value THEN statement-list WHEN value THEN statement-list ELSE statement-list END CASE
Repetition: WHILE, LOOP	WHILE condition LOOP statement-list END LOOP
Repetition: FOR cursor loop	FOR statement-list END FOR
Break: LEAVE	LEAVE label
CALL	CALL procname(arg, ...)

For complete descriptions of each, see the entries in "SQL Statements" on page 339 of the book *Adaptive Server Anywhere Reference Manual*

Using compound statements

A compound statement starts with the keyword BEGIN and ends with the keyword END. The body of a procedure or trigger is a **compound statement**. Compound statements can also be used in batches. Compound statements can be nested, and combined with other control statements to define execution flow in procedures and triggers or in batches.

A compound statement allows a set of SQL statements to be grouped together and treated as a unit. SQL statements within a compound statement should be delimited with semicolons.

A command delimiter is required after the first two SELECT statements. It is optional after the final statement in a statement list.

Declarations in compound statements

Local declarations in a compound statement immediately follow the BEGIN keyword. These local declarations exist only within the compound statement. The following may be declared within a compound statement:

- ◆ Variables
- ◆ Cursors
- ◆ Temporary tables
- ◆ Exceptions (error identifiers)

Local declarations can be referenced by any statement in that compound statement, or in any compound statement nested within it. Local declarations are not visible to other procedures called from the compound statement.

The following user-defined function illustrates local declarations of variables.

The **customer** table includes some Canadian customers sprinkled among those from the USA, but there is no **country** column. The user-defined function **nationality** uses the fact that the US zip code is numeric while the Canadian postal code begins with a letter to distinguish Canadian and US customers.

```
CREATE FUNCTION nationality( cust_id INT )
RETURNS CHAR( 20 )
BEGIN
    DECLARE natl CHAR(20);
    IF cust_id IN ( SELECT id FROM customer
                   WHERE LEFT(zip,1) > '9') THEN
        SET natl = 'CDN';
    ELSE
        SET natl = 'USA';
    END IF;
    RETURN ( natl );
END
```

This example declares a variable **natl** to hold the nationality string, uses a SET statement to set a value for the variable, and returns the value of the **natl** string to the calling environment.

The following query lists all Canadian customers in the **customer** table:

```
SELECT *
FROM customer
WHERE nationality(id) = 'CDN'
```

Declarations of cursors and exceptions are discussed in later sections.

Atomic compound statements

An **atomic** statement is a statement that is executed completely or not at all. For example, an UPDATE statement that updates thousands of rows might encounter an error after updating many rows. If the statement does not complete, all changes are undone. The UPDATE statement is atomic.

All noncompound SQL statements are atomic. A compound statement can be made atomic by adding the keyword **ATOMIC** after the **BEGIN** keyword.

```
BEGIN ATOMIC
  UPDATE employee
  SET manager_ID = 501
  WHERE emp_ID = 467;
  UPDATE employee
  SET birth_date = 'bad_data';
END
```

In this example, the two update statements are part of an atomic compound statement. They must either succeed or fail as one. The first update statement would succeed. The second one causes a data conversion error since the value being assigned to the **birth_date** column cannot be converted to a date.

The atomic compound statement fails and the effect of both UPDATE statements is undone. Even if the currently executing transaction is eventually committed, neither statement in the atomic compound statement takes effect.

COMMIT and ROLLBACK and some ROLLBACK TO SAVEPOINT statements are not permitted within an atomic compound statement (see "Transactions and savepoints in procedures and triggers" on page 266).

There is a case where some, but not all, of the statements within an atomic compound statement are executed. This is when an error occurs, and is handled by an exception handler within the compound statement.

☞ For more information, see "Using exception handlers in procedures and triggers" on page 261.

The structure of procedures and triggers

The body of a procedure or trigger consists of a compound statement as discussed in "Using compound statements" on page 239. A compound statement consists of a BEGIN and an END, enclosing a set of SQL statements. The statements must be delimited by semicolons.

☞ The SQL statements that can occur in procedures and triggers are described in "SQL statements allowed in procedures and triggers" on page 242.

☞ Procedures and triggers can contain control statements, which are described in "Control statements" on page 239.

SQL statements allowed in procedures and triggers

Almost all SQL statements are allowed within procedures and triggers, including the following:

- ◆ SELECT, UPDATE, DELETE, INSERT and SET VARIABLE.
- ◆ The CALL statement to execute other procedures.
- ◆ Control statements (see "Control statements" on page 239).
- ◆ Cursor statements (see "Using cursors in procedures and triggers" on page 251).
- ◆ Exception handling statements (see "Using exception handlers in procedures and triggers" on page 261).
- ◆ The EXECUTE IMMEDIATE statement.

Some SQL statements are not allowed within procedures and triggers. These include the following:

- ◆ CONNECT statement
- ◆ DISCONNECT statement.

COMMIT, ROLLBACK and SAVEPOINT statements are allowed within procedures and triggers with certain restrictions (see "Transactions and savepoints in procedures and triggers" on page 266).

☞ For details, see the **Usage** for each SQL statement in the chapter "SQL Statements" on page 339 of the book *Adaptive Server Anywhere Reference Manual*

Declaring parameters for procedures

Procedure parameters, or arguments, are specified as a list in the CREATE PROCEDURE statement. Parameter names must conform to the rules for other database identifiers such as column names. They must be a valid data types (see "SQL Data Types" on page 219 of the book *Adaptive Server Anywhere Reference Manual*), and must be prefixed with one of the keywords IN, OUT or INOUT. These keywords have the following meanings:

- ◆ **IN** The argument is an expression that provides a value to the procedure.
- ◆ **OUT** The argument is a variable that could be given a value by the procedure.
- ◆ **INOUT** The argument is a variable that provides a value to the procedure, and could be given a new value by the procedure.

Default values can be assigned to procedure parameters in the CREATE PROCEDURE statement. The default value must be a constant, which may be NULL. For example, the following procedure uses the NULL default for an IN parameter to avoid executing a query that would have no meaning:

```
CREATE PROCEDURE
CustomerProducts( IN customer_id
                  INTEGER DEFAULT NULL )
RESULT ( product_id INTEGER,
         quantity_ordered INTEGER )
BEGIN
  IF customer_id IS NULL THEN
    RETURN;
  ELSE
    SELECT product.id,
           sum( sales_order_items.quantity )
    FROM   product,
           sales_order_items,
           sales_order
    WHERE sales_order.cust_id = customer_id
    AND   sales_order.id = sales_order_items.id
    AND   sales_order_items.prod_id=product.id
    GROUP BY product.id;
  END IF;
END
```

The following statement causes the DEFAULT NULL to be assigned, and the procedure RETURNS instead of executing the query.

```
CALL customerproducts();
```

Passing parameters to procedures

You can take advantage of default values of stored procedure parameters with either of two forms of the CALL statement.

If the optional parameters are at the end of the argument list in the CREATE PROCEDURE statement, they may be omitted from the CALL statement. As an example, consider a procedure with three INOUT parameters:

```
CREATE PROCEDURE SampleProc( INOUT var1 INT
                             DEFAULT 1,
                             INOUT var2 int DEFAULT 2,
                             INOUT var3 int DEFAULT 3 )
...

```

We assume that the calling environment has set up three variables to hold the values passed to the procedure:

```
CREATE VARIABLE V1 INT;
CREATE VARIABLE V2 INT;
CREATE VARIABLE V3 INT;

```

The procedure **SampleProc** may be called supplying only the first parameter as follows:

```
CALL SampleProc( V1 )

```

in which case the default values are used for *var2* and *var3*.

A more flexible method of calling procedures with optional arguments is to pass the parameters by name. The **SampleProc** procedure may be called as follows:

```
CALL SampleProc( var1 = V1, var3 = V3 )

```

or as follows:

```
CALL SampleProc( var3 = V3, var1 = V1 )

```

Passing parameters to functions

User-defined functions are not invoked with the CALL statement, but are used in the same manner that built-in functions are. For example, the following statement uses the **fullname** function defined in "Creating user-defined functions" on page 229 to retrieve the names of employees:

❖ **To list the names of all employees:**

- ◆ Type the following:

```
SELECT fullname(emp_fname, emp_lname) AS Name
FROM employee

```


Name

Fran Whitney

Matthew Cobb

Philip Chin

Julie Jordan

Robert Breault

...

Notes

- ◆ Default parameters can be used in calling functions. However, parameters cannot be passed to functions by name.
- ◆ Parameters are passed by value, not by reference. Even if the function changes the value of the parameter, this change is not returned to the calling environment.
- ◆ Output parameters cannot be used in user-defined functions.
- ◆ User-defined functions cannot return result sets.

Returning results from procedures

Procedures can return results that are a single row of data, or multiple rows. In the former case, results can be passed back as arguments to the procedure. In the latter case, results are passed back as result sets. Procedures can also return a single value given in the RETURN statement.

☞ For simple examples of how to return results from procedures, see "Introduction to procedures" on page 224. For more detailed information, see the following sections.

Returning a value using the RETURN statement

A single value can be returned to the calling environment using the RETURN statement, which causes an immediate exit from the procedure. The RETURN statement takes the form:

```
RETURN expression
```

The value of the supplied expression is returned to the calling environment. To save the return value in a variable, an extension of the CALL statement is used:

```
CREATE VARIABLE returnval INTEGER ;  
returnval = CALL myproc() ;
```

Returning results as procedure parameters

Procedures can return results to the calling environment in the parameters to the procedure.

Within a procedure, parameters and variables can be assigned values in one of the following ways:

- ◆ The parameter can be assigned a value using the SET statement.
- ◆ The parameter can be assigned a value using a SELECT statement with an INTO clause.

Using the SET statement

The following somewhat artificial procedure returns a value in an OUT parameter that is assigned using a SET statement:

```
CREATE PROCEDURE greater ( IN a INT,  
                           IN b INT,  
                           OUT c INT)  
  
BEGIN  
  IF a > b THEN  
    SET c = a;
```

```

ELSE
    SET c = b;
END IF ;
END

```

Using single-row SELECT statements

Single-row queries retrieve at most one row from the database. This type of query is achieved by a SELECT statement with an INTO clause. The INTO clause follows the select list and precedes the FROM clause. It contains a list of variables to receive the value for each select list item. There must be the same number of variables as there are select list items.

When a SELECT statement is executed, the server retrieves the results of the SELECT statement and places the results in the variables. If the query results contain more than one row, the server returns an error. For queries returning more than one row, **cursors** must be used. For information about returning more than one row from a procedure, see "Returning result sets from procedures" on page 248.

If the query results in no rows being selected, a **row not found** warning is returned.

The following procedure returns the results of a single-row SELECT statement in the procedure parameters.

❖ To return the number of orders placed by a given customer:

- ◆ Type the following:

```

CREATE PROCEDURE OrderCount (IN customer_ID INT,
                             OUT Orders INT)
BEGIN
    SELECT COUNT(dba.sales_order.id)
        INTO Orders
    FROM dba.customer
        KEY LEFT OUTER JOIN "dba".sales_order
    WHERE dba.customer.id = customer_ID;
END

```

You can test this procedure in Interactive SQL using the following statements, which show the number of orders placed by the customer with ID 102:

```

CREATE VARIABLE orders INT;
CALL OrderCount ( 102, orders );
SELECT orders;

```

Notes

- ◆ The *customer_ID* parameter is declared as an IN parameter. This parameter holds the customer ID that is passed in to the procedure.
- ◆ The *Orders* parameter is declared as an OUT parameter. It holds the value of the orders variable that is returned to the calling environment.

- ◆ No DECLARE statement is required for the *Orders* variable, as it is declared in the procedure argument list.
- ◆ The SELECT statement returns a single row and places it into the variable *Orders*.

Returning result sets from procedures

If a procedure returns more than one row of results to the calling environment, it does so using result sets.

The following procedure returns a list of customers who have placed orders, together with the total value of the orders placed. The procedure does not list customers who have not placed orders.

```
CREATE PROCEDURE ListCustomerValue ()
RESULT ("Company" CHAR(36), "Value" INT)
BEGIN
    SELECT company_name,
           CAST( sum( sales_order_items.quantity *
                    product.unit_price)
               AS INTEGER ) AS value
    FROM customer
       INNER JOIN sales_order
       INNER JOIN sales_order_items
       INNER JOIN product
    GROUP BY company_name
    ORDER BY value DESC;
END
```

- ◆ Type the following:

```
CALL ListCustomerValue ()
```

Company	Value
Chadwicks	8076
Overland Army Navy	8064
Martins Landing	6888
Sterling & Co.	6804
Carmel Industries	6780
...	...

Notes

- ◆ The number of variables in the RESULT list must match the number of the SELECT list items. Automatic data type conversion is carried out where possible if data types do not match.

- ◆ The RESULT clause is part of the CREATE PROCEDURE statement, and is not followed by a command delimiter.
- ◆ The names of the SELECT list items do not need to match those of the RESULT list.
- ◆ When testing this procedure, Interactive SQL opens a cursor to handle the results. The cursor is left open following the SELECT statement, in case the procedure returns more than one result set. You should type RESUME to complete the procedure and close the cursor.
- ◆ Procedure result sets are modifiable, unless they are generated from a view. The user calling the procedure requires the appropriate permissions on the underlying table in order to modify procedure results. This differs from the usual permissions for procedure execution, where the procedure *owner* must have permissions on the table.

Returning multiple result sets from procedures

A procedure can return more than one result set to the calling environment. If a RESULT clause is employed, the result sets must be compatible: they must have the same number of items in the SELECT lists, and the data types must all be of types that can be automatically converted to the data types listed in the RESULT list.

The following procedure lists the names of all employees, customers, and contacts listed in the database:

```
CREATE PROCEDURE ListPeople()
RESULT ( lname CHAR(36), fname CHAR(36) )
BEGIN
    SELECT emp_lname, emp_fname
    FROM employee;
    SELECT lname, fname
    FROM customer;
    SELECT last_name, first_name
    FROM contact;
END
```

Notes

- ◆ To test this procedure in Interactive SQL, enter the following statement:

```
CALL ListPeople ()
```

You must enter a RESUME statement after each of the three result sets is displayed in the Interactive SQL Data window to continue, and then complete, the procedure.

Returning variable result sets from procedures

The RESULT clause is optional in procedures. Omitting the result clause allows you to write procedures that return different result sets, with different numbers or types of columns, depending on how they are executed.

If you are not using this feature of variable result sets, it is recommended that you employ a RESULT clause, for performance reasons.

For example, the following procedure returns two columns if the input variable is Y, but only one column otherwise:

```
CREATE PROCEDURE names( IN formal char(1))
BEGIN
  IF formal = 'y' THEN
    SELECT emp_lname, emp_fname
    FROM employee
  ELSE
    SELECT emp_fname
    FROM employee
  END IF
END
```

The use of variable result sets in procedures is subject to some limitations, depending on the interface used by the client application.

- ◆ **Embedded SQL** You must DESCRIBE the procedure call after the cursor for the result set is opened, but before any rows are returned, in order to get the proper shape of result set.

☞ For information about the DESCRIBE statement, see "DESCRIBE statement" on page 446 of the book *Adaptive Server Anywhere Reference Manual*.

- ◆ **ODBC** Variable result set procedures can be used by ODBC applications. The proper description of the variable result sets is carried out by the Adaptive Server Anywhere ODBC driver.
- ◆ **Open Client applications** Variable result set procedures can be used by Open Client applications. The proper description of the variable result sets is carried out by Adaptive Server Anywhere.
- ◆ **Interactive SQL** Interactive SQL does not support variable result set procedures, and so cannot be used for testing this feature.

Using cursors in procedures and triggers

Cursors are used to retrieve rows one at a time from a query or stored procedure that has multiple rows in its result set. A **cursor** is a handle or an identifier for the query or procedure, and for a current position within the result set.

Cursor management overview

Managing a cursor is similar to managing a file in a programming language. The following steps are used to manage cursors:

- 1 Declare a cursor for a particular SELECT statement or procedure using the DECLARE statement.
- 2 Open the cursor using the OPEN statement.
- 3 Use the FETCH statement to retrieve results one row at a time from the cursor.
- 4 Records are usually fetched until the warning Row Not Found is returned, signaling the end of the result set.
- 5 Close the cursor using the CLOSE statement.

By default, cursors are automatically closed at the end of a transaction (on COMMIT or ROLLBACK statements). Cursors that are opened using the WITH HOLD clause will be kept open for subsequent transactions until they are explicitly closed.

Cursor positioning

A cursor can be positioned at one of three places:

- ◆ On a row
- ◆ Before the first row
- ◆ After the last row

Absolute row from start		Absolute row from end
0	Before first row	-n - 1
1		-n
2		-n + 1
3		-n + 2
n - 2		-3
n - 1		-2
n		-1
n + 1	After last row	0

When a cursor is opened, it is positioned before the first row. The cursor position can be moved using the FETCH command (see "FETCH statement" on page 468 of the book *Adaptive Server Anywhere Reference Manual*). It can be positioned to an absolute position from the start or the end of the query results (using FETCH ABSOLUTE, FETCH FIRST, or FETCH LAST). It can also be moved relative to the current cursor position (using FETCH RELATIVE, FETCH PRIOR, or FETCH NEXT). The NEXT keyword is the default qualifier for the FETCH statement.

There are special **positioned** versions of the UPDATE and DELETE statements that can be used to update or delete the row at the current position of the cursor. If the cursor is positioned before the first row or after the last row, a No current row of cursor error will be returned.

Cursor positioning problems

Inserts and some updates to DYNAMIC SCROLL cursors can cause problems with cursor positioning. The server will not put inserted rows at a predictable position within a cursor unless there is an ORDER BY clause on the SELECT statement. In some cases, the inserted row will not appear at all until the cursor is closed and opened again.

With Adaptive Server Anywhere, this occurs if a temporary table had to be created to open the cursor (see "Temporary tables used in query processing" on page 640 for a description).

The UPDATE statement may cause a row to move in the cursor. This will happen if the cursor has an ORDER BY clause that uses an existing index (a temporary table is not created). Using STATIC SCROLL cursors alleviates these problems but requires more memory and processing.

Using cursors on SELECT statements in procedures

The following procedure uses a cursor on a SELECT statement. It illustrates several features of the stored procedure language. It is based on the same query used in the **ListCustomerValue** procedure described in "Returning result sets from procedures" on page 248.

```
CREATE PROCEDURE TopCustomerValue
(   OUT TopCompany CHAR(36),
    OUT TopValue INT )
BEGIN
-- 1. Declare the "error not found" exception
DECLARE err_notfound
    EXCEPTION FOR SQLSTATE '02000';
-- 2. Declare variables to hold
--    each company name and its value
DECLARE ThisName CHAR(36);
DECLARE ThisValue INT;
-- 3. Declare the cursor ThisCompany
--    for the query
DECLARE ThisCompany CURSOR FOR
SELECT company_name,
    CAST( sum( sales_order_items.quantity *
        product.unit_price ) AS INTEGER )
    AS value
FROM customer
    INNER JOIN sales_order
    INNER JOIN sales_order_items
    INNER JOIN product
GROUP BY company_name;
```

```
-- 4. Initialize the values of TopValue
SET TopValue = 0;
-- 5. Open the cursor
OPEN ThisCompany;
-- 6. Loop over the rows of the query
CompanyLoop:
LOOP
    FETCH NEXT ThisCompany
        INTO ThisName, ThisValue;
    IF SQLSTATE = err_notfound THEN
        LEAVE CompanyLoop;
    END IF;
    IF ThisValue > TopValue THEN
        SET TopCompany = ThisName;
        SET TopValue = ThisValue;
    END IF;
END LOOP CompanyLoop;
-- 7. Close the cursor
CLOSE ThisCompany;
END
```

Notes

The **TopCustomerValue** procedure has the following notable features:

- ◆ The "error not found" exception is declared. This exception is used later in the procedure to signal when a loop over the results of a query has completed.
☞ For more information about exceptions, see "Errors and warnings in procedures and triggers" on page 256.
- ◆ Two local variables **ThisName** and **ThisValue** are declared to hold the results from each row of the query.
- ◆ The cursor **ThisCompany** is declared. The SELECT statement produces a list of company names and the total value of the orders placed by that company.
- ◆ The value of **TopValue** is set to an initial value of 0, for later use in the loop.
- ◆ The **ThisCompany** cursor is opened.
- ◆ The LOOP statement loops over each row of the query, placing each company name in turn into the variables **ThisName** and **ThisValue**. If **ThisValue** is greater than the current top value, **TopCompany** and **TopValue** are reset to **ThisName** and **ThisValue**.
- ◆ The cursor is closed at the end of the procedure.

The LOOP construct in the **TopCompanyValue** procedure is a standard form, exiting after the last row is processed. You can rewrite this procedure in a more compact form using a FOR loop. The FOR statement combines several aspects of the above procedure into a single statement.

```
CREATE PROCEDURE TopCustomerValue2(
    OUT TopCompany CHAR(36),
    OUT TopValue INT )
BEGIN
    -- Initialize the TopValue variable
    SET TopValue = 0;
    -- Do the For Loop
    CompanyLoop:
    FOR CompanyFor AS ThisCompany
    CURSOR FOR
    SELECT company_name AS ThisName ,
        CAST( sum( sales_order_items.quantity *
            product.unit_price ) AS INTEGER )
        AS ThisValue
    FROM customer
        INNER JOIN sales_order
        INNER JOIN sales_order_items
        INNER JOIN product
    GROUP BY ThisName
    DO
        IF ThisValue > TopValue THEN
            SET TopCompany = ThisName;
            SET TopValue = ThisValue;
        END IF;
    END FOR CompanyLoop;
END
```

Errors and warnings in procedures and triggers

After an application program executes a SQL statement, it can examine a **return code**. This return code indicates whether the statement executed successfully or failed and gives the reason for the failure. The same mechanism can be used to indicate the success or failure of a CALL statement to a procedure.

Error reporting uses either the SQLCODE or SQLSTATE status descriptions. For full descriptions of SQLCODE and SQLSTATE error and warning values and their meanings, see "Database Error Messages" on page 581 of the book *Adaptive Server Anywhere Reference Manual*. Whenever a SQL statement is executed, a value is placed in special procedure variables called SQLSTATE and SQLCODE. That value indicates whether or not there were any unusual conditions encountered while the statement was being performed. You can check the value of SQLSTATE or SQLCODE in an IF statement following a SQL statement, and take actions depending on whether the statement succeeded or failed.

For example, the SQLSTATE variable can be used to indicate if a row is successfully fetched. The **TopCustomerValue** procedure presented in section "Using cursors on SELECT statements in procedures" on page 253 used the SQLSTATE test to detect that all rows of a SELECT statement had been processed.

☞ Possible values for the SQLSTATE and SQLCODE variables are listed in "Database Error Messages" on page 581 of the book *Adaptive Server Anywhere Reference Manual*.

Default error handling in procedures and triggers

This section describes how Adaptive Server Anywhere handles errors that occur during a procedure execution, if you have no error handling built in to the procedure.

☞ If you want to have different behavior from that described in this section, you can use exception handlers, described in "Using exception handlers in procedures and triggers" on page 261. Warnings are handled in a slightly different manner from errors: for a description, see "Default handling of warnings in procedures and triggers" on page 260.

There are two ways of handling errors without using explicit error handling:

- ◆ **Default error handling** The procedure or trigger fails and returns an error code to the calling environment.

- ◆ **ON EXCEPTION RESUME** If the ON EXCEPTION RESUME clause is included in the CREATE PROCEDURE statement, the procedure carries on executing after an error, resuming at the statement following the one causing the error.

Default error handling

Generally, if a SQL statement in a procedure or trigger fails, the procedure or trigger terminates execution and control is returned to the application program with an appropriate setting for the SQLSTATE and SQLCODE values. This is true even if the error occurred in a procedure or trigger invoked directly or indirectly from the first one. In the case of a trigger, the operation causing the trigger is also undone and the error is returned to the application.

The following demonstration procedures show what happens when an application calls the procedure **OuterProc**, and **OuterProc** in turn calls the procedure **InnerProc**, which then encounters an error.

```
CREATE PROCEDURE OuterProc()
BEGIN
    MESSAGE 'Hello from OuterProc.';
    CALL InnerProc();
    MESSAGE 'SQLSTATE set to ',
           SQLSTATE, ' in OuterProc.'
END
CREATE PROCEDURE InnerProc()
BEGIN
    DECLARE column_not_found
        EXCEPTION FOR SQLSTATE '52003';
    MESSAGE 'Hello from InnerProc.';
    SIGNAL column_not_found;
    MESSAGE 'SQLSTATE set to ',
           SQLSTATE, ' in InnerProc.';
END
```

Notes

- ◆ The DECLARE statement in **InnerProc** declares a symbolic name for one of the predefined SQLSTATE values associated with error conditions already known to the server. The DECLARE statement does not take any other action.
- ◆ The MESSAGE statement sends a message to the server window and the *dbconsole* message window.
- ◆ The SIGNAL statement generates an error condition from within the **InnerProc** procedure.

The following statement executes the **OuterProc** procedure:

```
CALL OuterProc();
```

The message window of the server then displays the following:

```
Hello from OuterProc.
```

Hello from InnerProc.

No statements following the SIGNAL statement in **InnerProc** are executed: **InnerProc** immediately passes control back to the calling environment, which in this case is the procedure **OuterProc**. No statements following the CALL statement in **OuterProc** are executed. The error condition is returned to the calling environment to be handled there. For example, Interactive SQL handles the error by displaying a message window describing the error.

The TRACEBACK function provides a list of the statements that were executing when the error occurred. You can use the TRACEBACK function from Interactive SQL by typing the following statement:

```
SELECT TRACEBACK(*)
```

Error handling with ON EXCEPTION RESUME

If the ON EXCEPTION RESUME clause is included in the CREATE PROCEDURE statement, the procedure checks the following statement when an error occurs. If the statement handles the error, then the procedure does not return control to the calling environment when an error occurs. Instead, it continues executing, resuming at the statement after the one causing the error.

The following statements are considered error-handling statements:

- ◆ IF
- ◆ SELECT @variable =
- ◆ CASE
- ◆ LOOP
- ◆ LEAVE
- ◆ CONTINUE
- ◆ CALL
- ◆ EXECUTE
- ◆ SIGNAL
- ◆ RESIGNAL
- ◆ DECLARE

The following example illustrates how this works.

Drop the procedures

Remember to drop both the **InnerProc** and **OuterProc** procedures before continuing with the tutorial. You can do this by entering the following commands in the command window:

```
DROP PROCEDURE OUTERPROC;
DROP PROCEDURE INNERPROC
```

The following demonstration procedures show what happens when an application calls the procedure **OuterProc**; and **OuterProc** in turn calls the procedure **InnerProc**, which then encounters an error. These demonstration procedures are based on those used earlier in this section:

```
CREATE PROCEDURE OuterProc()
ON EXCEPTION RESUME
BEGIN
    DECLARE res CHAR(5);
    MESSAGE 'Hello from OuterProc.';
    CALL InnerProc();
    SELECT @res=SQLSTATE;
    IF res='52003' THEN
        MESSAGE 'SQLSTATE set to ',
            res, ' in OuterProc.';
    END IF
END;

CREATE PROCEDURE InnerProc()
ON EXCEPTION RESUME
BEGIN
    DECLARE column_not_found
        EXCEPTION FOR SQLSTATE '52003';
    MESSAGE 'Hello from InnerProc.';
    SIGNAL column_not_found;
    MESSAGE 'SQLSTATE set to ',
        SQLSTATE, ' in InnerProc.';
END
```

The following statement executes the **OuterProc** procedure:

```
CALL OuterProc();
```

The message window of the server then displays the following:

```
Hello from OuterProc.
Hello from InnerProc.
SQLSTATE set to 52003 in OuterProc.
```

The execution path is as follows:

- 1 OuterProc executes and calls InnerProc
- 2 In InnerProc, the SIGNAL statement signals an error.
- 3 The MESSAGE statement is not an error-handling statement, so control is passed back to OuterProc and the message is not displayed.

- 4 In OuterProc, the statement following the error assigns the SQLSTATE value to the variable named **res**. This is an error-handling statement, and so execution continues and the OuterProc message is displayed.

Default handling of warnings in procedures and triggers

Warnings are handled differently from errors. While the default action for errors is to set a value for the SQLSTATE and SQLCODE variables, and return control to the calling environment, the default action for warnings is to set the SQLSTATE and SQLCODE values and continue execution of the procedure.

Drop the procedures

Remember to drop both the **InnerProc** and **OuterProc** procedures before continuing with the tutorial. You can do this by entering the following commands in the command window:

```
DROP PROCEDURE OUTERPROC;
DROP PROCEDURE INNERPROC
```

The following demonstration procedures illustrate default handling of warnings. These demonstration procedures are based on those used in "Default error handling in procedures and triggers" on page 256. In this case, the SIGNAL statement generates a row not found condition, which is a warning rather than an error.

```
CREATE PROCEDURE OuterProc()
BEGIN
    MESSAGE 'Hello from OuterProc.';
    CALL InnerProc();
    MESSAGE 'SQLSTATE set to ',
           SQLSTATE, ' in OuterProc.';
END
CREATE PROCEDURE InnerProc()
BEGIN
    DECLARE row_not_found
        EXCEPTION FOR SQLSTATE '02000';
    MESSAGE 'Hello from InnerProc.';
    SIGNAL row_not_found;
    MESSAGE 'SQLSTATE set to ',
           SQLSTATE, ' in InnerProc.';
END
```

The following statement executes the **OuterProc** procedure:

```
CALL OuterProc();
```

The message window of the server then displays the following:

```
Hello from OuterProc.
Hello from InnerProc.
```


SQLSTATE set to 02000 in InnerProc.

SQLSTATE set to 02000 in OuterProc.

The procedures both continued executing after the warning was generated, with SQLSTATE set by the warning (02000).

Using exception handlers in procedures and triggers

It is often desirable to intercept certain types of errors and handle them within a procedure or trigger, rather than pass the error back to the calling environment. This is done through the use of an **exception handler**.

An exception handler is defined with the EXCEPTON part of a compound statement (see "Using compound statements" on page 239). The exception handler is executed whenever an error occurs in the compound statement. Unlike errors, warnings do not cause exception handling code to be executed. Exception handling code is also executed if an error is encountered in a nested compound statement or in a procedure or trigger that has been invoked anywhere within the compound statement.

Drop the
procedures

Remember to drop both the **InnerProc** and **OuterProc** procedures before continuing with the tutorial. You can do this by entering the following commands in the command window:

```
DROP PROCEDURE OUTERPROC;
DROP PROCEDURE INNERPROC
```

The demonstration procedures used to illustrate exception handling are based on those used in "Default error handling in procedures and triggers" on page 256. In this case, additional code is added to handle the column not found error in the **InnerProc** procedure.

```
CREATE PROCEDURE OuterProc()
BEGIN
    MESSAGE 'Hello from OuterProc.';
    CALL InnerProc();
    MESSAGE 'SQLSTATE set to ',
           SQLSTATE, ' in OuterProc.'
END
CREATE PROCEDURE InnerProc()
BEGIN
    DECLARE column_not_found
        EXCEPTION FOR SQLSTATE '52003';
    MESSAGE 'Hello from InnerProc.';
    SIGNAL column_not_found;
    MESSAGE 'Line following SIGNAL.';
    EXCEPTION
        WHEN column_not_found THEN
            MESSAGE 'Column not found handling.';
```

```
        WHEN OTHERS THEN
            RESIGNAL ;
    END
```

The EXCEPTION statement declares the exception handler itself. The lines following the EXCEPTION statement are not executed unless an error occurs. Each WHEN clause specifies an exception name (declared with a DECLARE statement) and the statement or statements to be executed in the event of that exception. The WHEN OTHERS THEN clause specifies the statement(s) to be executed when the exception that occurred is not in the preceding WHEN clauses.

In this example, the statement RESIGNAL passes the exception on to a higher-level exception handler. RESIGNAL is the default action if WHEN OTHERS THEN is not specified in an exception handler.

The following statement executes the **OuterProc** procedure:

```
CALL OuterProc();
```

The message window of the server then displays the following:

```
Hello from OuterProc.
Hello from InnerProc.
Column not found handling.
SQLSTATE set to 00000 in OuterProc.
```

Notes

- ◆ The lines following the SIGNAL statement in **InnerProc** are not executed; instead, the EXCEPTION statements are executed.
- ◆ As the error encountered was a column not found error, the MESSAGE statement included to handle the error is executed, and SQLSTATE is reset to zero (indicating no errors).
- ◆ After the exception handling code is executed, control is passed back to **OuterProc**, which proceeds as if no error was encountered.
- ◆ You should not use ON EXCEPTION RESUME together with explicit exception handling. The exception handling code is not executed if ON EXCEPTION RESUME is included.
- ◆ If the error handling code for the column not found exception is simply a RESIGNAL statement, control is passed back to the **OuterProc** procedure with SQLSTATE still set at the value 52003. This is just as if there were no error handling code in **InnerProc**. As there is no error handling code in **OuterProc**, the procedure fails.

Exception handling and atomic compound statements	When an exception is handled inside a compound statement, the compound statement completes without an active exception and the changes before the exception are not undone. This is true even for atomic compound statements. If an error occurs within an atomic compound statement and is explicitly handled, some but not all of the statements in the atomic compound statement are executed.
---	---

Nested compound statements and exception handlers

The code following a statement that causes an error is not executed unless an ON EXCEPTION RESUME clause is included in a procedure definition.

You can use nested compound statements to give you more control over which statements are and are not executed following an error.

Drop the procedures

Remember to drop both the **InnerProc** and **OuterProc** procedures before continuing with the tutorial. You can do this by entering the following commands in the command window:

```
DROP PROCEDURE OUTERPROC;
DROP PROCEDURE INNERPROC
```

The following demonstration procedure illustrates how nested compound statements can be used to control flow. The procedure is based on that used as an example in "Default error handling in procedures and triggers" on page 256.

```
CREATE PROCEDURE InnerProc()
BEGIN
    DECLARE column_not_found
    EXCEPTION FOR SQLSTATE VALUE '52003';
    MESSAGE 'Hello from InnerProc';
    SIGNAL column_not_found;
    MESSAGE 'Line following SIGNAL'
EXCEPTION
    WHEN column_not_found THEN
        MESSAGE 'Column not found handling';
    WHEN OTHERS THEN
        RESIGNAL;
    MESSAGE 'Outer compound statement';
END
```

The following statement executes the **InnerProc** procedure:

```
CALL InnerProc();
```

The message window of the server then displays the following:

```
Hello from InnerProc
Column not found handling
```

Outer compound statement

When the SIGNAL statement that causes the error is encountered, control passes to the exception handler for the compound statement, and the Column not found handling message is printed. Control then passes back to the outer compound statement and the Outer compound statement message is printed.

If an error other than column not found is encountered in the inner compound statement, the exception handler executes the RESIGNAL statement. The RESIGNAL statement passes control directly back to the calling environment, and the remainder of the outer compound statement is not executed.

Using the EXECUTE IMMEDIATE statement in procedures

The EXECUTE IMMEDIATE statement allows statements to be built up inside procedures using a combination of literal strings (in quotes) and variables.

For example, the following procedure includes an EXECUTE IMMEDIATE statement that creates a table.

```
CREATE PROCEDURE CreateTableProc(  
    IN tablename char(30) )  
BEGIN  
    EXECUTE IMMEDIATE 'CREATE TABLE ' || tablename || ' (  
    column1 INT PRIMARY KEY) '  
END
```

In ATOMIC compound statements, you cannot use an EXECUTE IMMEDIATE statement that causes a COMMIT, as COMMITs are not allowed in that context.

Transactions and savepoints in procedures and triggers

SQL statements in a procedure or trigger are part of the current transaction (see "Using Transactions and Locks" on page 367). You can call several procedures within one transaction or have several transactions in one procedure.

COMMIT and ROLLBACK are not allowed within any atomic statement (see "Atomic compound statements" on page 241). Note that triggers are fired due to an INSERT, UPDATE, or DELETE which are atomic statements. COMMIT and ROLLBACK are not allowed in a trigger or in any procedures called by a trigger.

Savepoints (see "Savepoints within transactions" on page 418) can be used within a procedure or trigger, but a ROLLBACK TO SAVEPOINT statement can never refer to a savepoint before the atomic operation started. Also, all savepoints within an atomic operation are released when the atomic operation completes.

Some tips for writing procedures

This section provides some pointers for developing procedures.

Check if you need to change the command delimiter

You do not need to change the command delimiter in Interactive SQL or Sybase Central when you are writing procedures. However, if you are creating and testing procedures and triggers from some other browsing tool, you may need to change the command delimiter from the semicolon to another character.

Each statement within the procedure ends with a semicolon. For some browsing applications to parse the CREATE PROCEDURE statement itself, you need the command delimiter to be something other than a semicolon.

If you are using an application that requires changing the command delimiter, a good choice is to use two semicolons as the command delimiter (;;) or a question mark (?) if the system does not permit a multicharacter delimiter.

Remember to delimit statements within your procedure

You should terminate each statement within the procedure with a semicolon. Although you can leave off semicolons for the last statement in a statement list, it is good practice to use semicolons after each statement.

The CREATE PROCEDURE statement itself contains both the RESULT specification and the compound statement that forms its body. No semicolon is needed after the BEGIN or END keywords, or after the RESULT clause.

Use fully-qualified names for tables in procedures

If a procedure has references to tables in it, you should always preface the table name with the name of the owner (creator) of the table.

When a procedure refers to a table, it uses the group memberships of the procedure creator to locate tables with no explicit owner name specified. For example, if a procedure created by **user_1** references **Table_B** and does not specify the owner of **Table_B**, then either **Table_B** must have been created by **user_1** or **user_1** must be a member of a group (directly or indirectly) that is the owner of **Table_B**. If neither condition is met, a table not found message results when the procedure is called.

You can minimize the inconvenience of long fully qualified names by using a correlation name to provide a convenient name to use for the table within a statement. Correlation names are described in "FROM clause" on page 476 of the book *Adaptive Server Anywhere Reference Manual*.

Specifying dates and times in procedures

When dates and times are sent to the database from procedures, they are sent as strings. The date part of the string is interpreted according to the current setting of the DATE_ORDER database option. As different connections may set this option to different values, some strings may be converted incorrectly to dates, or the database may not be able to convert the string to a date.

You should use the unambiguous date format *yyyy-mm-dd* or *yyyy/mm/dd* when sending dates to the database from procedures. These strings are interpreted unambiguously as dates by the database, regardless of the DATE_ORDER database option setting.

☞ For more information on dates and times, see "Date and time data types" on page 233 of the book *Adaptive Server Anywhere Reference Manual*.

Verifying that procedure input arguments are passed correctly

You can verify that input arguments to a procedure are passed correctly in several ways.

You can display the value of the parameter on the message window of the server using the MESSAGE statement. For example, the following procedure simply displays the value of the input parameter *var*:

```
CREATE PROCEDURE message_test (IN var char(40))
BEGIN
    MESSAGE var;
END
```

You can do the following from Interactive SQL:

- 1 Create the procedure.
- 2 Call the procedure:

```
CALL MESSAGE_TEST ('Test Message');
```
- 3 After calling the procedure, double click the server icon in the system tray to ensure that the message was passed properly to the server.

```
SELECT GLOBALVAR
```


Statements allowed in batches

The following statements are not allowed in batches:

- ◆ CONNECT or DISCONNECT statement
- ◆ ALTER PROCEDURE or ALTER FUNCTION statement
- ◆ CREATE TRIGGER statement
- ◆ Interactive SQL commands such as INPUT or OUTPUT

Otherwise, any SQL statement is allowed, including data definition statements such as CREATE TABLE, ALTER TABLE, and so on.

The CREATE PROCEDURE statement is allowed, but must be the final statement of the batch. Therefore a batch can contain only a single CREATE PROCEDURE statement.

Using SELECT statements in batches

You can include one or more SELECT statements in a batch. Multiple SELECT statements are allowed only if they return the same result columns.

The following is a valid batch:

```
IF EXISTS( SELECT *
           FROM systable
           WHERE table_name='employee' )
THEN
    SELECT emp_lname AS LastName,
           emp_fname AS FirstName
    FROM employee;
    SELECT lname, fname
    FROM customer;
    SELECT last_name, first_name
    FROM contact;
END IF
```

The alias for the result set is required only in the first SELECT statement, as the server uses the first SELECT statement in the batch to describe the result set.

A RESUME statement is required following each query to retrieve the next result set.

The following is not a valid batch, as the two queries return different result sets:

```
IF EXISTS( SELECT * FROM systable
           WHERE table_name='employee' )
```

```
THEN
  SELECT emp_lname AS LastName,
         emp_fname AS FirstName
  FROM employee;
  SELECT id, lname, fname
  FROM customer;
END IF
```

Calling external libraries from procedures

You can call a function in an external Dynamic Link Library (DLL) from a stored procedure or user-defined functions under an operating system that supports DLLs. You can also call functions in an NLM under NetWare. You cannot use external functions on UNIX.

Adaptive Server Anywhere includes a set of system procedures that make use of this capability to send MAPI e-mail messages and carry out other functions. This section describes how to use the external library calls in procedures.

Caution: external libraries can corrupt your database

External libraries called from procedures share the memory of the server. If you call a DLL from a procedure and the DLL contains memory-handling errors, you can crash the server or corrupt your database. Ensure that your libraries are thoroughly tested before deploying them on production databases.

☞ For information on MAPI and other system procedures, see "System Procedures and Functions" on page 751 of the book *Adaptive Server Anywhere Reference Manual*.

Creating procedures and functions with external calls

This section presents some examples of procedures and functions with external calls.

☞ For a full description of the CREATE PROCEDURE statement syntax, see "CREATE PROCEDURE statement" on page 403 of the book *Adaptive Server Anywhere Reference Manual*.

☞ For a full description of the CREATE FUNCTION statement syntax for external calls, see "CREATE FUNCTION statement" on page 397 of the book *Adaptive Server Anywhere Reference Manual*.

DBA permissions required

You must have DBA permissions in order to create external procedures or functions. This requirement is more strict than the RESOURCE permissions required for creating other procedures or functions.

Syntax

A procedure that calls a function *function_name* in DLL *library.dll* can be created as follows:

```
CREATE PROCEDURE dll_proc ( parameter-list )
EXTERNAL NAME 'function_name@library.dll'
```

Such a procedure is called an **external stored procedure**. If you call an external DLL from a procedure, the procedure cannot carry out any other tasks; it just forms a wrapper around the DLL.

An analogous CREATE FUNCTION statement is as follows:

```
CREATE FUNCTION dll_func ( parameter-list )
RETURNS data-type
EXTERNAL NAME 'function_name@library.dll'
```

In these statements, *function_name* is the name of a function in the dynamic link library, and *library.dll* is the name of the library. The arguments in the procedure argument list must correspond in type and order to the arguments for the library function; they are passed to the external DLL function in the order in which they are listed. Any value returned by the external function is in turn returned by the procedure to the calling environment.

No other statements permitted

A procedure that calls an external function can include no other statements: its sole purposes are to take arguments for a function, call the function, and return any value and returned arguments from the function to the calling environment. You can use IN, INOUT, or OUT parameters in the procedure call in the same way as for other procedures: the input values get passed to the external function, and any parameters modified by the function are returned to the calling environment in OUT or INOUT parameters.

System-dependent calls

You can specify operating-system dependent calls, so that a procedure calls one function when run on one operating system, and another function (presumably analogous) on another operating system. The syntax for such calls is to prefix the function name with the operating system name. For example:

```
CREATE PROCEDURE dll_proc ( parameter-list )
EXTERNAL NAME
'OS2:os2_fn@os2_lib.dll;WindowsNT:nt_fn@nt_lib.dll'
```

The operating system identifier must be one of **OS2**, **WindowsNT**, **Windows95**, **Windows3X**, or **NetWare**.

If no system identifier for the current operating system is provided, and a function with no system identifier is provided, that function is called.

NetWare calls have a slightly different format than the other operating systems. All symbols are globally known under NetWare, so any symbol (such as a function name) that is exported must be unique to all NLMs on the system. Consequently, the NLM name is not necessary in the call, and the call has the following syntax:

```
CREATE PROCEDURE dll_proc ( parameter-list )
EXTERNAL NAME 'NetWare:nw_fn'
```

No library name needs to be provided.

External function prototypes

When an external function is called, a stack is fabricated with the arguments (or argument references in the case of INOUT or OUT parameters) and the DLL is called. Only the following data types can be passed to an external library:

- ◆ CHARACTER data types, but INOUT and OUT parameters must be no more than 255 bytes in length
- ◆ SMALLINT and INT data types
- ◆ REAL and DOUBLE data types

This section describes the format of the function prototype.

☞ For information about passing parameters to external functions, see "Passing parameters to external procedures and functions" on page 274.

For convenience, a header file named *dllapi.h* is provided in the *h* subdirectory. This header file handles the platform-dependent features of external function prototypes. If you use this header file, then all external function prototypes are of the following form.

```
return_type _entry function_name( argument-list );
```

If you do not use this header file, external function declarations should follow the following guidelines:

- ◆ **Windows NT and Windows 95** The function declaration should be of the following form for the Watcom C/C++ compiler:

```
return-type __stdcall function-name( argument-list )
```

- ◆ **Windows 3.x** All pointers are far pointers, so the DLL must be at least compiled under the large model. The function declaration should be of the following form for the Watcom C/C++ compiler:

```
return-type __far __pascal function-name( argument-list )
```

No more than 256 parameters can be used, of any type.

- ◆ **NetWare** The function declaration should be of the following form for the Watcom C/C++ compiler:

```
return-type function-name( argument-list );
```

Passing parameters to external procedures and functions

SQL data types are mapped to their C equivalents as follows:

SQL data type	C data type
INTEGER	long
SMALLINT	short
FLOAT	float
DOUBLE	double
CHAR(n), n < 254	char *

These are the only SQL data types you can use. Any other data type produces an error.

Procedure parameters that are INOUT or OUT parameters are passed to the external function by reference. For example, the procedure

```
CREATE PROCEDURE dll_proc( INOUT xvar REAL )
EXTERNAL NAME 'function_name@library.dll'
```

has an associated C function parameter declaration of

```
function_name( float * xvar )
```

Procedure parameters that are IN parameters are passed to the external function by value. For example, the procedure

```
CREATE PROCEDURE dll_proc( IN xvar REAL )
EXTERNAL NAME 'function_name@library.dll'
```

has an associated external function parameter declaration of

```
function_name( float xvar )
```

Character data types are an exception to IN parameters being passed. They are always passed by reference, whether they are IN, OUT, or INOUT parameters. For example, the procedure

```
CREATE PROCEDURE dll_proc ( IN invar CHAR( 128 ) )
EXTERNAL NAME 'function_name@library.dll'
```

has the following external function parameter declaration

```
function_name( char * invar )
```

External function return types

The following table lists the supported return types, and how they map to the return type of the SQL external function or external procedure.

C data type	SQL data type
void	Used for external procedures.
char *	External function returning CHAR(), up to 254 characters.
long	External function returning INTEGER
float	External function returning FLOAT
double	External function returning DOUBLE.

If a function in the external library returns NULL, and the SQL external function was declared to return CHAR(), then the return value of the SQL extended function is NULL. NULL is also returned if the pointer is determined to be invalid.

Special considerations when passing character types

For the character data type (CHAR), Adaptive Server Anywhere allocates a 255-byte buffer (including one for the null terminator) for each parameter. If the parameter is an INOUT parameter, the existing value is copied into the buffer and null terminated, and a pointer to this buffer is passed to the external function. The external function should therefore not allocate a buffer of its own for OUT or INOUT character parameters: the server has already allocated the space. If the external function writes beyond the 255 bytes (including the ending null character), it is writing over data structures in the server.

When the entry point returns, the parameter buffers are translated back into their server data structure string equivalents based on the **strlen()** value of the buffer.

The external function should be sure to null-terminate any output string parameters. OUT parameters follow the same procedure except that as there is no initial data, no initial value of the output buffer parameter is guaranteed.

