MICROSOFT
TRAINING
AND CERTIFICATION

Microsoft® Official Curriculum

Module 9: Implementing Stored Procedures

Contents

Microsoft®

Project Lead: Rich Rose
Instructional Designers: Rich Rose, Cheryl Hoople, Marilyn McGill
Instructional Software Design Engineers: Karl Dehmer, Carl Raebler, Rick Byham
Technical Lead: Karl Dehmer
Subject Matter Experts: Karl Dehmer, Carl Raebler, Rick Byham
Graphic Artist: Kirsten Larson (Independent Contractor)
Editing Manager: Lynette Skinner
Editor: Wendy Cleary
Copy Editor: Edward McKillop (S&T Consulting)
Production Manager: Miracle Davis
Production Coordinator: Jenny Boe
Production Support: Lori Walker (S&T Consulting)
Test Manager: Sid Benavente
Courseware Testing: TestingTesting123
Classroom Automation: Lorrin Smith-Bates
Creative Director, Media/Sim Services: David Mahlmann
Web Development Lead: Lisa Pease
CD Build Specialist: Julie Challenger
Online Support: David Myka (S&T Consulting)
Localization Manager: Rick Terek
Operations Coordinator: John Williams
Manufacturing Support: Laura King; Kathy Hershey
Lead Product Manager, Release Management: Bo Galford
Lead Product Manager, Data Base: Margo Crandall
Group Manager, Courseware Infrastructure: David Bramble
Group Product Manager, Content Development: Dean Murray
General Manager: Robert Stewart

Instructor Notes

Presentation:
90 Minutes

Lab:
60 Minutes

This module provides students with a description of how to use stored procedures to improve application design and performance by encapsulating business rules. It discusses ways to process common queries and data modifications.

The module begins by discussing what stored procedures are, the advantages of using them, and how they are processed.

The next section discusses how to create, execute, and modify stored procedures. The section following discusses using stored procedures with input and output parameters. It also discusses recompiling options.

Final sections describe executing extended stored procedures, how to handle error messages, and performance issues to consider when implementing stored procedures. Examples and demonstrations of stored procedures are provided throughout the module.

In the first lab, students create stored procedures based on provided models and use Microsoft® SQL Server™ Enterprise Manager and SQL Query Analyzer to display information on stored procedures. In the second lab, students use a wizard to create a stored procedure and generate a stored procedure script. Students also create stored procedures that accept information with input parameters and return output parameters.

After completing module, students will be able to:

- Describe how a stored procedure is processed.
- Create, execute, modify, and drop a stored procedure.
- Create stored procedures that accept parameters.
- Execute extended stored procedures.
- Create custom error messages.

Materials and Preparation

This section provides the materials and preparation tasks that you need to teach this module.

Required Materials

To teach this module, you need the following materials:

- Microsoft PowerPoint® file 2073a_09.ppt
- The C:\Moc\2073A\Demo\D09_Ex.sql example file, which contains all of the example scripts from the module, unless otherwise noted in the module

Preparation Tasks

To prepare for this module, you should:

- Read all of the materials for this module.
- Complete the labs.
- Complete all demonstrations.
- Practice the presentation.
- Review any relevant white papers located on the Trainer Materials compact disc.

Demonstration

This section provides demonstration procedures that will not fit in the margin notes or are not appropriate for the student notes.

Update Customer Phone

► **To prepare for the demonstration**

1. Open C:\Moc\2073A\Demo\ D09_UpdateCustomerPhone.sql, review its contents, and then execute it.
2. Open C:\Moc\2073A\Demo\D09_TestUpdate.sql, and review its contents. Select and execute each statement that follows the comment that explains what is being tested so that you are familiar with the results.

   In-line comments appear in the last SELECT statement in the script that finds a member who meets all of the error checking criteria. If time permits, you may want to execute this statement three times: the first time as-is; next, removing the first in-line comment; and, finally, removing the second in-line comment. This allows you to remove three different members from the database and compare the error messages.

► **To perform the demonstration**

1. In SQL Query Analyzer, open C:\Moc\2073A\Demo\D09_UpdateCustomerPhone.sql.
2. So that students in the back of the room can see the script easily, on the **View** menu, click **Properties**.
3. In the **Size** box, type **20** and then click **OK**.

Note The main points in the script are included in the Student Workbook.

4. Scroll past the "IF EXISTS…DROP" statements.
5. Point out the custom error messages that are created with the **sp_addmessage** system stored procedure.
6. When you review the section of the script that determines whether the customer number exists in the database, ask students why these statements are included.

   Answer: If you were to execute an UPDATE statement that specifies a customer number that does not exist, the result is "command completed successfully."

7. Point out that the actual transaction that updates the customer phone number is short and begins only after all error checking and business logic is complete. This example illustrates an optimistic model that prefers a smaller transaction to an all-encompassing one. The model that you select depends on your application environment and whether this has a negative impact on data integrity.
8. After you review the script, execute it to add the **UpdateCustomerPhone** stored procedure to the **Northwind** database.
9. Open C:\Moc\2073A\Demo\D09_TestUpdate.sql and then select and execute each statement that follows the comment that explains the error condition to be tested.

Module Strategy

Use the following strategy to present this module:

- Introduction to Stored Procedures

  Introduce the elements of a stored procedure, including an overview of the types of stored procedures—system, local, temporary, remote, or extended. The focus of this module is on creating stored procedures that are defined in a user's local database. Explain how stored procedures are processed and then point out why students would want to create stored procedures in their applications.

- Creating, Executing, and Modifying Stored Procedures

  Describe how to create, execute, and modify stored procedures. Include a discussion on the WITH ENCRYPTION option and programming guidelines when creating stored procedures.

- Using Parameters in Stored Procedures

  Give this section the greatest emphasis in the module, because parameters provide stored procedures with the greatest functionality and flexibility. Discuss input and output parameters. Compare stored procedures that specify parameters by reference and position. You should emphasize that passing by reference is preferred, as it provides the best documentation. Finally, briefly discuss the ability to recompile a stored procedure explicitly. Point out that students should use this feature infrequently.

- Executing Extended Stored Procedures

  Briefly review extended stored procedures. Emphasize that if students create a stored procedure with the sp_ prefix, which calls an extended stored procedure, they can then execute the extended stored procedure from within any database. If you have time, you may want to demonstrate how easily students can create a stored procedure to call the **xp_cmdshell** extended stored procedure.

- Handling Error Messages

  Emphasize the importance of handling errors when you discuss the RETURN statement, the **@@error** system function, the **sp_addmessage** system stored procedure, and the RAISERROR statement. Demonstrate how to remove a member from the **Northwind** database to illustrate strategies for handling error messages. You must review the setup information before performing the demonstration.

- Performance Considerations

  Discuss some of the performance considerations that are involved in using stored procedures.

Customization Information

This section identifies the lab setup requirements for a module and the configuration changes that occur on student computers during the labs. This information is provided to assist you in replicating or customizing Microsoft Official Curriculum (MOC) courseware.

Important The labs in this module are dependent on the classroom configuration that is specified in the Customization Information section at the end of the *Classroom Setup Guide* for course 2073A, *Programming a Microsoft SQL Server 2000 Database*.

Lab Setup

The following section describes the setup requirement for the labs in this module.

Setup Requirement 1

The lab in this module requires the **ClassNorthwind** database to be in a state required for this lab. To prepare student computers to meet this requirement, perform one of the following actions:

- Complete the prior lab
- Execute the C:\Moc\2073A\Batches\Restore09A.cmd batch file.

Setup Requirement 2

The lab in this module requires the **ClassNorthwind** database to be in a state required for this lab. To prepare student computers to meet this requirement, perform one of the following actions:

- Complete the prior lab
- Execute the C:\Moc\2073A\Batches\Restore09B.cmd batch file.

Warning If this course has been customized, students must execute the C:\Moc\2073A\Batches\Restore09A.cmd batch file to ensure that the first lab will function properly.

If this course has been customized, students must execute the C:\Moc\2073A\Batches\Restore09B.cmd batch file to ensure that the second lab will function properly.

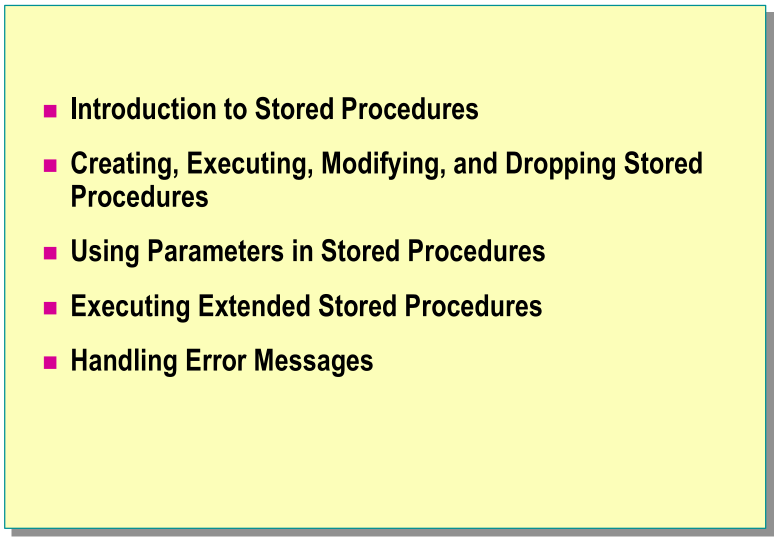
Lab Results

There are no configuration changes on student computers that affect replication or customization.

Overview

Topic Objective
To provide an overview of the module topics and objectives.

Lead-in
In this module, you will learn about stored procedures and why you will want to use them.



- Introduction to Stored Procedures
- Creating, Executing, Modifying, and Dropping Stored Procedures
- Using Parameters in Stored Procedures
- Executing Extended Stored Procedures
- Handling Error Messages

Objectives

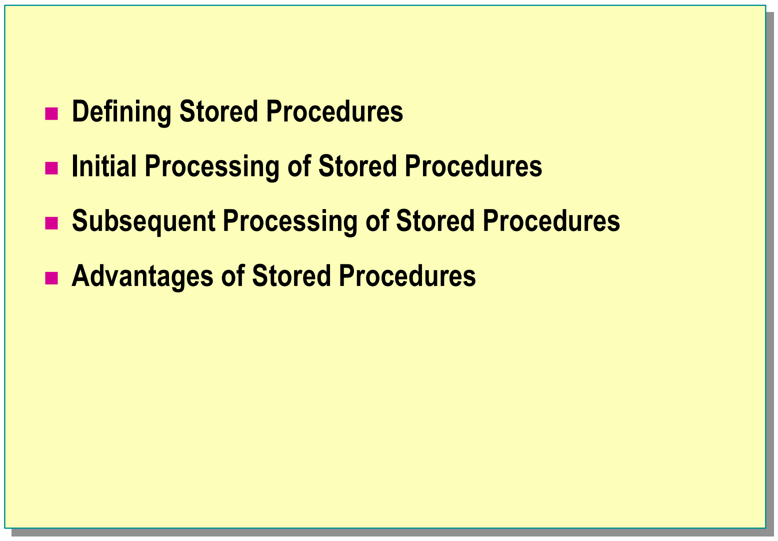
After completing this module, you will be able to:

- Describe how a stored procedure is processed.
- Create, execute, modify, and drop a stored procedure.
- Create stored procedures that accept parameters.
- Execute extended stored procedures.
- Create custom error messages.

◆ Introduction to Stored Procedures

Topic Objective
To list the topics in this section.

Lead-in
In this section, we'll cover…



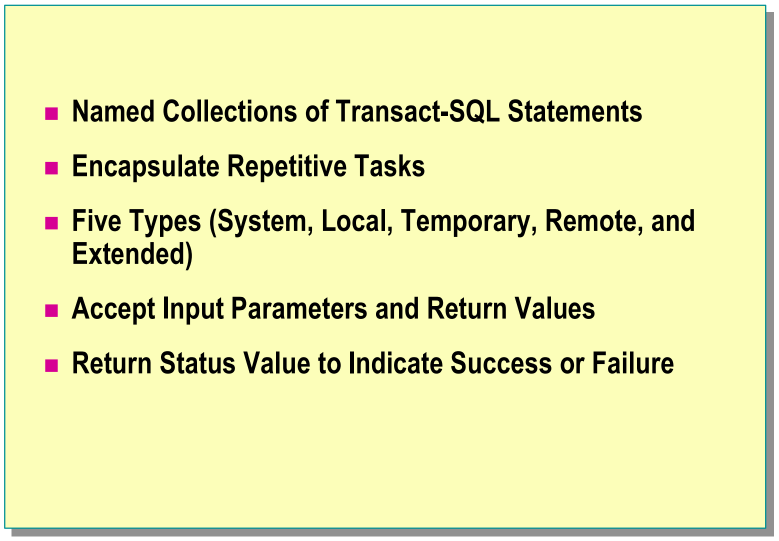
- Defining Stored Procedures
- Initial Processing of Stored Procedures
- Subsequent Processing of Stored Procedures
- Advantages of Stored Procedures

This section introduces the different types of Microsoft® SQL Server™ 2000 stored procedures, describes how stored procedures are processed—both initially and on subsequent execution—and lists some of the advantages of using stored procedures.

Defining Stored Procedures

Topic Objective
To define stored procedures.

Lead-in
A stored procedure is a precompiled collection of Transact-SQL statements that is stored on the server.



- Named Collections of Transact-SQL Statements
- Encapsulate Repetitive Tasks
- Five Types (System, Local, Temporary, Remote, and Extended)
- Accept Input Parameters and Return Values
- Return Status Value to Indicate Success or Failure

A stored procedure is a named collection of Transact-SQL statements that is stored on the server. Stored procedures are a method of encapsulating repetitive tasks. Stored procedures support user-declared variables, conditional execution, and other powerful programming features.

SQL Server supports five types of stored procedures:

System Stored Procedures (sp_) Stored in the **master** database, system stored procedures (identified by the sp_ prefix) provide an effective method to retrieve information from system tables. They allow system administrators to perform database administration tasks that update system tables even though the administrators do not have permission to update the underlying tables directly. System stored procedures can be executed in any database.

Local Stored Procedures Local stored procedures are created in individual user databases.

Temporary Stored Procedures Temporary stored procedures can be local, with names that start with a single number sign (#), or global, with names that start with a double number sign (##). Local temporary stored procedures are available within a single user session; global temporary stored procedures are available for all user sessions.

Remote Stored Procedures Remote stored procedures are an earlier feature of SQL Server. Distributed queries now support this functionality.

For Your Information
Some system stored procedures call extended stored procedures.

Extended Stored Procedures (xp_) Extended stored procedures are implemented as dynamic-link libraries (DLLs) executed outside of the SQL Server environment. Extended stored procedures are typically identified by the xp_ prefix. They are executed in a manner similar to that of stored procedures.

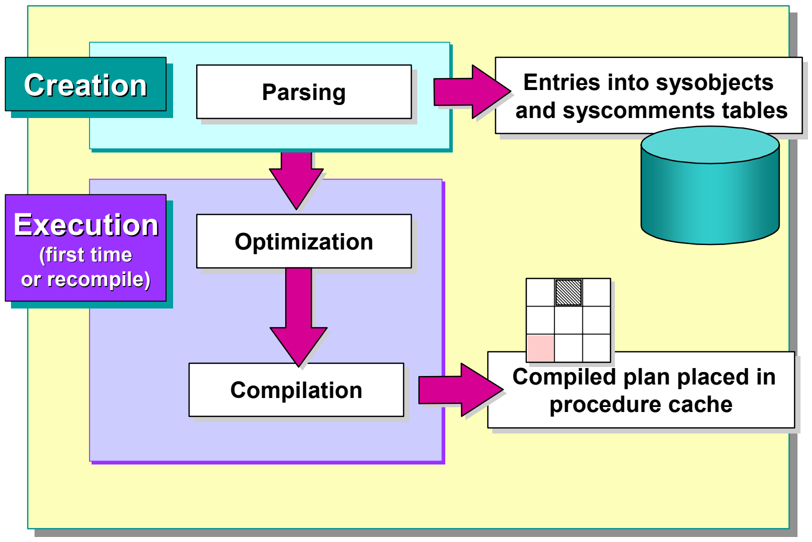
Stored procedures in SQL Server are similar to procedures in other programming languages, in that they can:

- Contain statements that perform operations in the database, including the ability to call other stored procedures.
- Accept input parameters.
- Return a status value to a calling stored procedure or batch to indicate success or failure (and the reason for failure).
- Return multiple values to the calling stored procedure or batch in the form of output parameters.

Initial Processing of Stored Procedures

Topic Objective
To show the steps in processing a stored procedure.

Lead-in
Processing a stored procedure includes creating it and then executing it the first time, which places its execution plan in the cache.


Creation
Parsing
Entries into sysobjects and syscomments tables
Execution (first time or recompile)
Optimization
Compilation
Compiled plan placed in procedure cache

Processing a stored procedure includes creating it and then executing it the first time, which places its execution plan in the procedure cache. The procedure cache is a pool of memory containing the execution plans for all currently executing Transact-SQL statements. The size of the procedure cache fluctuates dynamically according to activity levels. The procedure cache is located in the memory pool, which is the main unit of memory for SQL Server. It contains most of the data structures that use memory in SQL Server.

Creation

When a stored procedure is created, the statements in it are parsed for syntactical accuracy. SQL Server then stores the name of the stored procedure in the **sysobjects** system table and the text of the stored procedure in the **syscomments** system table in the current database. An error is returned if a syntax error is encountered, and the stored procedure is not created.

Delivery Tip
Point out the delayed name resolution process.

Delayed Name Resolution

A process called delayed name resolution allows stored procedures to refer to objects that do not exist when the stored procedure is created. This process permits flexibility, because stored procedures and the objects that they reference do not have to be created in a particular order. The objects must exist by the time the stored procedure is executed. Delayed name resolution is performed at the time the stored procedure is executed.

Execution (First Time or Recompile)

The first time that a stored procedure is executed, or if the stored procedure must be recompiled, the query processor reads the stored procedure in a process called resolution.

Certain changes in a database can cause an execution plan to be either inefficient or no longer valid. SQL Server detects these changes and automatically recompiles the execution plan when any of the following apply:

- Any structural change is made to a table or view referenced by the query (ALTER TABLE and ALTER VIEW).
- New distribution statistics are generated, either explicitly from a statement, such as UPDATE STATISTICS, or automatically.
- An index used by the execution plan is dropped.
- Significant changes are made to keys (the INSERT or DELETE statement) for a table referenced by the query.

Optimization

When a stored procedure successfully passes the resolution stage, the SQL Server query optimizer analyzes the Transact-SQL statements in the stored procedure and creates a plan that contains the fastest method to access the data. To do so, the query optimizer takes into account:

- The amount of data in the tables.
- The presence and nature of table indexes and the distribution of data in the indexed columns.
- The comparison operators and comparison values that are used in WHERE clause conditions.
- The presence of joins and the UNION, GROUP BY, or ORDER BY clause.

Compilation

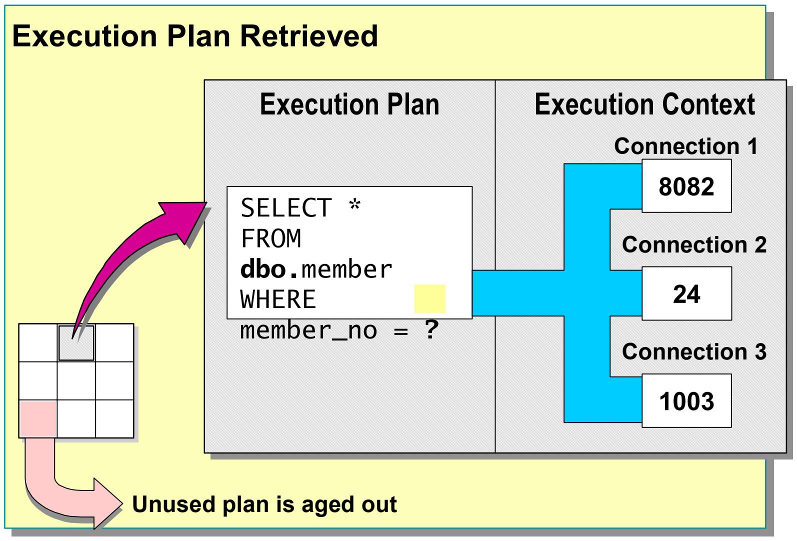
Compilation refers to the process of analyzing the stored procedure and creating an execution plan that is in the procedure cache. The procedure cache contains the most valuable stored procedure execution plans. Factors that increase the value of a plan include the following:

- Time required to recompile (high compile cost)
- Frequent usage

Subsequent Processing of Stored Procedures

Topic Objective
To describe the subsequent processing of stored procedures.

Lead-in
Subsequent processing of stored procedures is faster than initial processing, because SQL Server uses the execution plan in the procedure cache.



Subsequent processing of stored procedures is faster than initial processing, because SQL Server uses the optimized execution plan in the procedure cache.

For server settings, use **sp_configure**; for database settings, use **sp_dboption**; for connection settings, use SET options.

If the following conditions apply, SQL Server uses the in-memory plan to execute the query subsequently:

- The current environment is the same as the environment in which the plan was compiled.

  Server, database, and connection settings determine the environment.

- Objects to which the stored procedure refers do not require name resolution.

  Objects require name resolution when objects that are owned by different users have the same names. For example, if the **sales** role owns a **Product** table, and the **development** role owns a **Product** table, SQL Server must determine the table on which to operate each time that a **Product** table is referenced.

SQL Server execution plans have two main components:

- Execution Plan—most of the execution plan is in this reentrant, read-only data structure that any number of users can use.
- Execution Context—each user currently executing the query has this reusable data structure that holds the data specific to his or her execution, such as parameter values. If a user executes a query, and one of the structures is not in use, it is reinitialized with the context for the new user.

At most, there will always be one compiled plan in the cache for each unique combination of stored procedure plus environment. There can be many plans in cache for the same stored procedure if each is for a different environment.

The following factors result in different environments that affect compilation choices:

- Parallel versus serial compiled plans
- Implicit ownership of objects
- Different SET options

Note For more information on parallel execution plans, see the "Degree of Parallelism" topic in SQL Server Books Online.

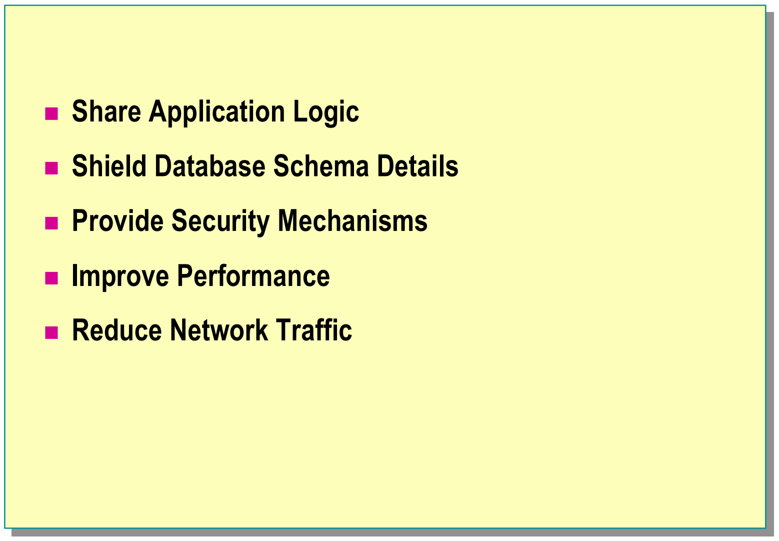
Developers should choose an environment for their applications and use it. Objects whose implicit ownership resolution is ambiguous should use explicit resolution by specifying the object owner. SET options should be consistent; they should be set at the start of a connection and not changed.

After an execution plan is generated, it stays in the procedure cache. SQL Server ages old, unused plans out of the cache only when space is needed.

Advantages of Stored Procedures

Topic Objective
To show the advantages of stored procedures.

Lead-in
Stored procedures significantly reduce resource and time requirements for execution.



- Share Application Logic
- Shield Database Schema Details
- Provide Security Mechanisms
- Improve Performance
- Reduce Network Traffic

Stored procedures offer numerous advantages. They can:

- Share application logic with other applications, thereby ensuring consistent data access and modification.

  Stored procedures can encapsulate business functionality. Business rules or policies encapsulated in stored procedures can be changed in a single location. All clients can use the same stored procedures to ensure consistent data access and modification.
- Shield users from exposure to the details of the tables in the database. If a set of stored procedures supports all of the business functions that users need to perform, users never need to access the tables directly.
- Provide security mechanisms. Users can be granted permission to execute a stored procedure even if they do not have permission to access the tables or views to which the stored procedure refers.
- Improve performance. Stored procedures implement many tasks as a series of Transact-SQL statements. Conditional logic can be applied to the results of the first Transact-SQL statements to determine which subsequent Transact-SQL statements are executed. All of these Transact-SQL statements and conditional logic become part of a single execution plan on the server.

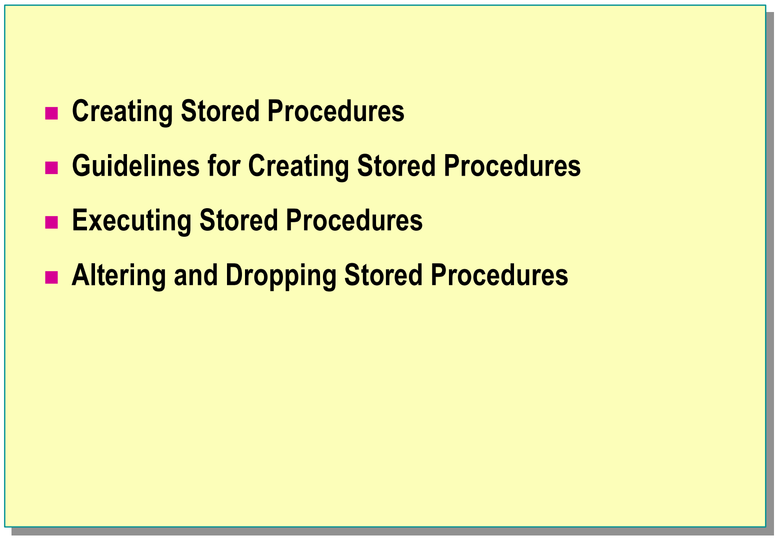
Network traffic is reduced because fewer packets are required to send requests.

- Reduce network traffic. Rather than sending hundreds of Transact-SQL statements over the network, users can perform a complex operation by sending a single statement, which reduces the number of requests that pass between client and server.

◆ Creating, Executing, Modifying, and Dropping Stored Procedures

Topic Objective
To introduce creating, executing, and modifying stored procedures.

Lead-in
Now that we've defined stored procedures and how they're processed, we'll discuss how to create, execute, and modify them.



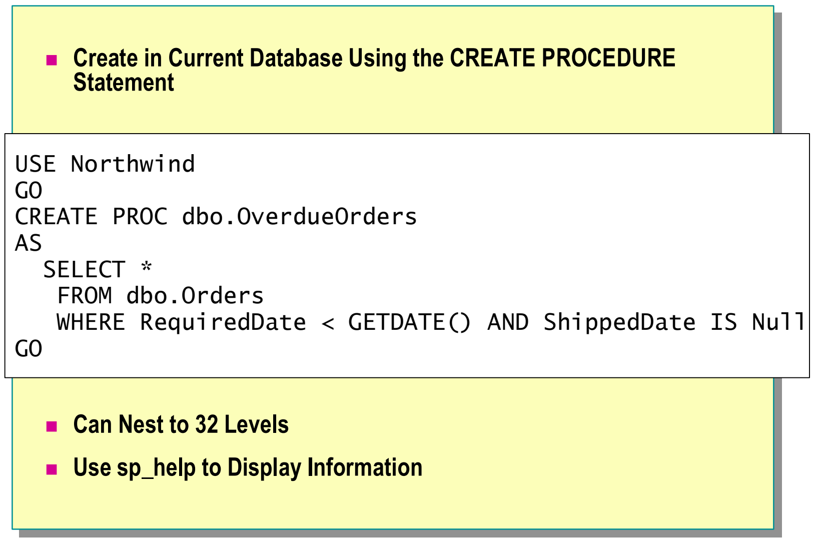
- **Creating Stored Procedures**
- **Guidelines for Creating Stored Procedures**
- **Executing Stored Procedures**
- **Altering and Dropping Stored Procedures**

This section describes how to create, execute, modify, and drop stored procedures.

Creating Stored Procedures

Topic Objective
To introduce the CREATE PROCEDURE syntax.

Lead-in
Use the CREATE PROCEDURE statement to create stored procedures in the current database.



- **Create in Current Database Using the CREATE PROCEDURE Statement**

```
USE Northwind
GO
CREATE PROC dbo.OverdueOrders
AS
  SELECT *
   FROM dbo.Orders
   WHERE RequiredDate < GETDATE() AND ShippedDate IS Null
GO
```

- **Can Nest to 32 Levels**
- **Use sp_help to Display Information**

The syntax does not allow specifying the database name as a prefix to the object name.

You can create a stored procedure in the current database only—except for temporary stored procedures, which are always created in the **tempdb** database. Creating a stored procedure is similar to creating a view. First, write and test the Transact-SQL statements that you want to include in the stored procedure. Then, if you receive the results that you expect, create the stored procedure.

Using CREATE PROCEDURE

You create stored procedures by using the CREATE PROCEDURE statement. Consider the following facts when you create stored procedures:

- Stored procedures can reference tables, views, user-defined functions, and other stored procedures, as well as temporary tables.
- If a stored procedure creates a local temporary table, the temporary table only exists for the purpose of the stored procedure and disappears when stored procedure execution completes.
- A CREATE PROCEDURE statement cannot be combined with other Transact-SQL statements in a single batch.
- The CREATE PROCEDURE definition can include any number and type of Transact-SQL statements, with the exception of the following object creation statements: CREATE DEFAULT, CREATE PROCEDURE, CREATE RULE, CREATE TRIGGER, and CREATE VIEW. Other database objects can be created within a stored procedure and should be qualified with the name of the object owner.
- To execute the CREATE PROCEDURE statement, you must be a member of the system administrators (**sysadmin**) role, database owner (**db_owner**) role, or the Data Definition Language (DDL) administrator (**db_ddladmin**) role, or you must have been granted CREATE PROCEDURE permission.
- The maximum size of a stored procedure is 128 megabytes (MB), depending on available memory.

Partial Syntax

CREATE PROC [EDURE] *procedure_name* [; *number*]
  [{ *@parameter data_type* }
    [VARYING] [= *default*] [OUTPUT]
  ] [,...n]
 [WITH
  { RECOMPILE | ENCRYPTION | RECOMPILE , ENCRYPTION }]
 [FOR REPLICATION]
 AS *sql_statement* [*...n*]

Example

The following statements create a stored procedure that lists all overdue orders in the **Northwind** database.

```
USE Northwind
GO
CREATE PROC dbo.OverdueOrders
AS
  SELECT *
   FROM dbo.Orders
   WHERE RequiredDate < GETDATE() AND ShippedDate IS Null
GO
```

Nesting Stored Procedures

Stored procedures can be nested (one stored procedure calls another). Characteristics of nesting include the following:

- Stored procedures can be nested to 32 levels. Attempting to exceed 32 levels of nesting causes the entire calling stored procedure chain to fail.
- The current nesting level is stored in the **@@nestlevel** system function.
- If one stored procedure calls a second stored procedure, the second stored procedure can access all of the objects that the first stored procedure created, including temporary tables.
- Nested stored procedures can be recursive. For example, Stored Procedure X can call Stored Procedure Y. While Stored Procedure Y is executing, it can call Stored Procedure X.

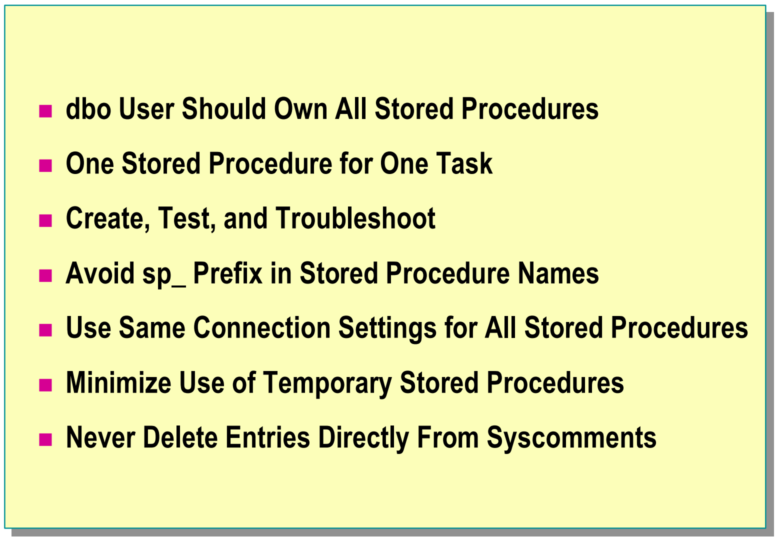
Viewing Information About Stored Procedures

As with other database objects, the following system stored procedures can be used to find additional information about all types of stored procedures: **sp_help**, **sp_helptext**, and **sp_depends**. To print a list of stored procedures and owner names in the database, use the **sp_stored_procedures** system stored procedure. You can also query the **sysobjects**, **syscomments**, and **sysdepends** system tables to obtain information.

Guidelines for Creating Stored Procedures

Topic Objective
To describe the guidelines for creating stored procedures.

Lead-in
Consider these guidelines when you create stored procedures.



- **dbo User Should Own All Stored Procedures**
- **One Stored Procedure for One Task**
- **Create, Test, and Troubleshoot**
- **Avoid sp_ Prefix in Stored Procedure Names**
- **Use Same Connection Settings for All Stored Procedures**
- **Minimize Use of Temporary Stored Procedures**
- **Never Delete Entries Directly From Syscomments**

Consider the following guidelines when you create stored procedures:

- To avoid situations in which the owner of a stored procedure and the owner of the underlying tables differ, it is recommended that the **dbo** user own all objects in a database. Because a user can be a member of multiple roles, always specify the **dbo** user as the owner name when you create the object. Otherwise, the object will be created with your user name as the owner:
  - You must also have appropriate permissions on all of the tables or views that are referenced within the stored procedure.
  - Avoid situations in which the owner of a stored procedure and the owner of the underlying tables differ.

Note If you are creating a user-defined system stored procedure, you must be logged in as a member of the system administrators (**sysadmin**) role and use the **master** database.

- Design each stored procedure to accomplish a single task.
- Create, test, and troubleshoot your stored procedure on the server; then test it from the client.
- To easily distinguish system stored procedures, avoid using the **sp_** prefix when you name local stored procedures.

Delivery Tip
Suggest that students first use **osql** to test stored procedure performance because of its low application overhead.

- All stored procedures should use the same connection settings.

  SQL Server saves the settings of both SET QUOTED_IDENTIFIER and SET ANSI_NULLS when a stored procedure is created or altered. These original settings are used when the stored procedure is executed. Therefore, any client session settings for these SET options are ignored during stored procedure execution.

  Other SET options, such as SET ARITHABORT, SET ANSI_WARNINGS, and SET ANSI_PADDINGS, are not saved when a stored procedure is created or altered.

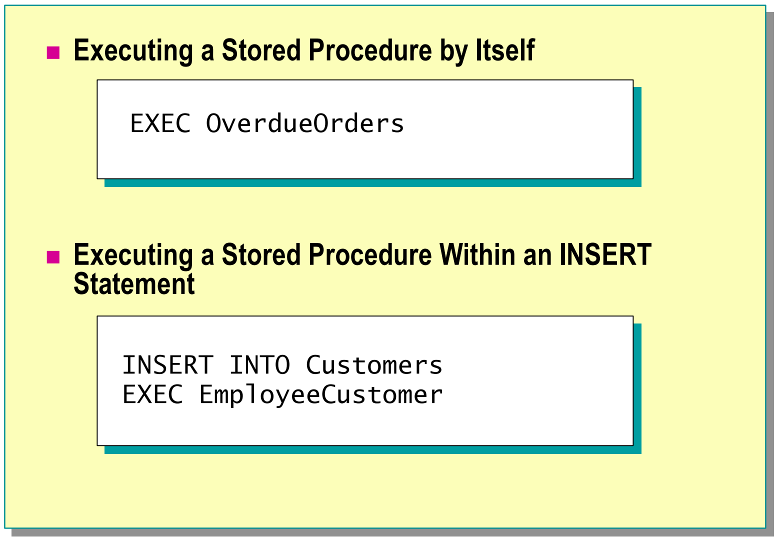
  To determine whether the ANSI SET options were enabled when a stored procedure was created, query the OBJECTPROPERTY system function. SET options should not be changed during the execution of stored procedures.

- Minimize use of temporary stored procedures to avoid contention on the system tables in **tempdb**, a situation that can adversely affect performance.
- Use **sp_executesql** instead of using the EXECUTE statement to dynamically execute a string in a stored procedure. **sp_executesql** is more efficient because it generates execution plans that SQL Server is more likely to reuse. SQL Server compiles the Transact-SQL statement or statements in the string into an execution plan that is separate from the execution plan of the stored procedure. You can use **sp_executesql** when executing a Transact-SQL statement multiple times, if the only variation is in the parameter values supplied to the Transact-SQL statement.
- Never delete entries directly from the **syscomments** system table. If you do not want users to be able to view the text of your stored procedures, you must create them by using the WITH ENCRYPTION option. If you do not use WITH ENCRYPTION, users can use SQL Server Enterprise Manager or execute the **sp_helptext** system stored procedure to view the text of stored procedures located in the **syscomments** system table.

Executing Stored Procedures

Topic Objective
To describe how to execute a stored procedure.

Lead-in
You can execute a stored procedure by itself or as part of an INSERT statement.



- **Executing a Stored Procedure by Itself**

```
EXEC OverdueOrders
```

- **Executing a Stored Procedure Within an INSERT Statement**

```
INSERT INTO Customers
EXEC EmployeeCustomer
```

Delivery Tip
It is recommended that you teach students to use the EXECUTE statement for single stored procedures to help them avoid errors when executing stored procedures in batches.

You can execute a stored procedure by itself or as part of an INSERT statement. You must have been granted EXECUTE permission on the stored procedure.

Executing a Stored Procedure by Itself

You can execute a stored procedure by issuing the EXECUTE statement along with the name of the stored procedure and any parameters.

Syntax

```
[ [ EXEC [ UTE ] ]
    {
        [ @return_status = ]
            { procedure_name [ ;number ] | @procedure_name_var
    }
    [ [ @parameter = ] { value | @variable [ OUTPUT ] | [ DEFAULT ] ]
        [ ,...n ]
  [ WITH RECOMPILE ]
```

Example 1

The following statement executes a stored procedure that lists all overdue orders in the **Northwind** database.

```
EXEC OverdueOrders
```

Stored procedures that are executed within an INSERT statement must return a relational result set. For example, you could not use a COMPUTE BY statement.

Executing a Stored Procedure Within an INSERT Statement

The INSERT statement can populate a local table with a result set that is returned from a local or remote stored procedure. SQL Server loads the table with data that is returned from SELECT statements in the stored procedure. The table must already exist, and data types must match.

Example 2

The following statement creates the **EmployeeCustomer** stored procedure, which inserts employees into the **Customers** table of the **Northwind** database.

```
USE Northwind
GO
CREATE PROC dbo.EmployeeCustomer
AS
SELECT
   UPPER(SUBSTRING(LastName, 1, 4)+SUBSTRING(FirstName, 1,1)),
   'Northwind Traders', RTRIM(FirstName)+' '+LastName,
   'Employee', Address, City, Region, PostalCode, Country,
   ('(206) 555-1234'+' x'+Extension), NULL
FROM Employees
WHERE HireDate < GETDATE ()
GO
```

The following statements execute the **EmployeeCustomer** stored procedure.

```
INSERT INTO Customers
EXEC EmployeeCustomer
```

The number of employees hired earlier than today's date is added to the **Customers** table.

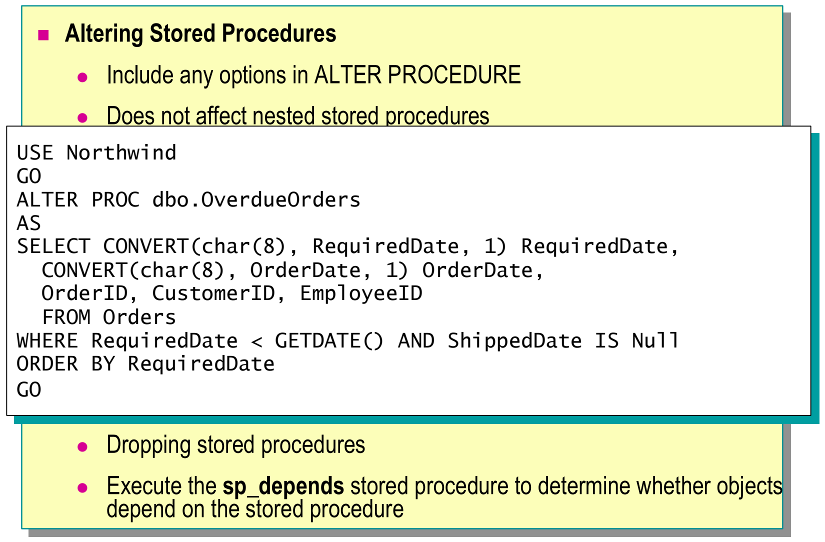
Result

```
(9 row(s) affected)
```

Altering and Dropping Stored Procedures

Topic Objective
To introduce the ALTER PROCEDURE statement.

Lead-in
Stored procedures are often modified in response to requests from users or to changes in the underlying table definitions.



- **Altering Stored Procedures**
  - Include any options in ALTER PROCEDURE
  - Does not affect nested stored procedures

```
USE Northwind
GO
ALTER PROC dbo.OverdueOrders
AS
SELECT CONVERT(char(8), RequiredDate, 1) RequiredDate,
  CONVERT(char(8), OrderDate, 1) OrderDate,
  OrderID, CustomerID, EmployeeID
  FROM Orders
WHERE RequiredDate < GETDATE() AND ShippedDate IS Null
ORDER BY RequiredDate
GO
```

  - Dropping stored procedures
  - Execute the **sp_depends** stored procedure to determine whether objects depend on the stored procedure

Stored procedures are often modified in response to requests from users or to changes in the underlying table definitions.

Altering Stored Procedures

To modify an existing stored procedure and retain permission assignments, use the ALTER PROCEDURE statement. SQL Server replaces the previous definition of the stored procedure when it is altered with ALTER PROCEDURE.

Caution It is strongly recommended that you do not modify system stored procedures directly. Instead, create a user-defined system stored procedure by copying the statements from an existing system stored procedure, and then modify it to meet your needs.

Consider the following facts when you use the ALTER PROCEDURE statement:

- If you want to modify a stored procedure that was created with any options, such as the WITH ENCRYPTION option, you must include the option in the ALTER PROCEDURE statement to retain the functionality that the option provides.
- ALTER PROCEDURE alters only a single procedure. If your procedure calls other stored procedures, the nested stored procedures are not affected.
- Permission to execute this statement defaults to the creators of the initial stored procedure, members of the **sysadmin** server role, and members of the **db_owner** and **db_ddladmin** fixed database roles. You cannot grant permission to execute ALTER PROCEDURE.

Syntax

```
ALTER PROC [ EDURE ] procedure_name [ ; number ]
    [ { @parameter data_type }
        [ VARYING ] [ = default ] [ OUTPUT ]
    ] [ ,...n ]
  [ WITH
    { RECOMPILE | ENCRYPTION
        | RECOMPILE , ENCRYPTION
    }
  ]
  [ FOR REPLICATION ]
  AS
    sql_statement [ ...n ]
```

Example

The following example modifies the **OverdueOrders** stored procedure to select only specific column names rather than all columns from the **Orders** table, as well as to sort the result set.

```
USE Northwind
GO
ALTER PROC dbo.OverdueOrders
AS
SELECT CONVERT(char(8), RequiredDate, 1) RequiredDate,
  CONVERT(char(8), OrderDate, 1) OrderDate,
  OrderID, CustomerID, EmployeeID
  FROM Orders
WHERE RequiredDate < GETDATE() AND ShippedDate IS Null
ORDER BY RequiredDate
GO
```

The following statement executes the **OverdueOrders** stored procedure.

```
EXEC OverdueOrders
```

Result

If the **OverdueOrders** stored procedure is executed based on today's date, the result set will look similar to the following.

```
RequiredDate   OrderDate   OrderID   CustomerID   EmployeeID
05/06/98       04/08/98    11008     ERNSH        7
05/11/98       04/13/98    11019     RANCH        6
05/19/98       04/21/98    11039     LINOD        1
05/21/98       04/22/98    11040     GREAL        4
05/25/98       04/23/98    11045     BOTTM        6
.
.
.
(21 row(s) affected)
```

Dropping Stored Procedures

Use the DROP PROCEDURE statement to remove user-defined stored procedures from the current database.

Before you drop a stored procedure, execute the **sp_depends** stored procedure to determine whether objects depend on the stored procedure.

Syntax

DROP PROCEDURE { *procedure* } [,...n]

Example

This example drops the **OverdueOrders** stored procedure.

```
USE Northwind
GO
DROP PROC dbo.OverdueOrders
GO
```

Lab A: Creating Stored Procedures

Topic Objective
To introduce the lab.

Lead-in
In this lab, you will create stored procedures and display information about them.

Explain the lab objectives.

Objectives

After completing this lab, you will be able to:

- Create a stored procedure by using SQL Query Analyzer.
- Display information about stored procedures that you create.

Prerequisites

Before working on this lab, you must have:

- Script files for this lab, which are located in C:\Moc\2073A\Labfiles\L09.
- Answer files for this lab, which are located in C:\Moc\2073A\Labfiles\L09\Answers.

Lab Setup

To complete this lab, you must have either:

- Completed the prior lab, or
- Executed the C:\Moc\2073A\Batches\Restore09A.cmd batch file.

  This command file restores the **ClassNorthwind** database to a state required for this lab.

For More Information

If you require help in executing files, search SQL Query Analyzer Help for "Execute a query".

Other resources that you can use include:

- The **Northwind** database schema.
- Microsoft SQL Server Books Online.

Scenario

The organization of the classroom is meant to simulate that of a worldwide trading firm named Northwind Traders. Its fictitious domain name is nwtraders.msft. The primary DNS server for nwtraders.msft is the instructor computer, which has an Internet Protocol (IP) address of 192.168.*x*.200 (where *x* is the assigned classroom number). The name of the instructor computer is London.

The following table provides the user name, computer name, and IP address for each student computer in the fictitious nwtraders.msft domain. Find the user name for your computer, and make a note of it.

User name	Computer name	IP address
SQLAdmin1	Vancouver	192.168.*x*.1
SQLAdmin2	Denver	192.168.*x*.2
SQLAdmin3	Perth	192.168.*x*.3
SQLAdmin4	Brisbane	192.168.*x*.4
SQLAdmin5	Lisbon	192.168.*x*.5
SQLAdmin6	Bonn	192.168.*x*.6
SQLAdmin7	Lima	192.168.*x*.7
SQLAdmin8	Santiago	192.168.*x*.8
SQLAdmin9	Bangalore	192.168.*x*.9
SQLAdmin10	Singapore	192.168.*x*.10
SQLAdmin11	Casablanca	192.168.*x*.11
SQLAdmin12	Tunis	192.168.*x*.12
SQLAdmin13	Acapulco	192.168.*x*.13
SQLAdmin14	Miami	192.168.*x*.14
SQLAdmin15	Auckland	192.168.*x*.15
SQLAdmin16	Suva	192.168.*x*.16
SQLAdmin17	Stockholm	192.168.*x*.17
SQLAdmin18	Moscow	192.168.*x*.18
SQLAdmin19	Caracas	192.168.*x*.19
SQLAdmin20	Montevideo	192.168.*x*.20
SQLAdmin21	Manila	192.168.*x*.21
SQLAdmin22	Tokyo	192.168.*x*.22
SQLAdmin23	Khartoum	192.168.*x*.23
SQLAdmin24	Nairobi	192.168.*x*.24

Estimated time to complete this lab: 15 minutes

Exercise 1
Writing and Executing a Stored Procedure

In this exercise, you will create a stored procedure that lists the five most expensive products ordered by price.

► To create a stored procedure by using SQL Query Analyzer

In this procedure, you will create a stored procedure that lists the five most expensive products. C:\Moc\2073A\Labfiles\L09\Answers\FiveMostExpensiveProducts.sql is a completed script for this procedure.

1. Log on to the **NWTraders** classroom domain by using the information in the following table.

Option	Value
User name	**SQLAdmin*x*** (where *x* corresponds to your computer name as designated in the nwtraders.msft classroom domain)
Password	**password**

2. Open SQL Query Analyzer and, if requested, log in to the (local) server with Microsoft Windows® Authentication.

   You have permission to log in to and administer SQL Server because you are logged as **SQLAdmin*x***, which is a member of the Microsoft Windows 2000 local group, Administrators. All members of this group are automatically mapped to the SQL Server **sysadmin** role.
3. Verify that you are using the **ClassNorthwind** database.
4. Write a query against the **Products** table that lists only the product name and unit price. Limit the rows returned to the five most expensive products, and order the result set by unit price.
5. Test your query to ensure that it returns the expected result set.
6. Modify your query to create a stored procedure named **FiveMostExpensiveProducts**.
7. Save your script as C:\MOC\2073A\Labfiles\L09\FiveMostExpensiveProducts.sql.
8. Execute your stored procedure to verify that it works as expected.

   What are the five most expensive products?

   Côte de Blaye, Thüringer Rostbratwurst, Mishi Kobe Niku, Sir Rodney's Marmalade, and Carnarvon Tigers.

Exercise 2
Locating Stored Procedure Information

In this exercise, you will execute system stored procedures and use SQL Server Enterprise Manager and SQL Query Analyzer to display information about the stored procedures that you have created.

► To display stored procedure definitions

In this procedure, you will use SQL Server Enterprise Manager and SQL Query Analyzer to display stored procedure definitions.

1. Open SQL Server Enterprise Manager.
2. Expand your server, expand **Databases**, expand **ClassNorthwind**, and then expand **Stored Procedures**.
3. In the details pane, right-click **FiveMostExpensiveProducts**, and then click **Properties**.
4. Review the stored procedure definition.
5. Open SQL Query Analyzer.
6. Verify that you are using the **ClassNorthwind** database.
7. In the query window, execute the following system stored procedure.

   ```
   sp_helptext FiveMostExpensiveProducts
   ```

8. Review the stored procedure definition.

► To display metadata information about stored procedures

In this procedure, you will use the OBJECT_ID and OBJECTPROPERTY functions to display metadata about stored procedures.

1. Using SQL QueryAnalyzer, determine the object ID of the **FiveMostExpensiveProducts** stored procedure by executing the following statement:

   ```
   SELECT OBJECT_ID('FiveMostExpensiveProducts')
   ```

   Write the object ID below.

   ____

2. Execute the following statement to determine whether the ANSI NULLs connection settings were turned on when you created the **FiveMostExpensiveProducts** stored procedure. Substitute the object ID of your stored procedure for *x*.

   ```
   SELECT OBJECTPROPERTY(x, 'ExecIsAnsiNullsOn')
   ```

   What was the result?

   1 = True.

   ____

   ____

3. Execute the following statement to determine whether the ANSI quoted identifer connection setting was turned on when you created the **FiveMostExpensiveProducts** stored procedure. Substitute the object ID of your stored procedure for *x*.

   ```
   SELECT OBJECTPROPERTY(x, 'ExecIsQuotedIdentOn')
   ```

   What was the result?

   1=True.

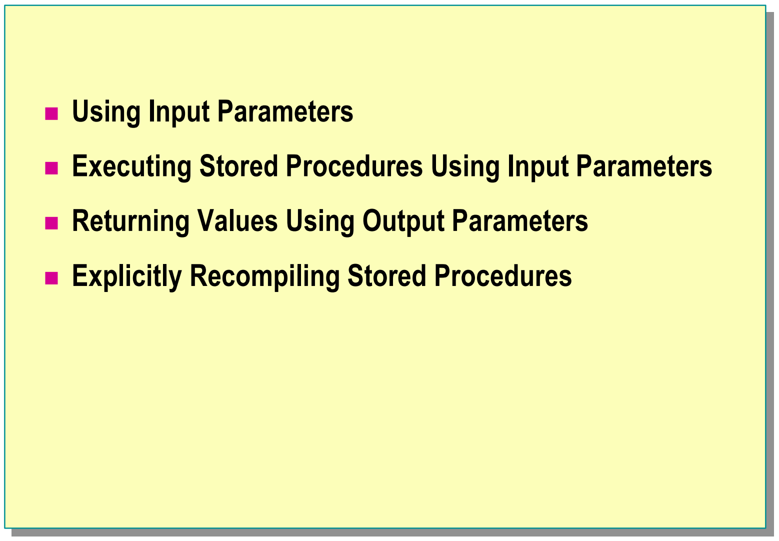
   ____

   ____

◆ Using Parameters in Stored Procedures

Topic Objective
To introduce the topics in this section.

Lead-in
In this section, we'll cover…



- **Using Input Parameters**
- **Executing Stored Procedures Using Input Parameters**
- **Returning Values Using Output Parameters**
- **Explicitly Recompiling Stored Procedures**

Parameters extend the functionality of stored procedures. You can pass information into and out of stored procedures by using parameters. They enable you to use the same stored procedure to search a database many times.

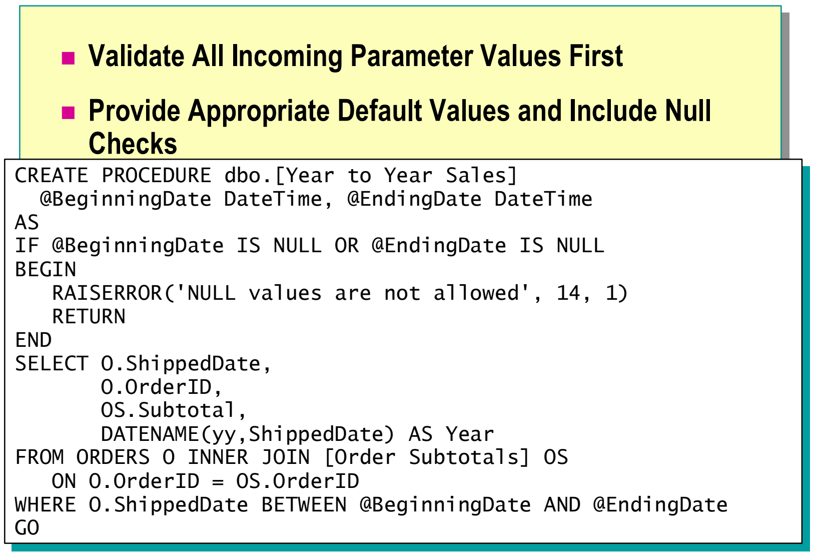
For example, you can add a parameter to a stored procedure that searches the **Employee** table for employees whose hire dates match a date that you specify. You then can execute the stored procedure each time that you want to specify a different hire date.

SQL Server supports two types of parameters: input parameters and output parameters.

Using Input Parameters

Topic Objective
To introduce input parameters.

Lead-in
Input parameters allow information to be passed into a stored procedure.



- **Validate All Incoming Parameter Values First**
- **Provide Appropriate Default Values and Include Null Checks**

```
CREATE PROCEDURE dbo.[Year to Year Sales]
  @BeginningDate DateTime, @EndingDate DateTime
AS
IF @BeginningDate IS NULL OR @EndingDate IS NULL
BEGIN
   RAISERROR('NULL values are not allowed', 14, 1)
   RETURN
END
SELECT O.ShippedDate,
       O.OrderID,
       OS.Subtotal,
       DATENAME(yy,ShippedDate) AS Year
FROM ORDERS O INNER JOIN [Order Subtotals] OS
   ON O.OrderID = OS.OrderID
WHERE O.ShippedDate BETWEEN @BeginningDate AND @EndingDate
GO
```

Input parameters allow information to be passed into a stored procedure. To define a stored procedure that accepts input parameters, you declare one or more variables as parameters in the CREATE PROCEDURE statement.

Partial Syntax

@parameter data_type [= *default*]

Consider the following facts and guidelines when you specify parameters:

- All incoming parameter values should be checked at the beginning of a stored procedure to trap missing and invalid values early.
- You should provide appropriate default values for a parameter.

  If a default is defined, a user can execute the stored procedure without specifying a value for that parameter.

Note Parameter defaults must be constants or NULL. When you specify NULL as a default value for a parameter, you must use =Null; IS NULL will not work because the syntax does not support the ANSI NULL designation.

- The maximum number of parameters in a stored procedure is 1,024.
- The maximum number of local variables in a stored procedure is limited only by available memory.
- Parameters are local to a stored procedure. The same parameter names can be used in other stored procedures.

Parameter information is stored in the **syscolumns** system table.

Example

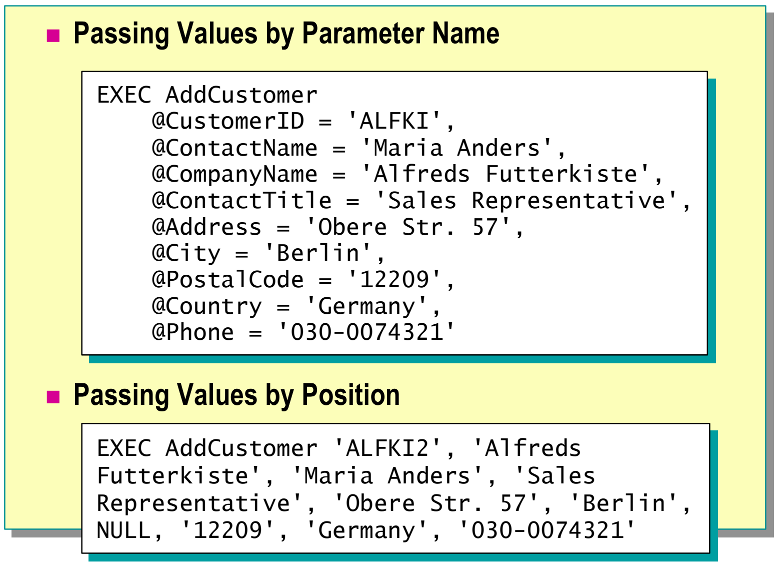
The following example creates the **Year to Year Sales** stored procedure, which returns all sales between specific dates.

```
CREATE PROCEDURE dbo.[Year to Year Sales]
  @BeginningDate DateTime, @EndingDate DateTime
AS
IF @BeginningDate IS NULL OR @EndingDate IS NULL
BEGIN
   RAISERROR('NULL values are not allowed', 14, 1)
   RETURN
END
SELECT O.ShippedDate,
       O.OrderID,
       OS.Subtotal,
       DATENAME(yy,ShippedDate) AS Year
FROM ORDERS O INNER JOIN [Order Subtotals] OS
   ON O.OrderID = OS.OrderID
WHERE O.ShippedDate BETWEEN @BeginningDate AND @EndingDate
GO
```

Executing Stored Procedures Using Input Parameters

Topic Objective
To show how to execute stored procedures by using the EXECUTE statement with parameters.

Lead-in
You can pass parameter values to a stored procedure by reference or by position.



- **Passing Values by Parameter Name**

```
EXEC AddCustomer
    @CustomerID = 'ALFKI',
    @ContactName = 'Maria Anders',
    @CompanyName = 'Alfreds Futterkiste',
    @ContactTitle = 'Sales Representative',
    @Address = 'Obere Str. 57',
    @City = 'Berlin',
    @PostalCode = '12209',
    @Country = 'Germany',
    @Phone = '030-0074321'
```

- **Passing Values by Position**

```
EXEC AddCustomer 'ALFKI2', 'Alfreds
Futterkiste', 'Maria Anders', 'Sales
Representative', 'Obere Str. 57', 'Berlin',
NULL, '12209', 'Germany', '030-0074321'
```

You can set the value of a parameter by either passing the value to the stored procedure, by parameter name, or by position. You should not mix the different formats when you supply values.

Delivery Tip
Recommend passing values by parameter name for readability.

Passing Values by Parameter Name

Specifying a parameter in an EXECUTE statement in the format *@parameter* = *value* is referred to as *passing by parameter name*. When you pass values by parameter name, the parameter values can be specified in any order, and you can omit parameters that allow null values or that have a default.

The default value of a parameter, if defined for the parameter in the stored procedure, is used when:

- No value for the parameter is specified when the stored procedure is executed.
- The DEFAULT keyword is specified as the value for the parameter.

Syntax

```
[ [ EXEC [ UTE ] ]
    {
        [ @return_status = ]
            { procedure_name [ ;number ] | @procedure_name_var
    }
    [ [ @parameter = ] { value | @variable [ OUTPUT ] | [ DEFAULT ] ]
        [ ,...n ]
 [ WITH RECOMPILE ]
```

Partial Example 1

The following partial example creates the **AddCustomer** stored procedure, which adds a new customer to the **Northwind** database. Notice that all variables except for **CustomerID** and **CompanyName** are specified to allow a null value.

```
USE Northwind
GO
CREATE PROCEDURE dbo.AddCustomer
    @CustomerID      nchar (5),
    @CompanyName     nvarchar (40),
    @ContactName     nvarchar (30) = NULL,
    @ContactTitle    nvarchar (30) = NULL,
    @Address         nvarchar (60) = NULL,
    @City            nvarchar (15) = NULL,
    @Region          nvarchar (15) = NULL,
    @PostalCode      nvarchar (10) = NULL,
    @Country         nvarchar (15) = NULL,
    @Phone           nvarchar (24) = NULL,
    @Fax             nvarchar (24) = NULL
    AS
.
.
.
```

Partial Example 2

The following example passes values by parameter name to the **AddCustomer** stored procedure. Notice that the order of the values is different from the CREATE PROCEDURE statement.

Also notice that values for the **@Region** and **@Fax** parameters are not specified. If the Region and Fax columns in the table allow null values, the **AddCustomer** stored procedure will execute successfully. However, if the Region and Fax columns do not allow null values, you must pass a value to a parameter, regardless of whether you have defined the parameter to allow a null value.

```
EXEC AddCustomer
    @CustomerID = 'ALFKI',
    @ContactName = 'Maria Anders',
    @CompanyName = 'Alfreds Futterkiste',
    @ContactTitle = 'Sales Representative',
    @Address = 'Obere Str. 57',
    @City = 'Berlin',
    @PostalCode = '12209',
    @Country = 'Germany',
    @Phone = '030-0074321'
.
.
.
```

Passing Values by Position

Passing only values (without reference to the parameters to which they are being passed) is referred to as *passing values by position*. When you specify only a value, the parameter values must be listed in the order in which they are defined in the CREATE PROCEDURE statement.

When you pass values by position, you can omit parameters where defaults exist, but you cannot interrupt the sequence. For example, if a stored procedure has five parameters, you can omit both the fourth and fifth parameters, but you cannot omit the fourth parameter and specify the fifth.

Partial Example 3

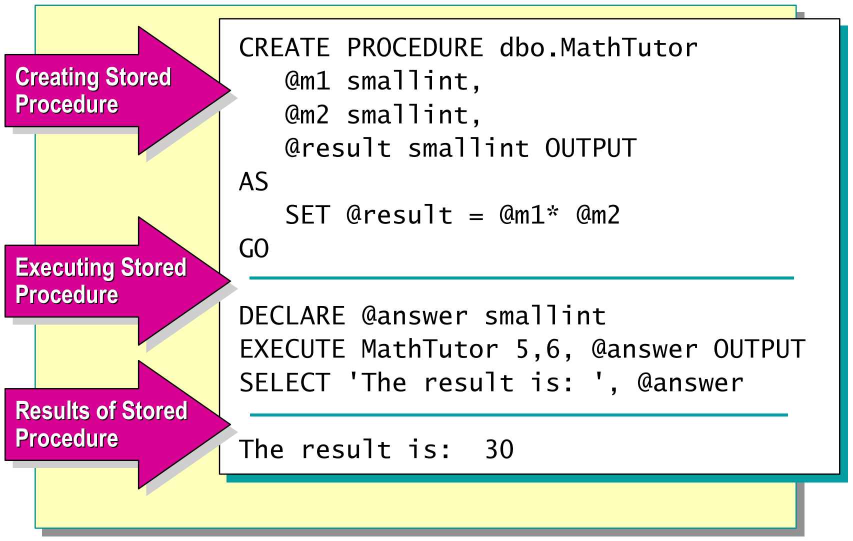
The following example passes values by position to the **AddCustomer** stored procedure. Notice that the **@Region** and **@Fax** parameter have no values. However, only the **@Region** parameter is supplied with NULL. The **@Fax** parameter is omitted because it is the last parameter.

```
EXEC AddCustomer 'ALFKI2', 'Alfreds Futterkiste', 'Maria
Anders', 'Sales Representative', 'Obere Str. 57', 'Berlin',
NULL, '12209', 'Germany', '030-0074321'
```

Returning Values Using Output Parameters

Topic Objective
To show how the OUTPUT keyword can produce a value.

Lead-in
A stored procedure can return several values. Each must be defined as a variable by using the OUTPUT keyword in both the stored procedure and the calling statements.


Creating Stored Procedure
Executing Stored Procedure
Results of Stored Procedure

```
CREATE PROCEDURE dbo.MathTutor
   @m1 smallint,
   @m2 smallint,
   @result smallint OUTPUT
AS
   SET @result = @m1* @m2
GO
```

```
DECLARE @answer smallint
EXECUTE MathTutor 5,6, @answer OUTPUT
SELECT 'The result is: ', @answer
```

```
The result is:  30
```

Delivery Tip
Discuss the slide example in detail and demonstrate how to create and use the **MathTutor** stored procedure.

Stored procedures can return information to the calling stored procedure or client with output parameters (variables designated with the OUTPUT keyword). By using output parameters, any changes to the parameter that result from the execution of the stored procedure can be retained, even after the stored procedure completes execution.

To use an output parameter, you must specify the OUTPUT keyword in both the CREATE PROCEDURE and EXECUTE statements. If the keyword OUTPUT is omitted when the stored procedure is executed, the stored procedure still executes but does not return a value. Output parameters have the following characteristics:

- The calling statement must contain a variable name to receive the return value. It is not possible to pass constants.
- You can use the variable subsequently in additional Transact-SQL statements in the batch or the calling stored procedure.
- The parameter can be of any data type, except **text** or **image**.
- They can be cursor placeholders.

Example 1

This example creates a **MathTutor** stored procedure that calculates the product of two numbers. This example uses the SET statement. However, you can also use the SELECT statement to dynamically concatenate a string. A SET statement requires that you declare a variable in order to print the string “The result is:”

```
CREATE PROCEDURE dbo.MathTutor
   @m1 smallint,
   @m2 smallint,
   @result smallint OUTPUT
AS
   SET @result = @m1* @m2
GO
```

This batch calls the **MathTutor** stored procedure and passes the values of 5 and 6. These values become variables, which are entered into the SET statement.

```
DECLARE @answer smallint
EXECUTE MathTutor 5,6, @answer OUTPUT
SELECT 'The result is: ', @answer
```

Result

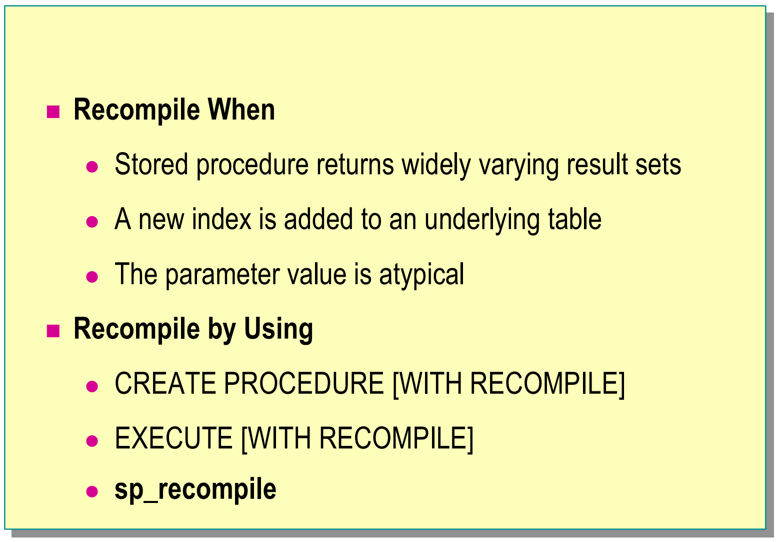
The **@result** parameter is designated with the OUTPUT keyword. SQL Server prints the content of the **@result** variable when you execute the **MathTutor** stored procedure. The result variable is defined as the product of the two values, 5 and 6.

```
The result is: 30
```

Explicitly Recompiling Stored Procedures

Topic Objective
To discuss when to recompile stored procedures and which options to use.

Lead-in
Stored procedures can be recompiled explicitly, but you should do so infrequently.



- **Recompile When**
  - Stored procedure returns widely varying result sets
  - A new index is added to an underlying table
  - The parameter value is atypical
- **Recompile by Using**
  - CREATE PROCEDURE [WITH RECOMPILE]
  - EXECUTE [WITH RECOMPILE]
  - **sp_recompile**

Stored procedures can be recompiled explicitly, but you should do so infrequently, and only when:

- Parameter values are passed to a stored procedure that returns widely varying result sets.
- A new index is added to an underlying table from which a stored procedure might benefit.
- The parameter value that you are supplying is atypical.

SQL Server provides three methods for recompiling a stored procedure explicitly.

CREATE PROCEDURE…[WITH RECOMPILE]

The CREATE PROCEDURE...[WITH RECOMPILE] statement indicates that SQL Server does not cache a plan for this stored procedure. Instead, the option recompiles the stored procedure each time that it is executed.

Example 1

The following example creates a stored procedure called **OrderCount** that is recompiled each time that it is executed.

```
USE Northwind
GO
CREATE PROC dbo.OrderCount
@CustomerID nchar (10)
WITH RECOMPILE
AS
   SELECT count(*) FROM [Orders Qry]
   WHERE CustomerID = @CustomerID
GO
```

EXECUTE…[WITH RECOMPILE]

The EXECUTE...[WITH RECOMPILE] statement creates a new execution plan each time that the procedure is executed, if you specify WITH RECOMPILE. The new execution plan is not stored in the cache. Use this option if the parameter that you are passing varies greatly from those that are usually passed to the stored procedure. Because this optimized plan is the exception rather than the rule, when execution is completed, you should re-execute the stored procedure by using a parameter that is typically passed. This option is also useful if the data has changed significantly since the stored procedure was last compiled.

Example 2

This example recompiles the **CustomerInfo** stored procedure at the time that it is executed.

```
EXEC  CustomerInfo WITH RECOMPILE
```

sp_recompile

The **sp_recompile** system stored procedure recompiles the specified stored procedure or trigger the next time that it is executed. If the **@objname** parameter specifies a table or view, all stored procedures that use the named object will be recompiled the next time that they are executed.

Use the **sp_recompile** system stored procedure with the *tablename* option if you have added a new index to an underlying table that the stored procedure references, and if you believe that the performance of the stored procedure will benefit from the new index.

Example 3

This example recompiles all stored procedures or triggers that reference the **Customers** table in the **Northwind** database.

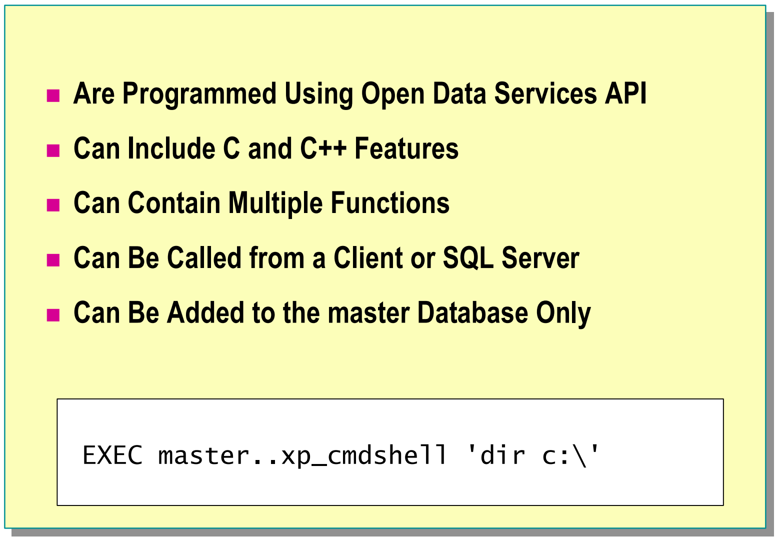
```
EXEC sp_recompile Customers
```

Note You can use DBCC FREEPROCCACHE to clear all stored procedure plans from the cache.

Executing Extended Stored Procedures

Topic Objective
To describe the features of extended stored procedures.

Lead-in
Extended stored procedures…



- Are Programmed Using Open Data Services API
- Can Include C and C++ Features
- Can Contain Multiple Functions
- Can Be Called from a Client or SQL Server
- Can Be Added to the master Database Only

```
EXEC master..xp_cmdshell 'dir c:\'
```

Extended stored procedures are functions inside a DLL that increase SQL Server functionality. They are executed in the same way as stored procedures, and they support input parameters, return status codes, and output parameters.

Example 1

This example executes the **xp_cmdshell** extended stored procedure that displays a list of files and subdirectories by executing the **dir** operating system command.

```
EXEC master..xp_cmdshell 'dir c:\ '
```

Extended stored procedures:

- Are programmed by using the Open Data Services (ODS) application programming interface (API).
- Allow you to create your own external routines in programming languages such as Microsoft Visual C++® and Visual C.
- Can contain multiple functions.
- Can be called from a client or SQL Server.
- Can be added to the **master** database only.

Note You can execute an extended stored procedure from the **master** database only, or by explicitly specifying the location **master**. You can also create a user-defined system stored procedure that calls the extended stored procedure. This allows you to execute the extended stored procedure from within any database.

Delivery Tip
Demonstrate how to execute the **xp_cmdshell** extended stored procedure to see files that are listed in one of the SQL Server directories.

The following table includes some commonly used extended stored procedures.

Extended stored procedure	Description
xp_cmdshell	Executes a given command string as an operating system command shell and returns output as rows of text
xp_logevent	Logs a user-defined message in a SQL Server log file or in the Windows 2000 Event Viewer.

Example 2

This example executes the **sp_helptext** system stored procedure to display the name of the DLL that contains the **xp_cmdshell** extended stored procedure.

```
EXEC master..sp_helptext xp_cmdshell
```

Result

This result displays the DLL that contains the **xp_cmdshell** extended stored procedure.

```
xplog70.dll
```

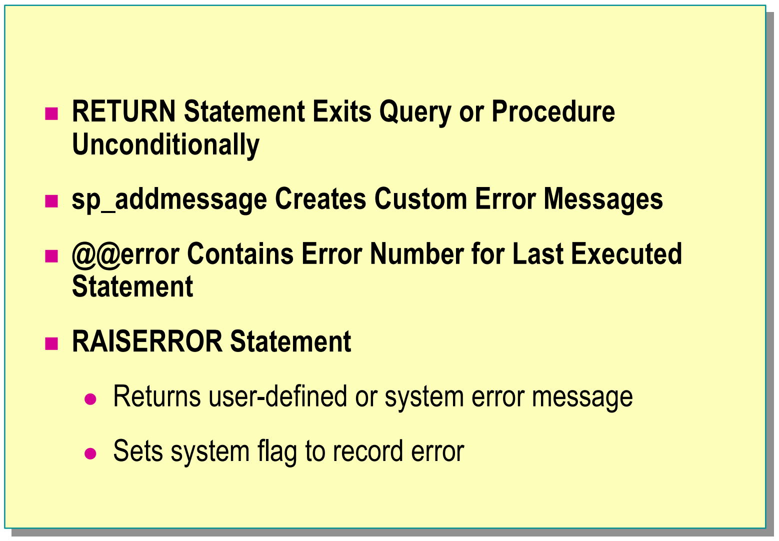
You can also create your own extended stored procedures. Generally, you call extended stored procedures to communicate with other applications or the operating system. For example, Sqlmap70.dll allows you to send e-mail messages from within SQL Server by using the **xp_sendmail** extended stored procedure.

When you select **Development Tools** during SQL Server Setup, SQL Server installs sample extended stored procedures in the C:\Program Files\Microsoft SQL Server\80\Tools\Devtools\Samples\ODS folder as a compressed self-extracting executable.

Handling Error Messages

Topic Objective
To discuss the various options for creating error messages in stored procedures.

Lead-in
To enhance the effectiveness of stored procedures, you should include error messages that communicate transaction status to the user.



- **RETURN Statement Exits Query or Procedure Unconditionally**
- **sp_addmessage Creates Custom Error Messages**
- **@@error Contains Error Number for Last Executed Statement**
- **RAISERROR Statement**
  - Returns user-defined or system error message
  - Sets system flag to record error

A demonstration of how error messages work is found on the following pages.

To enhance the effectiveness of stored procedures, you should include error messages that communicate transaction status (success or failure) to the user. You should perform the task logic, business logic, and error checking *before* you begin transactions, and you should keep your transactions short.

You can use coding strategies, such as existence checks, to recognize errors. When an error occurs, provide as much information as possible to the client. You can check the following in your error handling logic: return codes, SQL Server errors, and custom error messages.

RETURN Statement

The RETURN statement exits from a query or stored procedure unconditionally. It also can return an integer status value (return code).

A return value of 0 indicates success. Return values 0 through -14 are currently in use, and return values from -15 through -99 are reserved for future use. If a user-defined return value is not provided, the SQL Server value is used. User-defined return values always take precedence over those that SQL Server supplies.

Example 1

This example creates the **GetOrders** stored procedure that retrieves information from the **Orders** and **Customers** tables by querying the **Orders Qry** view. The RETURN statement in the **GetOrders** stored procedure returns the total number of rows from the SELECT statement to another stored procedure. You could also nest the **GetOrders** stored procedure within another stored procedure.

```
USE Northwind
GO
CREATE PROCEDURE dbo.GetOrders
   @CustomerID nchar (10)
AS
   SELECT OrderID, CustomerID, EmployeeID
   FROM [Orders Qry]
   WHERE CustomerID = @CustomerID
   RETURN (@@ROWCOUNT)
GO
```

sp_addmessage

This stored procedure allows developers to create custom error messages. SQL Server treats both system and custom error messages the same way. All messages are stored in the **sysmessages** table in the **master** database. These error messages also can be written automatically to the Windows 2000 application log.

Example 2

This example creates a user-defined error message that requires the message to be written to the Windows 2000 application log when it occurs.

```
EXEC sp_addmessage
@msgnum = 50010,
@severity = 10,
@msgtext = 'Customer cannot be deleted.',
@with_log = 'true'
```

@@error

This system function contains the error number for the most recently executed Transact-SQL statement. It is cleared and reset with each statement that is executed. A value of 0 is returned if the statement executes successfully. You can use the **@@error** system function to detect a specific error number or to exit a stored procedure conditionally.

Example 3

This example creates the **AddSupplierProduct** stored procedure in the **Northwind** database. This stored procedure uses the **@@error** system function to determine whether an error occurs when each INSERT statement is executed. If the error does occur, the transaction is rolled back.

```
USE Northwind
GO
CREATE PROCEDURE dbo.AddSupplierProduct
        @CompanyName nvarchar (40) = NULL,
        @ContactName nvarchar (40) = NULL,
        @ContactTitle nvarchar (40)= NULL,
        @Address nvarchar (60) = NULL,
        @City nvarchar (15) = NULL,
        @Region nvarchar (40) = NULL,
        @PostalCode nvarchar (10) = NULL,
        @Country nvarchar (15) = NULL,
        @Phone nvarchar (24) = NULL,
        @Fax nvarchar (24) = NULL,
        @HomePage ntext = NULL,
        @ProductName nvarchar (40) = NULL,
        @CategoryID int = NULL,
        @QuantityPerUnit nvarchar (20) = NULL,
        @UnitPrice money = NULL,
        @UnitsInStock smallint = NULL,
        @UnitsOnOrder smallint = NULL,
        @ReorderLevel smallint = NULL,
        @Discontinued bit  = NULL
AS
BEGIN TRANSACTION
   INSERT Suppliers (
     CompanyName,
     ContactName,
     Address,
     City,
     Region,
     PostalCode,
     Country,
     Phone)
   VALUES (
     @CompanyName,
     @ContactName,
     @Address,
     @City,
     @Region,
     @PostalCode,
     @Country,
     @Phone)
  IF @@error <> 0
     BEGIN
       ROLLBACK TRAN
          RETURN
      END
  DECLARE @InsertSupplierID int
  SELECT @InsertSupplierID=@@identity
  INSERT Products (
     ProductName,
     SupplierID,
     CategoryID,
     QuantityPerUnit,
     Discontinued)
  VALUES (
     @ProductName,
     @InsertSupplierID,
     @CategoryID,
     @QuantityPerUnit,
     @Discontinued)
  IF @@error <> 0
      BEGIN
          ROLLBACK TRAN
              RETURN
      END
COMMIT TRANSACTION
```

RAISERROR Statement

The RAISERROR statement returns a user-defined error message and sets a system flag to record that an error has occurred. You must specify an error severity level and message state when using the RAISERROR statement.

The RAISERROR statement allows the application to retrieve an entry from the **master..sysmessages** system table or build a message dynamically with user-specified severity and state information. The RAISERROR statement can write error messages to the SQL Server Error Log and to the Windows 2000 application log.

Example 4

This example raises a user-defined error message and writes the message to the Windows 2000 application log.

Delivery Tip
The RAISERROR statement requires that you specify the error severity level and message states.

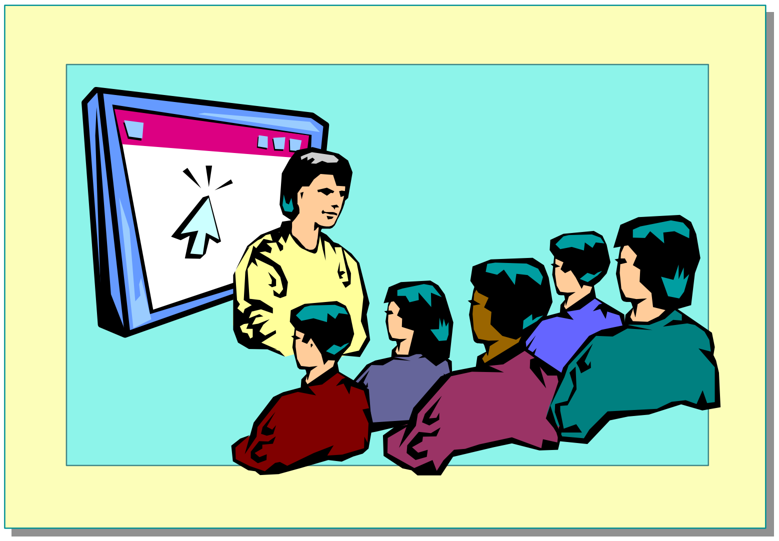
```
RAISERROR(50010, 16, 1) WITH LOG
```

Notes The PRINT statement returns a user-defined message to the message handler of the client; however, unlike the RAISERROR statement, the PRINT statement does not store the error number in the **@@error** system function.

Demonstration: Handling Error Messages

Topic Objective
To demonstrate handling error messages.

Lead-in
In this demonstration, we will review the stored procedure that you would use to remove a member from the **Northwind** database.



Delivery Tip
You must have reviewed the Instructor Notes for this module for demonstration setup information.

Review the D09_UpdateCustomerPhone.sql script in detail, and then execute it.

Use the D09_TestUpdate.sql script to verify that the error handling logic is working.

Follow this script as the instructor points out the error handling techniques that are included in it.

```
/* UpdateCustomerPhone
Updates a customer phone number
Error checking ensures that a valid customer
identification number is supplied
*/
/*
The following user-defined message supports the
UpdateCustomerPhone stored procedure*/
EXEC sp_addmessage 50010, 16, 'CustomerID not found.',
@replace='replace'
USE Northwind
GO
CREATE PROCEDURE UpdateCustomerPhone
  @CustomerID nchar (5) = NULL,
  @Phone nvarchar (24) = NULL
AS
IF @CustomerID IS NULL
   BEGIN
      PRINT 'You must supply a valid CustomerID.'
      RETURN
END
/* Ensure a valid CustomerID is supplied' */
IF NOT EXISTS
   (SELECT * FROM Customers WHERE CustomerID = @CustomerID)
      BEGIN
         RAISERROR (50010, 16, 1) --Customer not found.
         RETURN
      END
```

Delivery Tip
Point out that the UPDATE transaction starts here. The preceding code is related to error checking.

Explain that this is a typical ratio of error checking to transaction statements.

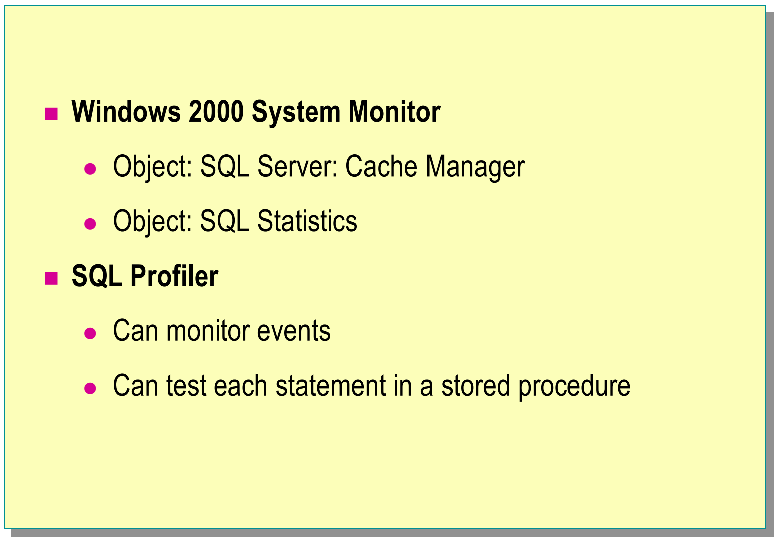
```
BEGIN TRANSACTION
UPDATE Customers
   SET Phone = @Phone
   WHERE CustomerID = @CustomerID

/* Display message that the phone number for CompanyName has
been updated */
SELECT 'The phone number for ' + @CustomerID + ' has been
updated to ' +
@Phone
COMMIT TRANSACTION
GO
```

Performance Considerations

Topic Objective
To explain some of the performance considerations for implementing stored procedures.

Lead-in
You should be aware of these performance considerations when you implement stored procedures.



- **Windows 2000 System Monitor**
  - Object: SQL Server: Cache Manager
  - Object: SQL Statistics
- **SQL Profiler**
  - Can monitor events
  - Can test each statement in a stored procedure

You can use the following tools to help you find the source of performance problems that may be related to stored procedure execution.

Windows 2000 System Monitor

Windows 2000 System Monitor monitors the use of the procedure cache, in addition to many other related activities.

The following objects and counters provide general information about the compiled plans in the procedure cache and the number of recompilations. You can also monitor a specific instance, such as **procedure plan**.

Object	Counters
SQL Server: Cache Manager	Cache Hit-Ratio
	Cache Object Counts
	Cache Pages
	Cache Use Count/sec
SQL Statistics	SQL Re-compilations/sec

SQL Profiler

SQL Profiler is a graphical tool that allows you to monitor events, such as when the stored procedure has started or completed, or when individual Transact-SQL statements within a stored procedure have started or completed. In addition, you can monitor whether a stored procedure is found in the procedure cache.

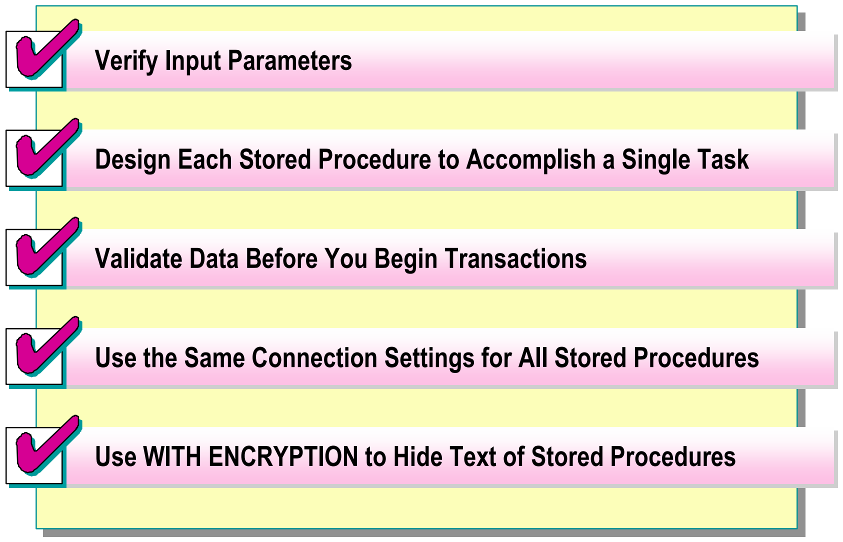
In the development phase of a project, you can also test stored procedure statements one line at a time to confirm that the statements work as expected.

Note Use caution when you create nested stored procedures. Nesting stored procedures adds a level of complexity that can make troubleshooting performance problems difficult.

Recommended Practices

Topic Objective
To summarize tips and techniques for writing effective stored procedures.

Lead-in
Use these recommended practices to implement stored procedures.



- Verify Input Parameters
- Design Each Stored Procedure to Accomplish a Single Task
- Validate Data Before You Begin Transactions
- Use the Same Connection Settings for All Stored Procedures
- Use WITH ENCRYPTION to Hide Text of Stored Procedures

To write more effective and efficient stored procedures, follow these recommended practices:

- Verify all input parameters at the beginning of each stored procedure to trap missing and invalid values early.
- Design each stored procedure to accomplish a single task.
- Perform task and business logic error checking and data validation *before* you begin transactions. Keep your transactions short.
- Use the same connection settings for all stored procedures.
- To conceal the text of stored procedures, use the WITH ENCRYPTION option. Never delete entries from the **syscomments** system table.

Additional information on the following topic is available in SQL Server Books Online.

Topic	Search on
Procedure cache	“SQL Server memory pool”
	“execution plan caching and reuse”

Lab B: Creating Stored Procedures Using Parameters

Topic Objective
To introduce the lab.

Lead-in
In this lab, you will execute a script that creates several stored procedures, and create and modify your own stored procedures.

Objectives

After completing this lab, you will be able to:

- Create a stored procedure by using the Create Stored Procedure Wizard.
- Test a stored procedure that includes error-handling techniques.
- Create custom error messages.
- Create stored procedures that return codes.

Prerequisites

Before working on this lab, you must have:

- Script files for this lab, which are located in C:\Moc\2073A\Labfiles\L09.
- Answer files for this lab, which are located in C:\Moc\2073A\Labfiles\L09\Answers.

Lab Setup

To complete this lab, you must have either:

- Completed the prior lab, or
- Executed the C:\Moc\2073A\Batches\Restore09B.cmd batch file.

  This command file restores the **ClassNorthwind** database to a state required for this lab.

For More Information

If you require help in executing files, search SQL Query Analyzer Help for "Execute a query".

Other resources that you can use include:

- The **Northwind** database schema.
- Microsoft SQL Server Books Online.

Scenario

The organization of the classroom is meant to simulate that of a worldwide trading firm named Northwind Traders. Its fictitious domain name is nwtraders.msft. The primary DNS server for nwtraders.msft is the instructor computer, which has an Internet Protocol (IP) address of 192.168.*x*.200 (where *x* is the assigned classroom number). The name of the instructor computer is London.

The following table provides the user name, computer name, and IP address for each student computer in the fictitious nwtraders.msft domain. Find the user name for your computer, and make a note of it.

User name	Computer name	IP address
SQLAdmin1	Vancouver	192.168.*x*.1
SQLAdmin2	Denver	192.168.*x*.2
SQLAdmin3	Perth	192.168.*x*.3
SQLAdmin4	Brisbane	192.168.*x*.4
SQLAdmin5	Lisbon	192.168.*x*.5
SQLAdmin6	Bonn	192.168.*x*.6
SQLAdmin7	Lima	192.168.*x*.7
SQLAdmin8	Santiago	192.168.*x*.8
SQLAdmin9	Bangalore	192.168.*x*.9
SQLAdmin10	Singapore	192.168.*x*.10
SQLAdmin11	Casablanca	192.168.*x*.11
SQLAdmin12	Tunis	192.168.*x*.12
SQLAdmin13	Acapulco	192.168.*x*.13
SQLAdmin14	Miami	192.168.*x*.14
SQLAdmin15	Auckland	192.168.*x*.15
SQLAdmin16	Suva	192.168.*x*.16
SQLAdmin17	Stockholm	192.168.*x*.17
SQLAdmin18	Moscow	192.168.*x*.18
SQLAdmin19	Caracas	192.168.*x*.19
SQLAdmin20	Montevideo	192.168.*x*.20
SQLAdmin21	Manila	192.168.*x*.21
SQLAdmin22	Tokyo	192.168.*x*.22
SQLAdmin23	Khartoum	192.168.*x*.23
SQLAdmin24	Nairobi	192.168.*x*.24

Estimated time to complete this lab: 45 minutes

Exercise 1
Using the Create Stored Procedure Wizard

In this exercise, you will use the Create Stored Procedure Wizard to create a stored procedure in the **ClassNorthwind** database that updates the phone number of an employee.

► To use the Create Stored Procedure Wizard

In this procedure, you will use the Create Stored Procedure Wizard to create a stored procedure that updates an employee's phone number.

1. Log on to the **NWTraders** classroom domain by using the information in the following table.

Option	Value
User name	**SQLAdmin***x* (where *x* corresponds to your computer name as designated in the nwtraders.msft classroom domain)
Password	**password**

2. Open SQL Server Enterprise Manager.
3. In the console tree, click your server.
4. On the **Tools** menu, click **Wizards**.
5. Expand **Database**, and then double-click Create Stored Procedure Wizard.
6. Select the **ClassNorthwind** database.
7. Create a stored procedure that updates an employee's phone number. The phone number is maintained in the **Employees** table. Select the **Update** action for the **Employees** table.
8. Click the **Edit** button to edit the stored procedure properties.
9. Name your stored procedure **UpdateEmployeePhone**.
10. Include only the **HomePhone** column in the SET clause and only **EmployeeID** in the WHERE clause.
11. In the console tree, expand the **ClassNorthwind** database, and then expand **Stored Procedures**.
12. Verify that the **UpdateEmployeePhone** stored procedure is listed in the details pane.
13. Review the properties of the **UpdateEmployeePhone** stored procedure.

    What parameters were defined in the stored procedure?

    @EmployeeID_1 and **@HomePhone_2**

14. Open SQL Query Analyzer and, if requested, log in to the (local) server with Windows Authentication.

    You have permission to log in to and administer SQL Server because you are logged as **SQLAdmin*x***, which is a member of the Windows 2000 local group, Administrators. All members of this group are automatically mapped to the SQL Server **sysadmin** role.

15. Execute the **UpdateEmployeePhone** stored procedure to verify that it works as expected. Update the phone number to (503) 555-1212 for employee Nancy Davolio, whose EmployeeID is 1.

```
EXEC UpdateemployeePhone
@EmployeeID_1 = 1,
@HomePhone_2 = '(503)555-1212'
```

► To generate a script

In this procedure, you will generate and save a script for the stored procedure that you created with the Create Stored Procedure Wizard.

1. Switch to SQL Server Enterprise Manager.
2. In the console tree, expand **Databases**, expand **ClassNorthwind**, and then click **Stored Procedures**.
3. In the details pane, right-click **UpdateEmployeePhone**, point to **All Tasks**, and then click **Generate SQL Script**.
4. Click **OK** to generate the script.
5. Save the script as **UpdateEmployeePhone.sql**.
6. Open and review the saved script.

Exercise 2
Using Error Handling in Stored Procedures

In this exercise, you will execute a script that creates a stored procedure to add a supplier and a product to the **ClassNorthwind** database. You will then test the error handling contained in this script.

► To create and test a stored procedure

In this procedure, you will open and review a script that creates a stored procedure to add a new supplier and a new product. Then you will test to ensure that the stored procedure executes as expected by using it to insert a new supplier and a new product. Finally, you will test the error handling of the stored procedure.

1. Switch to SQL Query Analyzer.
2. Open C:\Moc\2073A\Labfiles\L09\SupplierProduct.sql, and review its contents.

   What is the benefit of using the **@@error** system function while inserting values into the **Suppliers** and **Products** tables?

   If the insertion fails due to a data type or constraint violation, the transaction is rolled back.

   ____

   ____

3. Execute the script to create the **SupplierProductInsert** stored procedure.
4. Open C:\Moc\2073A\Labfiles\L09\SupplierProductInsert.sql. Modify the script by entering the appropriate values to add a new supplier and a new product. (You may use any values that you want.)
5. Execute the modified script.
6. Test the error handling in the **SupplierProductInsert** stored procedure by modifying the values and placing an in-line comment in front of the **@contactname** parameter. Execute your modified script to ensure that the value will be ignored.

   What error message did you receive?

   You must provide Company Name, Contact Name, Address, City, Region, Postal Code, Country, Phone, Product Name and Discontinued. (Contact Title, Fax, Home Page, Unit Price, Units in Stock, Units on Order, and Reorder Level can be null.)

   ____

   ____

Exercise 3
Customizing Error Messages

In this exercise, you will create a custom error message that will be logged into the Windows 2000 Event Viewer application log that lists the supplier ID that was inserted, along with the SQL Server user who performed the insertion.

► To create a custom error message

In this procedure, you will modify the **SupplierProduct** stored procedure to call custom error messages. C:\Moc\2073A\Labfiles\L09\Answers\CustomErrorAnswer.sql is a complete script for this procedure.

1. Open C:\Moc\2073A\Labfiles\L09\CustomError.sql, review its contents, and then execute it.
2. Search for the comment /* #1 Substitute student code here */, and then add a variable to the **CustomError** stored procedure that will store the value of the user name that inserts the supplier.

   ```
   /* #1 Substitute Student Code Here. */
   DECLARE @UserName nvarchar (60)
   SELECT @UserName = suser_sname()
   ```

   Tip Use the SUSER_SNAME system function.

3. Search for the next comment /* #2 Substitute student code here */. Add a RAISERROR statement that indicates that a new supplier has been added.

   The RAISERROR statement should call error #50018 and pass the parameters for the supplier number and the user who is executing the stored procedure.

   See SQL Server Books Online for additional information about the RAISERROR statement.

   ```
   RAISERROR (50018, 16, 1, @InsertSupplierID, @UserName)
   ```

4. Search for the next comment /* #3 Substitute student code here */ to create the error message number 50018 by using the **sp_addmessage** system stored procedure. Include the Supplier and UserName values in your error message.

   ```
   EXEC sp_addmessage 50018, 16, 'Supplier %d was inserted by
   %s', 'us_english','true'
   ```

5. Execute the script to create the **CustomError** stored procedure.
6. Open C:\Moc\2073A\Labfiles\L09\SupplierProductInsert.sql. Modify the script by entering the appropriate values to add a new supplier and a new product. (You may use any values that you want.)
7. Execute the modified script.
8. Review the results, and then open Event Viewer and view the application log to verify that your information message was recorded.

Exercise 4
Using Return Codes

In this exercise, you will create a stored procedure with the OUTPUT keyword by using the C:\Moc\2073A\Labfiles\L09\Return1.sql script. Then you will execute that stored procedure and test it for different return codes by using the Return2.sql and Return3.sql scripts.

► To create the OrderCount stored procedure

In this procedure, you will create a stored procedure named **OrderCount** that counts the number of unfilled orders for a customer. If the customer has at least one unfilled order, it returns a status of 1. If the customer does not have unfilled orders, it returns a status of 0. This is an example of a nested stored procedure.

1. Using SQL Query Analyzer, open C:\Moc\2073A\Labfiles\L09\Return1.sql, review its contents, and then execute it.
2. Type and execute the following procedure:

   ```
   EXEC OrderCount 1,1
   ```

   What is the result?

   This command completes successfully but does not return data.

   ____

   ____

► To execute the OrderCount stored procedure with the OUTPUT option

In this procedure, you will observe the effects of using the OUTPUT option in the **OrderCount** stored procedure.

1. Using SQL Query Analyzer, open C:\Moc\2073A\Labfiles\L09\Return2.sql, review its contents, and then execute it.

   This script executes the **OrderCount** stored procedure and passes a value of a CustomerID that has unfilled orders.

   What is the result?

   Customer RATTC has 18 unfilled order(s).

   ____

   ____

2. Open C:\Moc\2073A\Labfiles\L09\Return3.sql, review its contents, and then execute it.

   This script executes the **OrderCount** stored procedure and passes a value of a CustomerID that has unfilled orders.

   What is the result?

   Customer WOLZA has NO unfilled order(s).

   ____

   ____

If Time Permits
Executing Extended Stored Procedures

In this exercise, you will execute an extended stored procedure and view the DLL file name in which the function is defined.

► **To execute an extended stored procedure**

In this procedure, you will execute the **xp_cmdshell** extended stored procedure to list all of the files and folders in the root of drive C.

1. Using SQL Query Analyzer, verify that you are using the **master** database.
2. Execute the **xp_cmdshell** extended stored procedure to view the list of all files in the C:\ folder.

   ```
   EXEC master..xp_cmdshell 'dir c:\'
   ```

   What was the result?

   The directory listing was returned as rows of text.

   ____

   ____

3. Execute the **sp_helptext** system stored procedure to view the definition for **xp_cmdshell**.

   ```
   EXEC master..sp_helptext xp_cmdshell
   ```

   What was the result?

   Xplog70.dll. This is the DLL that contains the extended stored procedure function.

   ____

   ____

If Time Permits
Tracing Stored Procedures Using SQL Profiler

In this exercise, you will use the SQL Profiler graphical tool to trace individual stored procedures.

► To trace stored procedure events by using SQL Profiler

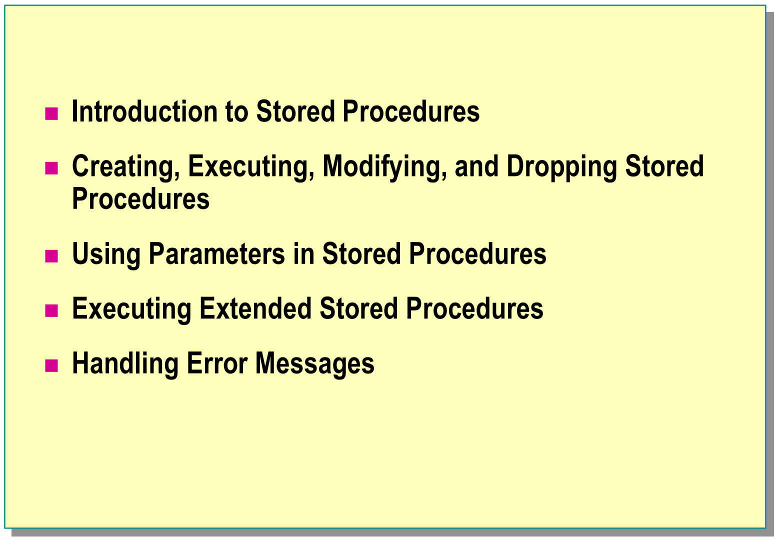
In this procedure, you will start a SQL Profiler trace by using a custom trace template to monitor stored procedures.

1. Open SQL Profiler.
2. On the toolbar, click **New Trace**.
3. Connect to the (local) server with Windows Authentication.
4. On the **Events** tab, add all stored procedures and Transact-SQL event classes.
5. Click **Run**.
6. Switch to SQL Query Analyzer, open C:\Moc\2073A\Labfiles\L09\SupplierProductInsert.sql, review its contents, and then execute it.
7. Switch to SQL Profiler.
8. Stop and then review the trace.

Review

Topic Objective
To reinforce module objectives by reviewing key points.

Lead-in
The review questions cover some of the key concepts taught in the module.



- Introduction to Stored Procedures
- Creating, Executing, Modifying, and Dropping Stored Procedures
- Using Parameters in Stored Procedures
- Executing Extended Stored Procedures
- Handling Error Messages

Use these questions to review module topics.

Ask students whether they have any questions before continuing.

1. You have created a stored procedure to remove a customer from your database. You would like to have a custom error message written to the Windows 2000 application log when the delete transaction completes. How would you perform this task?

   Create a custom error message by specifying the @with_log parameter in the sp_addmessage stored procedure. Issue the RAISERROR statement in your stored procedure to raise your custom error message when the delete transaction has been committed.

2. You want users in the payroll department to be able to insert, update, and delete data in the **payroll** database. However, you do not want them to have access to the underlying tables. How would you accomplish this goal, besides creating a view?

   Create stored procedures that accomplish each specific task. Grant EXECUTE permission to the payroll department users on the stored procedures.

3. You must modify a stored procedure in your database. Several users have been granted permission to execute this stored procedure. What statement would you execute to perform the modification without affecting the existing permissions?

   ALTER PROC. If you execute the DROP PROC and CREATE PROC statements with the desired modifications, you must grant EXECUTE permission to the users once again.