SQL Programming: Level 1

Handouts by Sandy Lux

Additional examples by Andrew Schwartz

Rev. B

University of Missouri–St. Louis
Division of Continuing Education

Computer Education & Training Center Classes

We can design specialized classes or provide group training to meet your needs.

For more information, call (314) 984-9000
http://www.cetc.umsl.edu

Table of Contents

What You Will Learn

Now that you're an experienced SQL user, you're ready to expand your SQL skills to design sophisticated database applications that combine SQL commands for retrieving and manipulating data with logical programming structures to control your data processing. While SQL is an indispensable tool for retrieving and manipulating data, it is a nonprocedural language, so it lacks the programming features that allow you to control the flow of statements using conditional processing and loops.

This course introduces you to **Transact-SQL**, Microsoft SQL Server's proprietary extension to the SQL language. Transact-SQL extends the capabilities of SQL by adding built-in functions and other statements not available in standard SQL. (A few of these statements and functions were introduced in the prerequisite SQL language courses.) Transact-SQL also includes tools and benefits of a procedural programming language, such as variables, conditional statements, loops, and error-handling capabilities. The programming constructs in Transact-SQL integrate easily with the SQL commands you learned about in the prerequisite SQL language courses, allowing you to write code that interacts with and manipulates your SQL Server data.

Using Transact-SQL, you can create **stored procedures**, which are reusable sets of SQL statements that are stored with your database. Stored procedures offer an efficient way to handle repetitive tasks, since you can call the procedure by name to execute a batch of statements quickly. As you will see, you can create stored procedures that include standard SQL statements, as well as Transact-SQL programming structures, like IF statements and loops. Creating stored procedures is a primary focus of this course. Specifically, by the end of this course, you will be able to:

- Identify the benefits of writing stored procedures.
- Create stored procedures to select and manipulate data in a SQL Server database.
- Define and reference local variables and parameters in your stored procedures.
- Process data conditionally, using IF statements and the CASE function.
- Use WHILE loops to control data processing iteratively.
- Declare and manage cursors to process individual rows returned by a multiple-row query.
- Use Transact-SQL's exception-handling capabilities, such as the TRY…CATCH construct and RAISERROR command, to raise, trap, and handle errors in your stored procedures.

Through hands-on examples, you will gain practical experience with Transact-SQL programming statements and techniques, using prepared SQL Server data. Each instructor-led, hands-on lesson culminates in an exercise that allows you to practice what you've learned and test your knowledge independently.

Writing Stored Procedures

A stored procedure is simply a small program or set of SQL statements that you can execute at one time. Stored procedures are kept on the server, in the database, rather than on the client computer. You can run **system procedures** that are built in to SQL Server, or you can create your own **user-defined stored procedures**.

Stored procedures offer several benefits over executing batches of SQL statements, including reusability and efficiency. Stored procedures are **pre-compiled**, so they do not normally have to be recompiled each time they are called. It is generally less taxing on your system to process the statements in a stored procedure, rather than repeatedly executing the same batch of statements outside of a stored procedure. In this lesson, you will learn about these and other benefits of stored procedures. You will also learn the syntax for creating and executing stored procedures.

Understanding System Procedures

Some stored procedures are supplied with Microsoft SQL Server. They are called **system procedures**, but they are nothing more than stored procedures that Microsoft has written. Other platforms also supply stored procedures with the software to improve its usability.

The Microsoft system procedures begin with the characters **sp_**. You may have used some of these system procedures in the prerequisite SQL language classes.

For example, the **sp_help** system procedure provides help on any object in the database. The following statement uses this stored procedure to return information about the Employee table:

```
sp_help employee
```

Another helpful system procedure is the **sp_rename** procedure, which allows you to rename an object, such as a table or column. This statement will change the name of the Hourrate column in the Employee table to HourlyPay:

```
sp_rename 'Employee.Hourrate', 'Employee.HourlyPay'
```

Executing Stored Procedures

You can use the **EXECUTE** statement to execute a system or user-defined stored procedure in SQL Server. Type the word EXECUTE (or **EXEC**), followed by the procedure name and any required arguments, such as:

```
EXEC sp_help employee
```

Note: The EXECUTE (or EXEC) statement isn't required if the statement is the first one in a batch (or the only statement you are running, outside of a batch). So, if the sp_help procedure above is the first or only stored procedure you want to execute, you can simply type:

```
sp_help employee
```

Also note that quotation marks surrounding the arguments to a procedure are optional unless the argument contains embedded spaces.

Creating User-Defined Stored Procedures

In addition to running system procedures, you can create your own user-defined stored procedures. This allows you—and other SQL Server users—to easily and efficiently execute a batch of statements by simply calling the procedure. As mentioned earlier, this increases **execution speed**, since the procedure does not need to be recompiled each time it is executed. Additionally, stored procedures are **easy for other users to execute**—without requiring them to re-create any complex logic behind a batch of statements. This also increases **consistency** and **accuracy**, by eliminating mistakes or other inconsistencies that could occur each time you re-create the batch of statements.

Additionally, you can control **security** by setting permissions for individual procedures, giving other users the ability to execute your procedures. Stored procedures allow you to set permissions independently of the objects that are referenced in the stored procedure. For example, you can grant a user the ability to modify an object through a stored procedure—even though that user wouldn't normally have permissions to modify that object. This allows you to grant permissions in a very limited and specific sense.

In this course, you will practice writing stored procedures using Transact-SQL, but beginning with SQL Server 2005, Microsoft also allows you to create stored procedures using Visual Basic 2005 or Visual C# 2005. Using any of these languages, you can **incorporate programming logic**—such as conditional statements and loops—into your stored procedures to control the way your statements execute. You can also use **exception handling** statements, such as Transact-SQL's TRY and CATCH blocks, to trap and handle errors.

Finally, stored procedures are **modular**, so you can call one stored procedure from another stored procedure. This allows you to create many simple stored procedures, each responsible for a very specific task, and then combine them in different ways to meet your needs.

To create a stored procedure, use the **CREATE PROCEDURE** command. The simple syntax of this statement is:

```
CREATE PROCEDURE procedure_name
AS
   sql statements
```

Stored procedure names must follow the same rules as other SQL objects, such as tables and columns. The procedure name cannot exceed 128 characters, it must be unique within the schema, and it cannot be a SQL or Transact-SQL reserved word.

The first character of the procedure name should be a Unicode letter or underscore (_), and subsequent characters can be a combination of Unicode letters, numbers, and certain symbols (specifically, @, $, #, or _). Microsoft also recommends that you don't begin your procedure names with the characters sp_, since these characters are used to identify system procedures. Additionally, the stored procedure name cannot include embedded spaces.

Note: Technically, object names in SQL Server can also begin with an at sign (@) or number sign (#), but these symbols are reserved for the names of special objects. Specifically, the at sign denotes a local variable or parameter, and the number sign denotes a temporary table or procedure.

The following command creates a simple stored procedure named **Emplist**, which selects and displays all of the records in the Employee table.

```
CREATE PROCEDURE emplist
AS
   SELECT * FROM employee
```

Executing a User-Defined Stored Procedure

You can execute a user-defined stored procedure in the same way you execute system procedures, using the **EXECUTE** (or **EXEC**) command. For example:

```
EXEC emplist
```

Note: If you are executing the stored procedure as the only statement or the first statement in a batch, you can omit the EXEC keyword and simply type the procedure name:

```
emplist
```

Displaying the Contents of a Stored Procedure

You can use the **sp_helptext** system procedure to display the contents of a stored procedure.

```
sp_helptext procedure_name
```

To display the contents of the Emplist stored procedure, run this statement:

```
EXEC sp_helptext emplist
```

SQL Server returns this information:

```
CREATE PROCEDURE emplist
AS
SELECT * FROM employee
```

Modifying a Stored Procedure

You can use the **ALTER PROCEDURE** statement to modify an existing procedure. The ALTER PROCEDURE command follows the same basic syntax as CREATE PROCEDURE:

```
ALTER PROCEDURE procedure_name
AS
   sql statements
```

For example, the following statement will modify the Emplist procedure to display the employee results in order by employee ID:

```
ALTER PROCEDURE emplist
AS
   SELECT * FROM employee ORDER BY empid
```

Note: The real advantage of using ALTER PROCEDURE instead of dropping and re-creating the stored procedure is that it allows you to keep all the permissions associated with the procedure. If you drop the procedure and then re-create it, you will have to reassign all of the permissions.

Deleting a Stored Procedure

To delete a stored procedure, use the **DROP PROCEDURE** command:

```
DROP PROCEDURE procedure_name
```

The following statement will drop the Emplist stored procedure:

```
DROP PROCEDURE emplist
```

Defining and Using Variables

Variables are temporary storage areas for data in your stored procedures. In class, you will practice defining **local variables**, which can only be referenced in the procedure where they are declared. Once you declare a local variable, you can use it throughout the procedure to process data without repeatedly accessing the database.

In Transact-SQL, you can create and use two types of variables: **declared variables** and **parameter variables**. The primary differences between the two types of variables are the way in which they are assigned values and the way in which they are defined.

In this lesson, you will learn how to define declared variables and parameters of various data types, assign values to your variables, and reference the variables in your stored procedures.

Using Declared Variables in Your Stored Procedures

A **declared variable** is assigned a value during the execution of the stored procedure; its value is not passed as an argument on the execution line. Through the following examples, you will learn how to declare a local variable, assign a value to the variable, and return the variable's value to the client (for example, to display the value in the Messages section of SQL Server Management Studio).

Declaring a Local Variable

In Transact-SQL, you use the **DECLARE** statement to define a variable within a batch of SQL statements. The syntax is shown below:

```
DECLARE @variable_name variable_data_type
```

In a CREATE PROCEDURE (or ALTER PROCEDURE) statement, local variables are declared after the AS keyword.

Note that local variables are identified by an at sign (**@**) at the beginning of the variable name. Additionally, variable names must follow the same rules as other identifiers, such as tables, columns, and stored procedures. (These rules were outlined in the previous lesson.)

Data types for variables are the same as data types for columns in SQL Server—for example, INT, NUMERIC, CHAR, DATETIME, and so on. You can also define your own types using Transact-SQL or the Microsoft .NET Framework, although this is beyond the scope of this class.

For example, the following statement declares a variable named **@msg** with the **CHAR** data type (allowing 12 characters):

```
DECLARE @msg char(12)
```

Assigning a Value to the Variable

You can use the **SET** or **SELECT** statement, along with the equal (=) operator to assign a value to a variable. This statement assigns the message "Hello world!" to the @msg variable:

```
SET @msg = 'Hello world!'
```

The following statement uses the SELECT command to accomplish the same task:

```
SELECT @msg = 'Hello world!'
```

Note: There are a few restrictions on using SELECT to assign values to a variable—specifically: a SELECT statement that contains a variable assignment cannot also be used to perform typical data retrieval operations. For example, the following would NOT work:

```
DECLARE @testvar char(1)
SELECT @testvar = 'A', empid
FROM employee
```

This statement would fail because the SELECT statement is used to assign a value to the @testvar variable and to select data from the Empid column of the Employee table.

Displaying the Variable's Value

You can use the **PRINT** statement in Transact-SQL to return a user-defined message to the client. For example, in a stored procedure, you can use PRINT to display the value of a variable in SQL Server Management Studio, such as:

```
PRINT @msg
```

Putting this together, you can write a stored procedure (named **mymessage**) that declares the variable, assigns it a value, and then returns its value to the client:

```
CREATE PROCEDURE mymessage
AS
   DECLARE @msg char(12)
   SELECT @msg = 'Hello world!'
   PRINT @msg
```

To execute this procedure:

```
EXEC mymessage
```

Note: You can also use PRINT to return a string expression—or concatenate a string expression to a variable and then return the entire message. For example, the first statement below prints a string expression (without using a variable); the second statement concatenates the string "Today's message is" with the value of the @msg variable and displays the entire message on the screen:

```
PRINT 'Do I have your attention?'
PRINT 'Today''s message is: ' + @msg
```

The following procedure demonstrates this technique:

```
ALTER PROCEDURE mymessage
AS
   DECLARE @msg char(12)
   SELECT @msg = 'Hello world!'
   PRINT 'Do I have your attention?'
   PRINT 'Today''s message is: ' + @msg
```

Declaring Multiple Variables

To declare multiple variables using one DECLARE statement, place a comma between each variable, as in the following:

```
DECLARE @numerator int, @denominator int,
        @answer int, @msg varchar(20)
```

This statement declares four variables: three with the INT data type and one with the VARCHAR data type.

The **divide** procedure declares these variables and assigns integers to the **@numerator** and **@denominator** variables. Also note that you can manipulate variables using mathematical operators, as well as Transact-SQL (or user-defined) functions.

For example, the procedure assigns a value to the **@answer** variable by dividing the **@numerator** variable by the **@denominator** variable. It also uses the **CONVERT** function to convert the answer from an integer value to a character string before concatenating it to another string and assigning the result to the **@msg** variable.

The entire **divide** procedure is printed below:

```
CREATE PROCEDURE divide
AS
   DECLARE @numerator int, @denominator int,
           @answer int, @msg varchar(20)
   SET @numerator = 12
   SET @denominator = 3
   SET @answer = @numerator/@denominator
   SET @msg = 'The answer is: ' + CONVERT(char(3),@answer)
   PRINT @msg
```

Note: You must use one SET statement per variable. You cannot assign values to multiple variables using a single SET statement. However, if you are using SELECT to assign values to variables, you can assign values to multiple variables using one statement, such as:

```
ALTER PROCEDURE divide
AS
   DECLARE @numerator int, @denominator int,
           @answer int, @msg varchar(20)
   SELECT @numerator = 12, @denominator = 3,
          @answer = @numerator/@denominator,
          @msg = 'The answer is: ' + CONVERT(char(3),@answer)
   PRINT @msg
```

Keep in mind, however, that this is not the preferred method in SQL Server 2005. This also makes it more tempting to try to write an illegal SELECT statement that combines variable assignment with data retrieval.

Also note: When you use the string concatenation operator (+) to concatenate two expressions, both expressions must be in the character or binary data type categories (except text, ntext, and image). If the concatenated expressions are not the exact same data type (like CHAR and VARCHAR, for example), SQL Server must be able to implicitly convert one type to the other. If you are concatenating a non-string value, you must explicitly convert the value to a string, using the CONVERT or CAST function. The CONVERT function, used in the previous example, follows this syntax:

```
CONVERT(new_data_type, value_to_be_converted)
```

When you are using CONVERT to convert a date to a string, this function accepts a third argument, which specifies the style of the converted date. You will see an example of this later in class. This syntax is also covered in *Appendix B*.

Using Parameter Variables in Your Stored Procedures

The procedures in the previous examples relied on declared variables, whose values are assigned inside the procedure. To achieve more flexibility, you can create **parameter variables** and assign values to the variables **when the procedure is called**. You can declare up to 2,100

parameter variables in a stored procedure. You can also use parameter variables and declared variables within the same procedure.

A parameter variable allows you to enter a value at the time you execute the procedure. Like declared variables, parameter variables are identified with a leading at sign (@). The rules for naming parameter variables are the same as the rules for naming declared variables.

Defining a Parameter Variable

You define a parameter variable in a stored procedure by specifying the variable name (with a preceding @) and the data type. Parameter variables are declared immediately after the procedure name in the CREATE PROCEDURE (or ALTER PROCEDURE) statement, before the AS keyword. The line in bold shows the syntax for defining parameter variables in a CREATE PROCEDURE statement.

```
CREATE PROCEDURE procedure_name
   @param_name param_data_type [ = default_value ]
AS ...
```

Optionally, you can assign a default value to the parameter, so the parameter will use the default value if no other value is assigned at the time the procedure is executed. (Default values are discussed in more detail below.)

The following procedure defines one parameter variable: **@firstname**. It is defined with the **VARCHAR** data type, allowing a maximum of 25 characters. The value for this variable is specified when the procedure is executed.

Also note that this procedure defines a declared variable (**@msg**), which is assigned the string "Hello," concatenated with the parameter variable's value. The PRINT statement then displays the entire message in SQL Server Management Studio.

```
CREATE PROCEDURE param_msg
   @firstname varchar(25)
AS
   DECLARE @msg varchar(60)
   SET @msg = 'Hello, ' + @firstname + '!'
   PRINT @msg
```

Note: You will sometimes see parentheses around the parameter declaration, such as:

```
CREATE PROCEDURE param_msg
   ( @firstname varchar(25) )
AS
...
```

Assigning the Parameter Value

To assign a value to a parameter variable, specify the value when you execute the stored procedure. You can rely on positional assignment or specify the parameter's name when you assign the value.

Positional Assignment

If you use the positional assignment method, the values to be assigned to the parameters are listed after the procedure name on the execution line, separated by commas. The first value (or **argument**) is assigned to the first parameter, the second value to the second parameter, and so forth.

The following command executes the param_msg stored procedure, passing in "Sandy" as the value for the parameter variable.

```
EXEC param_msg 'Sandy'
```

When you run this statement, the stored procedure will display the following:

```
Hello, Sandy!
```

Note: As long as your string argument does not include spaces, it's not essential to enclose the value in single quotation marks. Single quotation marks, however, are required if the value has embedded spaces.

Name Assignment

If you use the name assignment method, the parameter names and their corresponding values are specified on the execution line. Use the equal sign operator (=) to assign the value to the specified variable, such as:

```
EXEC param_msg @firstname = 'Sandy'
```

Although this method is a little more cumbersome, it gives you some additional flexibility and clarity when you have multiple parameters in one procedure. It also allows you to assign values to variables in no specific order. (With positional assignment, values are assigned to the parameters in the order in which the parameters are defined.) You can also assign values to specific parameters while relying on default values for others.

Declaring Multiple Parameter Variables

To define multiple parameter variables in one procedure, separate the parameter definitions with a comma, as shown below:

```
ALTER PROCEDURE param_msg
   @firstname varchar(25), @lastname varchar(25)
AS
   DECLARE @msg varchar(60)
   SET @msg = 'Hello, ' + @firstname + ' ' + @lastname + '!'
   PRINT @msg
```

This procedure now includes two parameters: one for first name and one for last name.

When you run a procedure with multiple parameters, you can still use positional assignment or name assignment to assign values to the parameters.

If you assign values to the variables using the **positional assignment** method, you must specify the values in the order that you declared the variables in the procedure, using a comma to separate each value. For example:

```
EXEC param_msg 'Sandy', 'Lux'
```

When you run this statement, you will see the following message:

```
Hello, Sandy Lux!
```

If you use **name assignment**, however, the order of entry is unimportant. The following statements return the same message:

```
EXEC param_msg @firstname = 'Sandy', @lastname = 'Lux'

EXEC param_msg @lastname = 'Lux', @firstname = 'Sandy'
```

Specifying Default Values

In SQL Server, if you don't specify values for all parameter variables when you execute a procedure, you will get an error message. For example, if you run the following statement:

```
EXEC param_msg 'Sandy'
```

You will get this error message:

```
Msg 201, Level 16, State 4, Procedure param_msg, Line 0
Procedure or Function 'param_msg' expects parameter '@lastname', which was
not supplied.
```

To avoid this, it may be advantageous to assign a default value to a parameter variable, allowing the user to execute the procedure either with or without supplying an argument. You can assign the default value in the variable declaration, using the equal sign operator (=).

The following ALTER PROCEDURE statement modifies the **param_msg** procedure by assigning a zero-length string (represented by two single quotation marks) as the default value for the @firstname and @lastname variables.

```
ALTER PROCEDURE param_msg
   @firstname varchar(25) = '', @lastname varchar(25) = ''
AS
   DECLARE @msg varchar(60)
   SET @msg = 'Hello, ' + @firstname + ' ' + @lastname + '!'
   PRINT @msg
```

Now, SQL Server will use the default value for any parameter values not specified in the execution statement. For example, the following statement:

```
EXEC param_msg 'Sandy'
```

will return these results:

```
Hello, Sandy !
```

Note: If you want to use the default value of the first parameter and supply a value for the second, use name assignment to assign the value to the second parameter, such as:

```
EXEC param_msg @lastname = 'Lux'
```

Also note: Instead of an empty string, you can assign NULL as the default value for a parameter variable; however, in the param_msg procedure, you may get unexpected results if you do this. This is because the string concatenation operator treats zero-length strings and NULL values differently. If you try to concatenate a NULL value onto a string, by default, the result will be NULL. To change this default behavior, you can turn the **CONCAT_NULL_YIELDS_NULL** setting off by running this statement:

```
SET CONCAT_NULL_YIELDS_NULL OFF
```

Keep in mind, however, that Microsoft warns against relying too heavily on this setting, as it may not be available in future releases of SQL Server.

To restore the default setting, execute the following statement:

```
SET CONCAT_NULL_YIELDS_NULL ON
```

You can modify the **divide** procedure you created earlier to allow users to supply values for the numerator and denominator. This makes the procedure much more flexible, so you aren't limited to performing the same calculation over and over again. The modifications below (in bold) add

parameter variables for the numerator and denominator. Notice that the @answer and @msg variables are still declared variables, with their values assigned in the procedure.

```
ALTER PROCEDURE divide
   @numerator int, @denominator int
AS
   DECLARE @answer int, @msg varchar(20)
   SET @answer = @numerator/@denominator
   SET @msg = 'The answer is: ' + CONVERT(char(3),@answer)
   PRINT @msg
```

Now, when you run the procedure, you can supply numerator and denominator values on the execution line, such as:

```
EXEC divide 10, 5
```

(Note that numbers are not enclosed in quotation marks.)

When you run this statement, you will see the following results:

```
The answer is: 2
```

Commenting Your Code

You can use comments in your stored procedures to help others (or yourself) interpret your code. You can write comments in one of two formats:

`-- comment`	Double hyphens indicate a single-line comment. You can include this comment as the only statement on a line, or you can add the comment to the end of a Transact-SQL statement. These comments are very useful in explaining a single statement or small section of code.
`/* comment */`	You can extend a comment over multiple lines by enclosing it in these symbols. A forward slash-asterisk (`/*`) marks the beginning of the comment, and an asterisk-forward slash (`*/`) marks the end of the comment. This style of commenting is good for general, introductory comments or for documenting large sections of code.

The following ALTER PROCEDURE statement includes both types of comments (in bold):

```
/* This procedure accepts user input for first and last
   names and prints a personalized welcome message */
ALTER PROCEDURE param_msg
-- get user's first name and last name as arguments
   @firstname varchar(25) = '', @lastname varchar(25) = ''
AS
   DECLARE @msg varchar(60)
   SET @msg = 'Hello, ' + @firstname + ' ' + @lastname + '!'
   PRINT @msg
```

Note also that you can also include blank lines in your procedures to make the code easier to read. These lines will be ignored when you run the procedure.

Exercise 1: Writing Simple Stored Procedures

1. Write a short SQL procedure to compute parking charges. The procedure should accept the number of hours parked, calculate the charge at $.75 per hour, and display the result.

2. Write a program that calculates the total price of a purchase, including sales tax. The procedure should accept the price of the purchase (before tax) and calculate tax based on a rate of 7.5%. Display the original purchase price, amount of tax, and total price.

3. Write a procedure to compute gas mileage. The procedure requires three arguments: the odometer reading when you started your trip, the reading when you finished the trip, and how many gallons of fuel were used. The computer should then process the information and display your miles per gallon.

4. Given the assumption that you sleep a healthy eight hours a night, calculate and display the number of hours of your life that you have spent sleeping. Supply the date of your birth and use the system date for today's date.

   Hint: The **GETDATE** function returns the system date. The **DATEDIFF** function calculates the difference between two dates, using this syntax:

   ```
   DATEDIFF(datepart, first_date, second_date)
   ```

   where *datepart* is a component of a DATETIME value, such as **hh** for hours or **yy** for years.

Extra: The Department of Transportation and the National Society of Beekeepers need your help. The "New York Special" train is leaving New York City, heading for Los Angeles at 40 miles per hour. At the same time, the "Los Angeles Express" train is leaving Los Angeles for New York at 60 miles per hour. These two trains are on the same track, 3,000 miles apart.

During the time the trains are traveling toward each other, a bee goes back and forth between the two locomotives. As soon as the bee touches the front of one engine, it immediately turns around and flies toward the other train. The bee travels at 100 miles per hour.

Your job is to write a procedure that determines the distance the bee travels.

Selecting and Manipulating Data

As you've already learned, you can integrate Transact-SQL programming statements, like variable declaration and assignment, with standard SQL commands for retrieving and manipulating data. In this lesson, you'll practice writing stored procedures that run SQL statements against your database tables. Specifically, you'll use the following commands to interact with your data:

- You'll use SQL SELECT statements to retrieve data from your SQL Server tables.
- You'll use SQL's Data Manipulation Language (DML) commands, such as INSERT, UPDATE, and DELETE, to manipulate data.

You'll also learn to assign data from your tables to local variables, and you'll practice using the @@ROWCOUNT function to return the number of rows affected by a DML command.

Selecting Data

You can use the SQL **SELECT** statement to retrieve data and assign it to variables in your stored procedures.

The following simple stored procedure accepts a member ID and then returns the data for that member from the Members table:

```
CREATE PROCEDURE meminfo
   @memid char(5)
AS
   SELECT * FROM Members
   WHERE mem_id = @memid
```

You can execute this procedure to retrieve information about a specific member—for example, the member with the ID AC135:

```
EXEC meminfo 'AC135'
```

Note: Although it's not required, it's common practice to avoid using the same name for variables and associated columns in a table (for example, memid and mem_id).

Selecting a Column Value into a Variable

You can select a column value into a local variable so you can work with table data within your stored procedures. To assign a column value to a variable, use the syntax:

```
SELECT @var_name = column
FROM table
[ WHERE condition ]
```

For example, the following statement alters the **meminfo** procedure by declaring three local variables: one for the member's first name, one for the member's last name, and one for the year the member joined the museum. When the user executes the procedure, he or she supplies a member ID; then the procedure retrieves and displays the member's name and membership year, in this format:

```
Member name: ANNE COOPER
Member since: 2004
```

The entire ALTER PROCEDURE statement is printed below. Note the **SELECT** statement, used to assign values from the Members table to the declared variables. Also note that you can use a single SELECT statement to initialize multiple variables.

```
ALTER PROCEDURE meminfo
   @memid char(5)
AS
   -- declare variables to hold member's first name, last name,
   -- and membership year
   DECLARE @fname varchar(15), @lname varchar(15), @memyear char(4)
   -- assign column values to variables
   SELECT @fname = firstname,
          @lname = lastname,
          @memyear = DATENAME(yyyy, memdate)
   FROM members
   WHERE mem_id = @memid
   -- display member name and original membership year
   PRINT 'Member name: ' + @fname + ' ' + @lname
   PRINT 'Member since: ' + @memyear
```

Note: Keep in mind the restrictions on the SELECT statement: that you cannot combine variable assignment and data retrieval in a single SELECT statement. The following would be illegal:

```
SELECT firstname,
       lastname,
       @memyear = DATENAME(yyyy, memdate)
FROM members
WHERE mem_id = @memid
```

If the SELECT statement returns no rows, the variable retains its present value. If you do not assign a default value to the variable, the variable's value is NULL.

It's also essential to note that each of the declared variables in the previous example holds a single value. Because you are using the membership ID (a primary key value) as criteria in the previous example, your results should be limited to a single record. But what happens if the SELECT statement returns multiple values? The variable is assigned the **last value** returned by the SELECT statement.

Selecting Data from Multiple Tables

Now suppose you want to include the member's dues along with his or her name and membership year. This information is stored in the Memtype table, which includes one record for each membership type. To determine the dues for a specific member, you need to create a join between the **Members** and **Memtype** tables, based on the **Memcode** field.

Run the following ALTER PROCEDURE statement (with the modified code in bold) to display the dues with the other member information:

```
ALTER PROCEDURE meminfo
   @memid char(5)
AS
   DECLARE @fname varchar(15), @lname varchar(15), @memyear char(4),
           @mdues numeric(5,2)
   SELECT @fname = firstname,
          @lname = lastname,
          @memyear = DATENAME(yyyy, memdate),
          @mdues = dues
   FROM members INNER JOIN memtype
   ON members.memcode = memtype.memcode
   WHERE mem_id = @memid
   PRINT 'Member name: ' + @fname + ' ' + @lname
   PRINT 'Member since: ' + @memyear
   PRINT 'Membership dues: $' + CONVERT(char(6), @mdues)
```

When you test the procedure for member AC135, you will see these results:

```
Member name: ANNE COOPER
Member since: 2004
Membership dues: $49.95
```

Using Group Functions in SELECT Statements

You can also use local variables to capture—and then manipulate or display—aggregate data from a SQL Server table. For example, you could create a stored procedure to retrieve the first and last membership dates from the Members table and display the dates in this format:

```
First membership: Dec 14, 2002
Most recent membership: Feb 22, 2007
```

The following procedure assigns the first and last membership dates to the @firstdate and @lastdate variables and then uses PRINT statements to return these values.

```
CREATE PROCEDURE min_max
AS
   DECLARE @firstdate smalldatetime, @lastdate smalldatetime
   SELECT @firstdate = MIN(memdate), @lastdate = MAX(memdate)
   FROM members
   PRINT 'First membership: ' + CONVERT(char(20), @firstdate, 107)
   PRINT 'Most recent membership: ' + CONVERT(char(20), @lastdate, 107)
```

Note: The third argument to the **CONVERT** function is used to a specify a style for converting a datetime value to a string. The style 107 specifies this format:

Mon dd, yyyy

such as:

Jan 01, 1999

Manipulating Data

Along with the SELECT statement, you can use SQL's DML commands to manipulate data within your stored procedures. In this lesson, you will practice using INSERT, UPDATE, and DELETE statements within your stored procedures to insert rows into a table, modify existing table data, and delete rows from a table.

Inserting Data

The **INSERT** statement in a stored procedure follows the same format as the standard SQL INSERT statement:

```
INSERT INTO table (column1[, column2, ...])
VALUES (value1[, value2, ...])
```

One option is to hard-code data into the **VALUES** clause, as shown in the following example. The following procedure inserts a new row into the Memtype table:

```
CREATE PROCEDURE hard_insert
AS
   INSERT INTO memtype (memcode, mem_descrip, dues)
   VALUES ('DN', 'DONOR', 0)
```

When you execute the stored procedure, a new row will be inserted into the Memtype table for the DONOR member type.

Alternately, you can use parameter variables to specify values for the new row when you execute the stored procedure. The following procedure creates parameter variables for the membership code (@mcode), membership description (@mdescrip), and membership dues (@mdues). Also note that it specifies default values (NULL) for the description and dues.

It then inserts a new row into the Memtype table, based on the values passed into the parameters. Finally, it uses a SELECT statement to display the new record.

```
CREATE PROCEDURE param_insert
   @mcode char(2), @mdescrip varchar(10) = NULL,
   @mdues numeric(5,2) = NULL
AS
   -- insert row into Memtype table
   INSERT INTO memtype (memcode, mem_descrip, dues)
   VALUES (@mcode, @mdescrip, @mdues)
   -- display data from new row
   SELECT * FROM memtype
   WHERE memcode = @mcode
```

To test this, you can run the following statement, which inserts a new member type for artists:

```
EXEC param_insert 'AR', 'ARTIST', 10.95
```

You can also test the default values by omitting the values for description and dues when you run the procedure:

```
EXEC param_insert 'AA'
```

Note: If you're inserting data into all the columns in the named table, it's not necessary to list each column in the **INSERT INTO** clause. However, when you're using the INSERT statement in a stored procedure, it's a good idea to list the columns into which you're inserting data. This way, if you add new columns to the table, your procedure won't fail. However, keep in mind that dropping or renaming columns can prevent the stored procedure from running.

Updating Data

You can also write stored procedures that update your table data by modifying the values in one or more columns. To update data in a stored procedure, use the standard SQL **UPDATE** command:

```
UPDATE table
SET column1 = new_value1[, column2 = new_value2, ...]
[WHERE condition]
```

The *new_value* can be a literal value, a variable, or an expression, as shown in the following example, which raises the membership dues for all membership types. The **@incr** parameter allows you to pass the percent increase (a decimal value) into the stored procedure, or you can rely on the default increase of 10%.

```
CREATE PROCEDURE update_dues
-- default 10% increase, unless you specify otherwise
   @incr numeric(5,2) = .1
AS
   UPDATE memtype
   SET dues = dues * (1 + @incr)
```

Now, to implement a 5% increase for all membership types, you can run the following statement:

```
EXEC update_dues .05
```

Deleting Rows

You can include the SQL **DELETE** statement in a stored procedure to delete rows in a SQL Server table. The syntax for the DELETE statement is:

```
DELETE FROM table
[WHERE condition]
```

If you omit the WHERE clause from the DELETE statement, the command will delete all rows in the named table.

The following is an example of a DELETE statement in a stored procedure, used to delete all members of a specific type from the Members table. You can pass in the membership type as an argument when you execute the procedure.

```
CREATE PROCEDURE delete_members
   @mcode char(2) = NULL
AS
   DELETE FROM members
   WHERE memcode = @mcode
```

To delete all senior members, you can run this procedure with SR as the argument:

```
EXEC delete_members 'SR'
```

Using the @@ROWCOUNT Function

When you issue a DML (insert, update, or delete) statement, you can use the **@@ROWCOUNT** function to determine the number of rows affected by the query.

You can modify the **delete_members** procedure from the previous example to display the number of members deleted when you run the procedure. The following ALTER PROCEDURE statement adds a PRINT statement to the end of the block. This PRINT statement uses the @@ROWCOUNT function to return the number of deleted members.

```
ALTER PROCEDURE delete_members
   @mcode char(2) = NULL
AS
   DELETE FROM members
   WHERE memcode = @mcode
   PRINT 'Number of deleted members: ' + CONVERT(char(3), @@ROWCOUNT)
```

Note: The @@ROWCOUNT function returns an integer value, which must be converted to a string before you can concatenate it onto another string.

Now, if you use this procedure to delete all child members, the procedure will return this message:

```
Number of deleted members: 3
```

Exercise 2: Selecting and Manipulating Data

In this exercise, you will use data from the JOBS and EMPLOYEE tables to practice writing stored procedures that include SELECT, INSERT, UPDATE, and DELETE statements.

Note: The column names and data types for all tables are listed in *Appendix A.*

1. Using the EMPLOYEE table, write a stored procedure that accepts an employee ID as an argument and then displays the first name, last name, and job code for the employee. The program output should appear as follows:

   ```
   Employee ID: A1465
   Name: ROBERTA PICKNEY
   Job code: SEC
   ```

   Challenge: Instead of displaying the job code (e.g., SEC), display the associated job title (e.g., SECRETARY), using data from JOBS table.

2. Create a stored procedure to insert a new job into the JOBS table. Include one parameter variable for each column in the JOBS table, so users can supply the job information when they execute the procedure. Specify default NULL values for all columns except the job code. Then execute the procedure with the following job information.

JOBCODE	**INT**
JOBDESC	**INTERN**
CLASS	**NON-EXEMPT**
MINRATE	**6.75**
MAXRATE	**9.75**

3. Write a procedure to update the maximum salaries in the JOBS table. Include two parameter variables: one to specify the percent increase for EXEMPT jobs and one to specify the percent increase for NON-EXEMPT jobs. The default increase for EXEMPT and NON-EXEMPT jobs is 0%. After the update, display all job codes and their new maximum rates.

   Run the procedure, increasing maximum rates for EXEMPT jobs by 5% and maximum rates for NON-EXEMPT jobs by 8%.

Writing Conditional IF Statements

Unlike ANSI-standard SQL, Transact-SQL includes **control structures** for controlling the flow of your executable statements. You can control the flow of execution **conditionally**, using the **IF** statement, or iteratively, using the **WHILE** loop. In this lesson, you will learn how to use various constructions of Transact-SQL's IF statement to process statements conditionally, based on criteria you define.

Writing Simple IF Statements

Computer programs almost always have to make decisions, based on certain conditions: If one condition is true, the program operates one way; if another condition is true, the program performs a different set of tasks. Many programming languages use **IF** statements to control the flow of the program, based on these conditions. Transact-SQL features an IF statement that is equivalent to the IF statement in other programming languages.

At its most basic, Transact-SQL's IF statement uses this syntax:

```
IF condition
   sql statement or block to execute if condition is true
```

The *condition*, specified after the IF keyword, must be a Boolean expression that evaluates to TRUE or FALSE.

Note: If the *condition* contains a SQL SELECT statement, the SELECT statement must be enclosed in parentheses.

If the condition is true, the statement following the IF condition will be executed. Only a single statement can be executed—unless the statement is included in a **block**. To define a block, use the **BEGIN** and **END** keywords. The following defines a block that includes three SQL statements.

```
BEGIN
   first sql statement
   second sql statement
   third sql statement
END
```

If the condition does not evaluate to TRUE, the statement (or block) following the IF condition does not execute.

Consider the following simple stored procedure, which allows the user to enter integers for two variables and then compares the values. If the value of @num1 is greater than @num2, the procedure will return a message to this effect. Finally, the stored procedure prints the message

"Thanks for playing!". Because the final PRINT statement is outside of the IF block, the message is printed regardless of the IF condition.

```
CREATE PROCEDURE simple_if
   @num1 int = 0, @num2 int = 0
AS
   IF @num1 > @num2
      PRINT 'Number1 is greater than Number2'
   PRINT 'Thanks for playing!'
```

To test the procedure, run the following statement:

```
EXEC simple_if 2, 1
```

This should return these messages:

```
Number1 is greater than Number2
Thanks for playing!
```

Executing a Block of Statements

If you want to execute more than a single SQL statement as the result of a condition, you must include the statements in a **block**, as mentioned earlier. Use the **BEGIN** keyword to signify the start of the block; the **END** keyword signifies the end of the block.

In the previous procedure, imagine you want to display a second message if @num1 is greater than @num2. To do this, include both PRINT statements in a block, as shown below:

```
ALTER PROCEDURE simple_if
   @num1 int = 0, @num2 int = 0
AS
   IF @num1 > @num2
      BEGIN
         PRINT 'Number1 is greater than Number2'
         PRINT 'You''re so smart!'
      END
   PRINT 'Thanks for playing!'
```

Note: It's acceptable to create a block with a single statement, if you think that makes your code clearer. This requires a little extra typing (since you need to supply the BEGIN and END keywords), but it may clarify which statements will run in response to a condition.

Adding an ELSE Clause

In the last example, when the value of @num1 is greater than @num2, the procedure displays a message stating this. If, however, @num1 is less than or equal to @num2, you receive only the "Thanks for playing!" message as feedback, and the procedure ends.

To execute alternate statements when a condition is not true, you can add an **ELSE** clause to your IF statement. The IF…ELSE construction uses this syntax:

```
IF condition
   sql statement or block to execute if condition is true
ELSE
   sql statement or block to execute if condition is not true
```

The statements that follow the IF condition execute only if the condition evaluates to true. If the condition is not true, the procedure goes directly to the ELSE clause and executes the statements following ELSE.

```
CREATE PROCEDURE if_else
@num1 int = 0, @num2 int = 0
AS
   IF @num1 > @num2
      BEGIN
         PRINT 'Number1 is greater than Number2'
         PRINT 'You''re so smart!'
      END
   ELSE
      BEGIN
         PRINT 'Number1 is NOT greater than Number2'
         PRINT 'Please try again.'
      END
   PRINT 'Thanks for playing!'
```

Evaluating Multiple Conditions

Transact-SQL allows you to string together multiple IF…ELSE statements to handle multiple conditions. Use the following syntax to handle multiple conditions:

```
IF condition1
   sql statement or block to execute if condition1 is true
ELSE
   IF condition2
      sql statement or block to execute if condition2 is true
   ELSE
      IF condition3
         sql statement or block to execute if condition3 is true
      ...
      ELSE
         sql statement or block to execute if no condition is true
```

If *condition1* is not true, control is passed to the ELSE clause, which evaluates a second condition (*condition2*) within another IF statement, and so on.

If none of the specified conditions is true, the statement or block following the final (optional) ELSE clause is executed.

```
CREATE PROCEDURE multiple_ifs
@num1 int = 0, @num2 int = 0
AS
   IF @num1 > @num2
      BEGIN
         PRINT 'Number1 is greater than Number2'
         PRINT 'You''re so smart!'
      END
   ELSE
      IF @num1 < @num2
      BEGIN
         PRINT 'Number1 is less than Number2'
         PRINT 'Please try again.'
      END
      ELSE
      BEGIN
         PRINT 'The numbers are equal'
         PRINT 'Are you trying to trick me?'
      END
   PRINT 'Thanks for playing!'
```

Dealing with NULL Values and Constructing Complex Conditions

When you're writing IF statements, you should be careful to consider the effect of NULL values on your conditions. This is particularly important if you use NULL as the default value for your parameter.

For example, consider what happens when you run the multiple_ifs procedure with one NULL value:

```
EXEC multiple_ifs 1, NULL
```

The procedure runs the block associated with the ELSE clause (even though the numbers are not equal).

Because the NULL variable prevents any of the conditions from evaluating to true, control passes to the final ELSE clause, which prints that the two values are equal.

To avoid this problem, be careful not to rely too heavily on the ELSE clause. Because ELSE is a catchall that runs whenever the explicitly stated conditions aren't true, it's important to consider every possible situation when the ELSE clause might run. In this example, you really want the ELSE clause to run under a specific condition: when the values of both variables are equal.

Instead of relying on the final ELSE clause to accomplish this, you can add an additional condition to deal with this possibility:

```
ALTER PROCEDURE multiple_ifs
   @num1 int = 0, @num2 int = 0
AS
   IF @num1 > @num2
      BEGIN
         PRINT 'Number1 is greater than Number2'
         PRINT 'You''re so smart!'
      END
   ELSE
      IF @num1 < @num2
         BEGIN
            PRINT 'Number1 is less than Number2'
            PRINT 'Please try again.'
         END
      ELSE
         IF @num1 = @num2
            BEGIN
               PRINT 'The numbers are equal'
               PRINT 'Keep trying...'
            END
         ELSE
            BEGIN
               PRINT 'Something else is going on.'
               PRINT 'Are you trying to trick me?'
            END
   PRINT 'Thanks for playing!'
```

While this modification prevents the procedure from displaying blatantly wrong information, it still may not handle NULL values as effectively as you'd like. To check for NULL values, you can use the **IS NULL** comparison in the condition, as you would in the WHERE clause of a SELECT statement.

```
IF @num1 IS NULL
   ...
```

In this example, you may want to use the same condition to test whether either parameter is NULL. You can combine conditions in an IF clause the same way you combine conditions in the WHERE clause of a SELECT statement: by using the logical operators AND, OR, or NOT. For example, the following condition will be true if **either** @num1 or @num2 is NULL:

```
IF @num1 IS NULL OR @num2 IS NULL
   ...
```

To add this to your procedure:

```
ALTER PROCEDURE multiple_ifs
   @num1 int = 0, @num2 int = 0
AS
   IF @num1 IS NULL OR @num2 IS NULL
      PRINT 'You must enter a non-null value for both numbers.'
   ELSE
      IF @num1 > @num2
         BEGIN
            PRINT 'Number1 is greater than Number2'
            PRINT 'You''re so smart!'
         END
      ELSE
...
```

Note: The ... at the end of the procedure indicates that there is no change to the rest of the procedure.

Using SQL Server Data to Make Decisions

Now that you're comfortable with the components of the IF statement, you can practice writing IF statements that make decisions based on your table data.

For example, you might want to provide a special "GOLD" membership and a 10% membership discount for members who joined before January 1, 2005. All other members will be considered "WHITE" members, and their membership fee will not be discounted.

You can write a procedure that evaluates a membership ID (passed in as an argument) and uses an IF statement to determine whether that member is eligible for the new membership benefits. The procedure should display the member ID, membership level, and discount, as shown below:

```
Member ID: RP090
Membership level: GOLD
Membership discount: 10.00%
```

The following procedure performs these tasks:

```
CREATE PROCEDURE mem_disc
   @memid char(5) = NULL
AS
   DECLARE @level varchar(10), @disc numeric(5,2),
           @mdate smalldatetime
   -- selects membership date based on ID entered by user
   SELECT @mdate = memdate
   FROM   members
   WHERE  mem_id = @memid
   -- determines membership level and discount based on memdate
   IF @mdate < '01-01-05'
      BEGIN
         SET @level = 'GOLD'
         SET @disc = .1
      END
   ELSE
      BEGIN
         SET @level = 'WHITE'
         SET @disc = 0
      END
   -- displays membership level and discount
   PRINT 'Member ID: ' + @memid
   PRINT 'Membership level: ' + @level
   PRINT 'Membership discount: ' + CONVERT(char(5), @disc*100) + '%'
```

Adding More Conditions

In the last example, the museum offered only two levels of membership: GOLD and WHITE. You can modify the block to deal with additional membership levels, using another IF clause for each additional level. For example, the following procedure assigns a PLATINUM level to members who joined before January 1, 2003, and gives these members a 15% discount. Members who joined between January 1, 2003, and December 31, 2004, still qualify for a 10% discount as GOLD members, and all other members have WHITE memberships.

The addition to the previous procedure is formatted in bold:

```
ALTER PROCEDURE mem_disc
   @memid char(5) = NULL
AS
   DECLARE @level varchar(10), @disc numeric(5,2),
           @mdate smalldatetime
   -- selects membership date based on ID entered by user
   SELECT @mdate = memdate
   FROM   members
   WHERE  mem_id = @memid
```

```
-- determines membership level and discount based on memdate
IF @mdate < '01-01-03'
   BEGIN
      SET @level = 'PLATINUM'
      SET @disc = .15
   END
ELSE
   IF @mdate < '01-01-05'
      BEGIN
         SET @level = 'GOLD'
         SET @disc = .1
      END
   ELSE
      BEGIN
         SET @level = 'WHITE'
         SET @disc = 0
      END
-- displays membership level and discount
PRINT 'Member ID: ' + @memid
PRINT 'Membership level: ' + @level
PRINT 'Membership discount: ' + CONVERT(char(5), @disc*100) + '%'
```

Nesting IF Statements

You can also nest additional IF statements within an IF block to test complex conditions. Building on the previous example, assume that only student members are eligible for the special GOLD and PLATINUM memberships. All other members (non-student members, as well as student members who joined on or after January 1, 2005) are WHITE members.

To deal with this condition, you must declare a variable to hold the member's membership code. Then in the SELECT statement, assign the member's code to the new variable (using the value of the Memcode column in the Members table).

Next, you want to test the variable to see if the membership code is 'ST' (for a student member). If that condition is true, a second (nested) IF statement checks the membership date to determine whether the member is eligible for a special membership. If the member is not a student member, or if the membership date does not qualify for a special membership, the member is assigned a WHITE membership.

The code in bold implements these changes. Note the BEGIN and END statements around the nested IF…ELSE statement. These are necessary if you want to associate the final ELSE clause (to set the membership level to WHITE) with the original (not nested) IF.

```
ALTER PROCEDURE mem_disc
   @memid char(5) = NULL
AS
   DECLARE @level varchar(10), @disc numeric(5,2),
          @mdate smalldatetime, @mcode char(2)
   -- selects membership date based on ID entered by user
   SELECT @mdate = memdate, @mcode = memcode
   FROM   members
   WHERE  mem_id = @memid
   -- determines membership level and discount based on memdate
   IF @mcode = 'ST'
      BEGIN
         IF @mdate < '01-01-03'
            BEGIN
               SET @level = 'PLATINUM'
               SET @disc = .15
            END
         ELSE
            IF @mdate < '01-01-05'
               BEGIN
                  SET @level = 'GOLD'
                  SET @disc = .1
               END
      END
   ELSE
      BEGIN
         SET @level = 'WHITE'
         SET @disc = 0
      END
   -- displays membership level and discount
   PRINT 'Member ID: ' + @memid
   PRINT 'Membership level: ' + @level
   PRINT 'Membership discount: ' + CONVERT(char(5), @disc*100) + '%'
   PRINT 'Membership code: ' + @mcode
```

Exercise 3: Writing IF Statements

In this exercise, you will practice writing IF statements to evaluate and process data based on conditions you specify. You will use data from the JOBS and EMPLOYEE tables to complete some exercise questions.

1. Set up a procedure that requests the user's age and then determines whether the person is old enough to drive. If the person is 16 years or older, display the following:

   ```
   You are old enough to drive a car.
   ```

   Otherwise, indicate how many years the person must wait before driving, such as:

   ```
   You are only 14
   You must wait 2 years to drive.
   ```

2. Modify the previous procedure to add a condition for 15-year-olds who qualify for a learner's permit. If the user is 15, display the following statement:

   ```
   You can get a learner's permit.
   ```

3. Write a procedure that calculates a raise percentage based on an employee's score on a performance review. The raises are awarded as follows:

Evaluation Score	Raise
3	5%
2	3%
1	0%

   Have the user input the evaluation score; then display the raise based on that score. (Note that 2 should be the default score.) If the user enters an invalid score (anything other than 1, 2, or 3), display the following message:

   ```
   Enter 1, 2, or 3 as the rating.
   ```

(continued on the following page)

Exercise 3
(continued)

Challenge: Modify the previous procedure to update a museum employee's hourly pay rate (in the EMPLOYEE table), based on his or her evaluation rating. You should accept input for the employee ID and raise percentage and update the record associated with that ID. When you're finished with the update, display the employee's hourly rate from the EMPLOYEE table.

You can test the procedure with the following employees:

Employee ID	Rating	Raise	New Hourrate
A1465	3	5%	11.18
B2559	2	3%	40.43
A5150	1	0%	11.25 (no change)
A4115	5	Invalid input.	10.05 (no change)

Extra: Write a procedure that determines overtime eligibility for museum employees. Define parameter variables to get user input for employee ID and hours worked. Then determine overtime eligibility for the employee whose ID you entered, based on the following factors:

- Only employees with jobs classified as NON-EXEMPT are eligible for overtime.
- Employees are only eligible for overtime when they work more than 40 hours in a week.

In the procedure, display a message stating whether the employee is eligible for overtime.

Hint: You will have to use a join in your SELECT statement to determine an employee's job class (from the JOBS table) based on the employee's ID (from the EMPLOYEE table).

Challenge: Now modify the previous procedure to calculate and display the employee's weekly pay, using the pay rate from the HOURRATE column in the EMPLOYEE table. Consider the following factors in calculating weekly pay:

- NON-EXEMPT employees receive their regular rate for the first 40 hours they work.
- NON-EXEMPT employees receive 1.5 times their regular pay for all hours over 40.
- EXEMPT employees are always paid for 40 hours per week at their regular rate, regardless of the number of hours worked.

Using the RETURN Statement to Exit a Procedure

You can use the **RETURN** statement in Transact-SQL to exit from a query or procedure. The following is a simple example that uses RETURN to exit a procedure if the parameter value is NULL.

```
CREATE PROCEDURE exit_if_null
   @num int = NULL
AS
   IF @num IS NULL
      BEGIN
         PRINT 'You entered a null value. Goodbye.'
         -- exits procedure if value is null
         RETURN
      END
PRINT 'Your number is...'
PRINT @num
```

If you run this with a null parameter value, the procedure will display this statement only and then quit:

```
You entered a null value. Goodbye.
```

The PRINT statements at the end of the procedure will not be executed.

Think back to the procedure you wrote for the challenge part of question 3 in the previous exercise. (This procedure updated an employee's hourly rate, based on the employee ID and a performance evaluation rating.) You could rewrite this procedure to include a RETURN statement, so the procedure exits if the user enters an invalid performance review rating.

The changes to the original solution are in bold. The lines that are no longer necessary have been commented out (and formatted in italic font in the following code).

```
ALTER PROCEDURE emp_raise
   @emp_id char(5) = NULL, @rating int = 2
AS
DECLARE @raise numeric(5,3), @newrate numeric(5,2)
IF @rating = 3
   SET @raise = .05
ELSE
   IF @rating = 2
      SET @raise = .03
   ELSE
      IF @rating = 1
         SET @raise = 0
```

```
--IF @raise IS NULL
      ELSE
         BEGIN
            PRINT 'Enter 1, 2, or 3 as the rating.'
            PRINT 'No change to employee''s hourly rate'
            -- exit procedure
            RETURN
         END
--ELSE
--BEGIN
PRINT 'Your raise is ' + CONVERT(varchar(5), (@raise * 100))
     + '%'
-- update employee's rate in Employee table
UPDATE employee
SET hourrate = hourrate * (1 + @raise)
WHERE empid = @emp_id
PRINT 'Update complete.'
--END
-- get rate from Employee table
SELECT @newrate = hourrate
FROM employee
WHERE empid = @emp_id
-- display hourly rate
PRINT 'Hourly rate is: ' + CONVERT(varchar(8), @newrate)
```

Now, when you run this procedure with an invalid rating (anything other than 1, 2, or 3), the procedure will display the following message and then exit:

```
Enter 1, 2, or 3 as the rating.
No change to employee's hourly rate
```

The PRINT and UPDATE statements near the end of the procedure will not execute.

Testing for the Existence of Data

Many of the procedures you've written so far rely on users to enter valid data from a table, such as a membership code, member ID, or employee ID. But what happens if the data is not valid?

You can use a subquery to determine whether a specific value is present in a table. The following examples demonstrate the use of the **IN** and **EXISTS** keywords in subqueries to test for the existence of data in a table.

The following addition to the emp_raise procedure tests whether the employee ID entered by the user matches an employee ID in the Employee table. If it does not, the procedure prints the message "Please enter a valid employee ID" and exits. (The rest of the procedure remains unchanged.)

```
ALTER PROCEDURE emp_raise
   @emp_id char(5) = NULL, @rating int = 2
AS
-- check if valid employee ID, exit procedure if not
   IF @emp_id NOT IN (SELECT empid FROM employee)
      BEGIN
         PRINT 'Please enter a valid employee ID.'
         RETURN
      END
   DECLARE @raise numeric(5,3), @newrate numeric(5,2)
   IF @rating = 3
      SET @raise = .05
   ...
```

Note: If a SELECT statement is included in the condition, it must be enclosed in parentheses.

You can also use the **EXISTS** keyword in a subquery to test for the existence of a particular value in a table. The following produces the same results as the procedure above.

```
ALTER PROCEDURE emp_raise
   @emp_id char(5) = NULL, @rating int = 2
AS
--check if valid employee ID, exit procedure if not
   IF NOT EXISTS (SELECT * FROM employee WHERE empid = @emp_id)
      BEGIN
         PRINT 'Please enter a valid employee ID.'
         RETURN
      END
   DECLARE @raise numeric(5,3), @newrate numeric(5,2)
   IF @rating = 3
      SET @raise = .05
   ...
```

Note: These techniques are functionally equivalent. It's up to you to decide which method to use.

Using the CASE Function for Conditional Processing

You can use Transact-SQL's **CASE** function to evaluate a list of conditions and return one of multiple possible results. Among other things, you can use the CASE function to display a derived value in a SELECT statement or assign a value to a variable in a stored procedure.

The CASE function can be written in two different formats:

- The **simple CASE function** compares an expression to a simple set of expressions to determine the result.

- The **searched CASE function** evaluates a set of Boolean conditions to determine the result.

Often, the CASE function can provide a simpler alternative to an IF statement that deals with multiple conditions and returns a simple expression.

As with most other Transact-SQL functions, you can use the CASE function within or outside of a stored procedure.

Using the Simple CASE Function

The simple CASE function follows this syntax:

```
CASE input_expression
   WHEN when_expression1 THEN result1
   WHEN when_expression2 THEN result2
   ...
   [ ELSE else_result ]
END
```

This function begins with the **CASE** keyword, followed by the expression you want to evaluate. Often this *input_expression* is the name of a column or variable. The CASE function works by comparing the *input_expression* to the *when_expression* specified in each **WHEN** clause. When a matching expression is found, the function returns the result associated with the matching expression, and control is resumed with the statement after the END keyword.

You can include an optional ELSE clause to return a value if no *when_expression* matches the *input_expression*.

For example, you could use the simple CASE function to assign a membership discount associated with a membership type, according to the following table:

Member Code	Discount
CH	25%
ST	20%
FM	10%
All others	0%

The following CASE function would evaluate the value of the Memcode column and return a discount value (.25, .2, .1, or 0) based on the membership code:

```
CASE memcode
   WHEN 'CH' THEN .25
   WHEN 'ST' THEN .2
   WHEN 'FM' THEN .1
   ELSE 0
END
```

You can include this function in the column list of a SELECT statement to display the member discount, along with the member ID and membership code for each record in the Members table:

```
SELECT mem_id, memcode,
   CASE memcode
      WHEN 'CH' THEN .25
      WHEN 'ST' THEN .2
      WHEN 'FM' THEN .1
      ELSE 0
   END
   AS mem_discount
FROM members
```

You can also incorporate the CASE function into a stored procedure. The following stored procedure accepts a membership code as an argument and then displays the membership code and associated discount, in this format:

```
Member type: CH
Discount: 25.00%
```

The following procedure accomplishes this task:

```
CREATE PROCEDURE simple_case
   @mcode char(2) = NULL
AS
   DECLARE @disc numeric(3,2)
   SET @disc =
      CASE @mcode
         WHEN 'CH' THEN .25
         WHEN 'ST' THEN .2
         WHEN 'FM' THEN .1
         ELSE 0
      END
PRINT 'Member type: ' + @mcode
PRINT 'Discount: ' + CONVERT(varchar(5), @disc * 100) + '%'
```

Using the Searched CASE Function

The searched CASE function is more flexible than the simple CASE function, as you can evaluate a number of conditional expressions to return a result. Unlike the simple CASE function, you are not restricted to comparing an input expression to a number of expressions to find an equal value.

The searched CASE function follows this syntax:

```
CASE
   WHEN condition1 THEN result1
   WHEN condition2 THEN result2
   ...
   [ ELSE else_result ]
END
```

The searched CASE function returns the result associated with the first condition that evaluates to true.

Note that you do not supply an input expression with the searched CASE function. Additionally, in the WHEN clauses, the condition can be any Boolean expression that evaluates to true or false. You can use the same comparison operators available in the WHERE clause of a SELECT statement, such as =, <, or >, to name only a few.

As with the simple CASE function, you can include an ELSE clause in the searched CASE function. The *else_result* is returned if none of the specified conditions evaluates to true. (A NULL value is returned if no condition evaluates to true and no ELSE clause is present.)

You can use the searched CASE function to assign a membership type based on a member's age, according to this table:

Age	Member code
<= 10	CH
11-21	ST
22-54	AD
55+	SR

Because this isn't a test of simple equality (each code is associated with a range of ages, not one specific age), the solution requires a searched CASE function instead of a simple CASE function.

The following procedure accepts an age as an input parameter and returns a membership code associated with the age:

```
CREATE PROCEDURE searched_case
   @age int = 0
AS
   DECLARE @mcode char(2)
   SET @mcode =
      CASE
         WHEN @age <= 10 THEN 'CH'
         WHEN @age <= 21 THEN 'ST'
         WHEN @age <= 54 THEN 'AD'
         ELSE 'SR'
      END
   PRINT 'Membership type: ' + @mcode
```

While the procedure above uses only age to determine the membership type, you may realistically need to consider other conditions, such as the number of family members who will be using the membership. If the membership will be used by multiple members of the same family, you should assign the family membership type (FM), regardless of the member's age. You can incorporate this condition into your searched CASE function, as shown below:

```
ALTER PROCEDURE searched_case
   @age int = 0, @famnum int = 1
AS
   DECLARE @mcode char(2)
   SET @mcode =
      CASE
         WHEN @famnum > 1 THEN 'FM'
         WHEN @age <= 10 THEN 'CH'
         WHEN @age <= 21 THEN 'ST'
         WHEN @age <= 54 THEN 'AD'
         ELSE 'SR'
      END
   PRINT 'Membership type: ' + @mcode
```

It's important to note the order of conditions in this CASE function. Since the number of family members supersedes the age-related conditions, it should be first in the list of conditions to test.

Using the CASE Function with SQL Server Data

You can use the CASE function with your museum data to determine membership discounts or membership codes, based on data from individual members. You could easily use the results from your CASE functions to update information in your Members table.

The following procedure takes user-supplied data (a membership ID, the member's age, and the number of family members who will use the museum membership) and uses this input to update the membership code in the Members table. The added code is in bold:

```
ALTER PROCEDURE searched_case
   @memid char(5) = NULL, @age int = 0, @famnum int = 1
AS
   DECLARE @mcode char(2)
   SET @mcode =
      CASE
         WHEN @famnum > 1 THEN 'FM'
         WHEN @age <= 10 THEN 'CH'
         WHEN @age <= 21 THEN 'ST'
         WHEN @age <= 54 THEN 'AD'
         ELSE 'SR'
      END
-- update members table with new code
   UPDATE members
   SET memcode = @mcode
   WHERE mem_id = @memid
   PRINT 'Member ID: ' + @memid
   PRINT 'Membership type changed to: ' + @mcode
```

Although this procedure successfully assigns the correct membership code to each member, the output may be a little misleading: the final statement prints "Membership type changed to…" even if the procedure reassigns the member's existing code.

To avoid this ambiguity, you could add a conditional statement to the procedure to compare the current membership code to the new membership code and only run the update if the codes are different. This solution requires an additional variable to hold the current membership code—named **@oldcode** in the following example. It also includes an IF statement to compare the new code (@mcode) to the existing code (@oldcode) and only runs the update if they are not equal.

The changes to the existing procedure are formatted in bold:

```
ALTER PROCEDURE searched_case
   @memid char(5) = NULL, @age int = 0, @famnum int = 1
AS
   DECLARE @mcode char(2), @oldcode char(2)
   SET @mcode =
      CASE
         WHEN @famnum > 1 THEN 'FM'
         WHEN @age <= 10 THEN 'CH'
         WHEN @age <= 21 THEN 'ST'
         WHEN @age <= 54 THEN 'AD'
         ELSE 'SR'
      END
-- get existing code from Members table
   SELECT @oldcode = memcode
   FROM members
   WHERE mem_id = @memid
-- if old code and new code are different,
-- update members table with new code
   IF @oldcode <> @mcode
      BEGIN
         UPDATE members
         SET memcode = @mcode
         WHERE mem_id = @memid
         PRINT 'Member ID: ' + @memid
         PRINT 'Membership type changed to: ' + @mcode
      END
   ELSE
      BEGIN
         PRINT 'No change to current code.'
         PRINT 'Membership code is: ' + @oldcode
      END
```

Exercise 4: Using the CASE Function

In this exercise, you will practice using simple and searched CASE functions to process data in your stored procedures. This exercise uses data from the EMPLOYEE table.

1. Rewrite the procedure in Exercise 3, question 3, using a simple CASE function. This procedure asks the user to input an evaluation score and then displays the employee's raise based on that score:

Evaluation Score	Raise
3	5%
2	3%
1	0%

   If the user enters an invalid score (anything other than 1, 2, or 3), display the following message instead of the employee's raise:

```
Enter 1, 2, or 3 as the rating.
```

   Hint: You can use a combination of the CASE function and IF statement to deal effectively with invalid inputs.

2. Write a stored procedure that uses the CASE function to assign a salary grade based on an employee's hourly wage:

Hourly Rate	Grade
$9.99 or less	A
between $10.00 and $19.99	B
between $20.00 and $29.99	C
$30.00 or more	D

   A user will input an hourly rate, and the program should return a grade based on this rate, such as:

```
Hourly rate: $11.95
Salary grade: B
```

Challenge: Modify the previous procedure to have the user input an employee ID (from the EMPLOYEE table) and calculate the salary grade based on that employee's hourly rate. Consider adding a conditional statement to validate the employee ID entered by the user and exit the procedure if the user provides an invalid ID.

Working with Loops

While IF statements give you conditional control over your procedures, loops offer **iterative control**, allowing you to execute the same code repeatedly. In this lesson, you will learn how to use Transact-SQL's **WHILE loop** to execute blocks of SQL statements repeatedly in your stored procedures.

Writing WHILE Loops

Transact-SQL offers the WHILE loop for iterative processing. A simple WHILE statement follows this syntax:

```
WHILE condition
   sql statement or block
```

The statement(s) are executed repeatedly as long as the condition evaluates to true.

The condition is any Boolean expression that evaluates to true or false. If the expression contains a SELECT statement, the SELECT statement must be enclosed in parentheses.

Note: As with the IF statement, if you want to execute more than one statement in a WHILE loop, you must include the statements in a block, designated with the BEGIN and END keywords.

Consider the following example, which calculates your new credit card balance after each payment (assuming no interest). The procedure accepts an argument for the initial balance, and the payment is set to a fixed amount ($300). The statements in the loop subtract the payment from the remaining balance, reset the balance variable, and display the new balance.

```
CREATE PROCEDURE credit_loop
   @bal numeric(7,2) = 0
AS
   DECLARE @pmt numeric(7,2)
   SET @pmt = 300
   WHILE @bal > 0
      BEGIN
         SET @bal = @bal - @pmt
         PRINT 'New balance: $' + CONVERT(char(10), @bal)
      END
```

When you test the procedure with an initial balance of $1,500, it will generate the following output:

```
New balance: $1200.00
New balance: $900.00
New balance: $600.00
New balance: $300.00
New balance: $0.00
```

Testing the Loop Condition

If you tested the procedure with an initial balance of $1500, the loop ends when the credit card balance equals 0. Because the payment amount was evenly divisible by the total balance, your procedure worked as you expected. However, this may not always be the case. What happens if your initial balance isn't evenly divisible by the payment amount? Or what if the initial balance is less than the regular payment? You will be left with a negative balance.

For example, when you test with an initial balance of $1,600, the output is:

```
New balance: $1300.00
New balance: $1000.00
New balance: $700.00
New balance: $400.00
New balance: $100.00
New balance: $-200.00
```

To solve this problem, you could rewrite the procedure to refigure your final payment amount when the remaining balance is less than the regular payment. This first involves changing the loop condition—so the statements in the loop only execute when the balance is greater than or equal to the payment amount. You can also include an IF statement to test whether the remaining balance is greater than 0, and if it is, print a message that the final payment is equal to the remaining balance.

```
ALTER PROCEDURE credit_loop
   @bal numeric(7,2) = 0
AS
   DECLARE @pmt numeric(7,2)
   SET @pmt = 300
   WHILE @bal >= @pmt
      BEGIN
         SET @bal = @bal - @pmt
         PRINT 'New balance: $' + CONVERT(char(10), @bal)
      END
   IF @bal > 0
      PRINT 'Final payment: $ ' + CONVERT(char(10), @bal)
```

Using a Counter

Loops frequently rely on **counters** to keep track of or test the number of iterations. In the credit card example, you may want to use a counter to calculate the number of payments you need to make to pay off the entire balance.

To use a counter, you must first declare a counter variable (like **@num_pmt** in the following example). You should also initialize the counter; in this case, you'll want the counter to begin at 0 and increment by one each time the loop runs.

To increment the counter by one for each iteration of the loop, use the following statement inside the loop:

```
SET @num_pmt = @num_pmt + 1
```

The following code shows how this counter is implemented in the credit loop procedure. In addition to the statements declaring and incrementing the counter, the PRINT statement has been changed to display the payment number, along with the current balance, and an additional PRINT statement prints the total number of payments (after the loop ends).

The changes to the procedure are formatted in bold:

```
ALTER PROCEDURE credit_loop
   @bal numeric(7,2) = 0
AS
   DECLARE @pmt numeric(7,2), @num_pmt int
   SET @pmt = 300
   SET @num_pmt = 0
   WHILE @bal >= @pmt
      BEGIN
         SET @bal = @bal - @pmt
         SET @num_pmt = @num_pmt + 1
         PRINT 'New balance after payment #' +
               CONVERT(char(5), @num_pmt) +
               '$' + CONVERT(char(10), @bal)
      END
   PRINT '' --print blank line
   PRINT 'Total number of regular payments until payoff: ' +
         CONVERT(char(5), @num_pmt)
   IF @bal > 0
      PRINT 'Plus one final payment of $ ' + CONVERT(char(10), @bal)
```

Using a Counter in the Loop Condition

You can also use a counter variable in the loop condition to run the loop a specific number of times. During each iteration, increment the counter, and when the counter reaches the maximum number of iterations, the loop will end.

Assume you want to keep track of your savings for a year. Write a procedure that allows you to enter a monthly savings amount, and the procedure will calculate interest (10% each month) and display your savings total after each month. **Hint:** 10% interest per year calculates to .83% interest per month.

```
CREATE PROCEDURE savings_loop
   @monthly_savings numeric(7,2) = 100
AS
   DECLARE @total_savings numeric(7,2),
           @interest numeric(4,4), @num_months int
   SET @total_savings = 0
   SET @interest = .0083
   SET @num_months = 0
   WHILE @num_months < 12
      BEGIN
         SET @total_savings = @total_savings + @monthly_savings
         SET @total_savings = @total_savings +
               (@total_savings * @interest)
         SET @num_months = @num_months + 1
         PRINT 'Savings after ' + CONVERT(char(2), @num_months)
               + ' months: ' + CONVERT(varchar(10), @total_savings)
      END
   PRINT ''
   PRINT 'Final savings: ' + CONVERT(varchar(10), @total_savings)
```

When you test this procedure with $100 monthly savings, you will see these results:

```
Savings after 1  months: 100.83
Savings after 2  months: 202.50
Savings after 3  months: 305.01
Savings after 4  months: 408.37
Savings after 5  months: 512.59
Savings after 6  months: 617.67
Savings after 7  months: 723.63
Savings after 8  months: 830.47
Savings after 9  months: 938.19
Savings after 10 months: 1046.81
Savings after 11 months: 1156.33
Savings after 12 months: 1266.76

Final savings: 1266.76
```

Breaking Out of a Loop

You can use Transact-SQL's **BREAK** statement to break out of a loop. Generally, the BREAK statement is initiated by an IF test, so if some condition is true, the loop ends (even if the loop condition is still true). After a BREAK, control resumes with the first statement following the loop.

Note: The BREAK statement is similar to the RETURN statement, except it is used to exit out of a loop—not the entire procedure.

In the previous example, you may want to break out of the loop as soon as you reach a specific savings goal—for example, after you've accumulated $1,000. The code in bold enforces this condition.

```
ALTER PROCEDURE savings_loop
   @monthly_savings numeric(7,2) = 100
AS
   DECLARE @total_savings numeric(7,2),
           @interest numeric(4,4), @num_months int
   SET @total_savings = 0
   SET @interest = .0083
   SET @num_months = 0
   WHILE @num_months < 12
      BEGIN
         SET @total_savings = @total_savings + @monthly_savings
         SET @total_savings = @total_savings +
               (@total_savings * @interest)
         SET @num_months = @num_months + 1
         PRINT 'Savings after ' + CONVERT(char(2), @num_months)
               + ' months: ' + CONVERT(varchar(10), @total_savings)
         IF @total_savings >= 1000
            BEGIN
               PRINT 'You reached your goal!'
               PRINT ''
               BREAK
            END
      END
   PRINT ''
   PRINT 'Final savings: ' + CONVERT(varchar(10), @total_savings)
```

Now, when you test the procedure with $100 monthly savings, the loop will exit after it runs seven times:

```
Savings after 1  months: 100.83
Savings after 2  months: 202.50
Savings after 3  months: 305.01
Savings after 4  months: 408.37
Savings after 5  months: 512.59
Savings after 6  months: 617.67
Savings after 7  months: 723.63
Savings after 8  months: 830.47
Savings after 9  months: 938.19
Savings after 10 months: 1046.81
You reached your goal!


Final savings: 1046.81
```

Using CONTINUE to Continue a Loop

You can also use the **CONTINUE** statement to control processing within a loop. The CONTINUE statement is used to break out of the current loop iteration and then continue execution of the loop from the beginning.

Building on the previous example, you may want to know when you reach your savings goal ($1,000), but you also want to find out how much money you would have if you keep saving for the entire 12 months.

If you replace the BREAK statement with a CONTINUE statement, the procedure will continue execution of the loop from the beginning as soon as it encounters the CONTINUE. The rest of the statements in the loop (in this case, the "Keep saving…" message) will not be executed.

```
ALTER PROCEDURE savings_loop
   @monthly_savings numeric(7,2) = 100
AS
   DECLARE @total_savings numeric(7,2),
           @interest numeric(4,4), @num_months int
   SET @total_savings = 0
   SET @interest = .0083
   SET @num_months = 0
   WHILE @num_months < 12
      BEGIN
         SET @total_savings = @total_savings + @monthly_savings
         SET @total_savings = @total_savings +
               (@total_savings * @interest)
         SET @num_months = @num_months + 1
         PRINT 'Savings after ' + CONVERT(char(2), @num_months)
               + ' months: ' + CONVERT(varchar(10), @total_savings)
         IF @total_savings >= 1000
            BEGIN
               PRINT 'You reached your goal!'
               PRINT ''
               CONTINUE
            END
         PRINT 'Keep saving...'
         PRINT ''
      END
   PRINT 'Final savings: ' + CONVERT(varchar(10), @total_savings)
```

Now, when you test with $100 monthly savings, the results will show "Keep saving…" for months 1 through 9 and "You reached your goal!" for months 10 through 12.

Creating an Infinite Loop

With Transact-SQL, you can create an infinite (or "endless") loop by specifying a loop condition that always evaluates to true (for example, 1=1). An infinite loop will never end, unless you include a BREAK statement that breaks out of the loop as a result of a condition. The benefit of

an infinite loop is that you can program it to always execute at least once (as long as the BREAK statement isn't the first statement in the loop).

The following is an example of a procedure that contains an infinite loop:

```
CREATE PROCEDURE infinite_loop
   @num int = 0
AS
   -- condition is always true
   WHILE 1=1
      BEGIN
         PRINT 'Your number is: ' + CONVERT(char(3), @num)
         -- break out of loop if number is 5 and today is Monday
         IF @num = 5 AND DATENAME(DW,GETDATE()) = 'MONDAY'
            BREAK
         -- break out of loop if number is greater than 10
         ELSE
            IF @num > 10
               BREAK
         SET @num = @num + 1
      END
```

This procedure accepts an integer argument and prints the value as the first statement in the loop. The loop can end as a result of one of two conditions: If the value is 5 and the current day is Monday, or if the value is greater than 10. The final statement in the loop increments the variable value.

One benefit of creating an infinite loop—rather than including the BREAK condition in the WHILE statement—is that the loop is guaranteed to execute at least once. In the previous example, even if the initial value is greater than 10, the loop will execute once (printing the value) before it ends.

Using Loops to Process SQL Server Data

Loops can be very useful in processing your SQL Server data. The previous examples included loops that didn't interact with the tables in your database, but now that you understand the loop syntax, it's easy to incorporate loops into procedures intended to manipulate your SQL Server data. The example in this section demonstrates how to use a loop to cycle through data in a table and re-evaluate an aggregate value after each loop. In subsequent lessons, you will practice using WHILE loops with cursors to process groups of records.

For this example, assume that you want to create a report that shows the number of new members for each year, along with a running total of the number of museum members (incremented for each year).

The following procedure declares three integer variables: one to hold the year, one to track a running total for the number of members, and one to hold the number of new members for each year. The initial value of the @memyear variable is equal to the first membership year.

The first statement in the WHILE loop sets the @new_mems variable to the number of memberships in the current year. The @total_mems variable is then updated to include all existing members, along with the new memberships. The PRINT statements then display the current year, number of new memberships, and total number of memberships up to that year. Finally, the @memyear is incremented by one. This loop runs as long as the @memyear variable is less than or equal to the current year.

```
CREATE PROCEDURE member_loop
AS
   DECLARE @memyear int, @total_mems int, @new_mems int
   SET @total_mems = 0
   SELECT @memyear = DATEPART(yyyy, MIN(memdate)) FROM members
   WHILE @memyear <= DATEPART(yyyy, GETDATE())
      BEGIN
         SELECT @new_mems = count(*) FROM members
         WHERE DATEPART(yyyy, memdate) = @memyear
         SET @total_mems = @total_mems + @new_mems
         PRINT 'Year: ' + CONVERT(char(4), @memyear)
         PRINT 'New members: ' + CONVERT(char(3), @new_mems)
         PRINT 'Total members: ' + CONVERT(char(3), @total_mems)
         PRINT ''
         SET @memyear = @memyear + 1
      END
```

Exercise 5: Working with Loops

In this exercise, you will practice writing procedures that use loops to execute statements repeatedly. The *Extra* question uses data from the EMPLOYEE table.

1. Write a procedure that uses a loop to display a name 10 times on the screen. Use a parameter variable to supply the name to the procedure.

2. Modify the procedure from question 1 to display a number next to each line, incrementing the number each time the loop runs. Add a condition to exit the procedure after 5 iterations if the name is 5 letters long. (You can use the LEN function to determine the length of the name.)

3. Write a procedure that starts with the number 2 and doubles the number with each iteration until you reach 32. End the loop after you reach the number 32; then display the number of loop iterations, so the output will look as follows:

```
2
4
8
16
32
The loop executed 5 times.
```

Important note: If you plan to do the Extra and Challenge questions, consider making a copy of the Employee table so you can test both procedures effectively. To make a copy of the table, run the following statement:

```
SELECT * INTO emp2 FROM employee
```

The solution to the *Challenge* question refers to this copy of the table, named **Emp2**.

Extra: The budget for your tour guides has expanded, so you've decided to give all your tour guides raises. Using a loop, give each tour guide a 5% raise until the average pay for tour guides equals or exceeds $11 per hour. Display the current average tour guide rate at the beginning of the procedure, and recalculate and display the average rate after each iteration of the loop.

Challenge: Modify the procedure in the Extra question to include a condition to exit the loop when any one tour guide exceeds the maximum rate per hour for tour guides. (The maximum rate is included in the JOBS table.)

Controlling Transactions in Your Procedures

In *Introduction to SQL*, you learned about some basic statements for creating and controlling transactions. By default, SQL Server automatically commits every statement you issue, so you cannot easily "undo" changes to your data. In order to explicitly commit or roll back statements in SQL Server, you must include the statements in a **transaction**. A transaction is a unit of work (one or more SQL statements) that can be explicitly committed or rolled back.

To start a new transaction in SQL Server, issue the **BEGIN TRANSACTION** command. You can end the transaction by executing a COMMIT TRANSACTION or ROLLBACK TRANSACTION command. The **COMMIT TRANSACTION** statement (or simply **COMMIT**) saves all modifications made by the transaction to the database. The **ROLLBACK TRANSACTION** statement (abbreviated **ROLLBACK**) rolls back ("undoes") all changes made by the transaction.

Conditionally Committing or Rolling Back Changes

In your stored procedures, it may be useful to execute commands in a transaction and then evaluate a condition to determine whether you want to commit or roll back the changes made by the transaction.

In the Extra part of the previous exercise, you were asked to write a stored procedure that updates all the tour guide salaries until the average equals or exceeds $11.00 per hour. When the procedure completed, the average tour guide rate exceeded this limit. Using transactions, it is possible to "undo" the last update to the hourly rates, which pushed the average over the $11.00 limit.

Note: In the procedure below, the maximum average hourly rate was raised to $13.00, so the procedure has an effect on the table data (since the initial average now exceeds $11.00).

```
ALTER PROCEDURE trg_raise_loop
AS
   DECLARE @avg_rate numeric(7,2)
   -- get current average rate
   SELECT @avg_rate = AVG(hourrate)
   FROM employee
   WHERE jobcode = 'TRG'
   PRINT 'Initial average rate: $' + CONVERT(char(10), @avg_rate)
   -- use loop to assign 5% raises until average >= $13/hour
   WHILE @avg_rate < 13
      BEGIN
         -- start new transaction
         BEGIN TRANSACTION
         UPDATE employee
         SET hourrate = hourrate * 1.05
         WHERE jobcode = 'TRG'
```

```
        -- calculate new average
        SELECT @avg_rate = AVG(hourrate)
        FROM employee
        WHERE jobcode = 'TRG'
        -- print new average
        PRINT 'New average rate: ' + CONVERT(char(10), @avg_rate)
        -- if average exceeds limit, rollback last update
        IF @avg_rate > 13
           BEGIN
              PRINT 'Average exceeds limit.'
              PRINT 'Rollback last update.'
              ROLLBACK TRANSACTION
           END
        -- if average is still within limit, commit changes
        ELSE
           BEGIN
              PRINT 'Average still within limit.'
              PRINT 'Commit last update.'
              COMMIT TRANSACTION
           END
    END
```

The BEGIN TRANSACTION statement starts a new transaction for each iteration of the loop. The procedure then executes the update and calculates the new average. If the new average exceeds the limit, the previous update is rolled back; if the new average is still within the acceptable range, the changes are committed, and the statements inside the loop are executed again.

Working with Cursors

Earlier in class, you practiced selecting data from tables and storing it in local variables. One limitation of these examples, however, is that you can only effectively handle queries that return a single row, since one local variable cannot store multiple values.

Cursors allow you to process rows in a result set one at a time. In this lesson, you will learn the techniques used to declare and manage cursors, and you will practice writing stored procedures that use cursors to process data.

Managing Cursors

To successfully use a cursor, you must follow these five steps:

1. **Declare** the cursor. In this declaration, you name the cursor and associate it with a SELECT statement.

2. **Open** the cursor.

3. **Fetch** a row of returned data from the cursor.

4. **Close** the cursor to release the active set of data.

5. **Deallocate** the cursor to undefine it and free up memory.

Each of these steps is broken down in the following sections. At the end of this section, you will write a stored procedure that retrieves and displays information about the first five museum members.

Declaring a Cursor

When you declare a cursor, you name the cursor and associate it with a query that will retrieve the data you want to process. Use the following syntax for simple cursor declaration:

```
DECLARE cursor_name CURSOR
   FOR select statement
```

The SELECT statement you use to define the cursor follows the same format as a normal SQL SELECT statement. In addition to the required SELECT and FROM clauses, you can optionally include WHERE and ORDER BY clauses to identify the data to retrieve and specify an order for the returned rows.

Note: You cannot include the keywords COMPUTE, COMPUTE BY, FOR BROWSE, or INTO in a SELECT statement used in a cursor declaration.

Keep in mind that you must declare a cursor before you can reference the cursor in your program. The following statement declares a cursor named **cur_mem** and associates the cursor with a query that retrieves members' first names, last names, and membership dates from the MEMBERS table, organized from the earliest memberships to the most recent.

```
DECLARE cur_students CURSOR
   FOR SELECT firstname, lastname, memdate FROM members
       ORDER BY memdate
```

Note: Many programmers use a prefix or suffix, like **cur_** in the cursor name. This isn't required, but it can make it easier to identify cursors in your code.

Also note: There are a number of optional attributes that you can define for a cursor. The following is the complete, ANSI-standard syntax for declaring a cursor:

```
DECLARE cursor_name [ INSENSITIVE ] [ SCROLL ] CURSOR
   FOR select statement
   [ FOR { READ ONLY | UPDATE [ OF column_list ] } ]
```

The **INSENSITIVE** option specifies that the cursor will use a temporary copy of the data, so the cursor does not allow modifications, and any modifications to the base tables are not reflected in subsequent fetches. (If you omit this option, committed deletes and updates are reflected in subsequent fetches.)

SCROLL specifies that you can use any of the following FETCH options: FIRST, LAST, PRIOR, NEXT, RELATIVE, and ABSOLUTE. If this option is not specified, only the NEXT option is available. (This is discussed in more detail in the next lesson.)

The **READ ONLY** option prevents you from making updates through the cursor. (By default, the cursor can be updated.) Alternately, you can specify **UPDATE OF *column_list*** to limit updates to specific columns. (If you don't specify a column list, all columns can be updated.)

In addition to these standard options, Transact-SQL supports its own options for cursor declarations (although these options are beyond the scope of this course). You can find a complete description of the DECLARE CURSOR syntax at *http://msdn2.microsoft.com/en-us/library/aa258831(SQL.80).aspx*.

Opening the Cursor

After you declare the cursor, the next step is to open the cursor, using the command:

```
OPEN cursor_name
```

The **OPEN** statement populates the result set, using the SELECT statement you defined for the cursor.

The statement below opens the **cur_mem** cursor:

```
OPEN cur_mem
```

Fetching Data

Next, use the FETCH statement to return a single row from the result set. FETCH is often used to fetch the returned data into local variables.

The **FETCH** statement uses this syntax to fetch the next row from the result set:

```
FETCH [ NEXT FROM ] cursor_name [ INTO variable_list ]
```

For example, this FETCH statement returns the next record from the cur_mem cursor.

```
FETCH NEXT FROM cur_mem
```

This FETCH statement does not fetch the data into variables. If you don't assign the data to variables, the FETCH statement returns the data to the client.

Note: The next record is the default record to fetch, so technically, you can omit the NEXT FROM keywords if you are fetching the next record, for example:

```
FETCH cur_mem
```

It is also possible to fetch the first record, previous record, last record, or a specific record, using either an absolute or relative record number. In order to do this, you must specify the **SCROLL** option when you declare the cursor. This is discussed in more detail later in this course.

In most cases, the FETCH statement will appear in a loop, which terminates based on a specific condition or after a fixed number of iterations.

Closing the Cursor

When you're finished with the cursor, use the **CLOSE** statement to release the current result set associated with the cursor.

```
CLOSE cursor_name
```

You should include this statement after all rows have been processed.

The following statement closes the **cur_mem** cursor:

```
CLOSE cur_mem
```

Once you've closed a cursor, you can no longer fetch data from the cursor. (You can, however, reopen the cursor and then fetch data.)

The CLOSE command must be issued on an open cursor; you cannot close a cursor that has not yet been opened or has already been closed.

Deallocating the Cursor

The final step is to deallocate the cursor, which releases the resources used by the cursor. To deallocate a cursor, use the **DEALLOCATE** command:

```
DEALLOCATE cursor_name
```

To deallocate the cur_mem cursor, run this statement (after you close the cursor):

```
DEALLOCATE cur_mem
```

Putting It All Together

To see how these statements work in action, you can create a new procedure (named **member_cursor**, in the following example) that uses a cursor to process the first names, last names, and membership dates of the first five members. The **FETCH** statement, included in a WHILE loop, returns the data from the current row to the client. A counter variable is used to execute the loop five times.

```
CREATE PROCEDURE member_cursor
AS
   -- declare and initialize counter
   DECLARE @counter int
   SET @counter = 1
   -- declare cursor
   DECLARE cur_mem CURSOR FOR
      SELECT firstname, lastname, memdate
      FROM members
      ORDER BY memdate
   -- open cursor
   OPEN cur_mem
   -- fetch next member from cursor and increment counter
   -- stop after first five members are processed
   WHILE @counter <= 5
      BEGIN
         FETCH NEXT FROM cur_mem
         SET @counter = @counter + 1
      END
```

```
-- close cursor
CLOSE cur_mem
-- deallocate cursor
DEALLOCATE cur_mem
```

Note: In general, there are many alternatives to using cursors, and cursors aren't always the best choice (more on cursor uses and disadvantages in the next lesson). For example, as an alternative to the cursor in the previous example, you can use the **TOP *n*** clause in a SELECT statement to select the first *n* records in a record set (where *n* is the number or percentage of records to select). The following shows the syntax for using TOP in a SELECT clause:

```
SELECT TOP (n) [PERCENT] [WITH TIES] column_list
FROM table
[ WHERE condition ]
[ ORDER BY column ]
```

In this syntax, *n* represents the number or percentage of records to select from the beginning of the result set. If *n* is a percentage, specify the **PERCENT** keyword.

You can also use the optional **WITH TIES** clause to include ties in the result set, even if that means returning more than *n* records. The WITH TIES clause can be used only with SELECT statements, and the ORDER BY clause must be used in conjunction with the WITH TIES clause.

The following query uses the TOP *n* clause to retrieve the first five members from the Members table:

```
SELECT TOP (5) firstname, lastname, memdate
FROM members
ORDER BY memdate
```

Fetching Data into Variables

In the previous example, your FETCH statement returned each row to the client, so the results included one row for each of the first five members. Alternately, you can fetch data from your cursor into local variables. This provides more flexibility in your output, since you can manipulate the variables and return their data in different ways.

To use variables to store data from your cursor, you must declare a local variable for each column included in the SELECT statement when you defined the cursor. The following statement declares local variables for the member's first name, last name, and membership date.

```
DECLARE @fname varchar(15), @lname varchar(15), @mdate smalldatetime
```

Next, you must revise the FETCH statement to fetch the data into the variables. To do this, include the **INTO** keyword, followed by the variable list, in the FETCH statement.

```
FETCH NEXT FROM cur_mem INTO @fname, @lname, @mdate
```

Note: The variables in the INTO clause must correspond in number and position to the columns listed in the SELECT statement used to declare the cursor.

Finally, you can include PRINT statements to return the values of the variables to the client. For example, you can include the following PRINT statement in the loop to return the data for each row:

```
PRINT 'Member ' + CONVERT(char(3), @counter) + @fname + ' ' + @lname
PRINT 'Member date: ' + CONVERT(char(20), @mdate, 107)
PRINT ''
```

These statements return the membership information in this format:

```
Member 1  LOU SMALLS
Member date: Dec 14, 2002

Member 2  JENNY JEFFRIES
Member date: Dec 14, 2002

Member 3  RICHARD POWERS
Member date: Aug 10, 2003

Member 4  TODD CHANG
Member date: May 15, 2004

Member 5  ANNE COOPER
Member date: Jul 05, 2004
```

The entire procedure is printed below, with the changes in bold.

```
ALTER PROCEDURE member_cursor
AS
   -- declare counter and variables for each column
   DECLARE @counter int, @fname varchar(15), @lname varchar(15),
           @mdate smalldatetime
   -- initialize counter
   SET @counter = 1
   -- declare cursor
   DECLARE cur_mem CURSOR FOR
      SELECT firstname, lastname, memdate
      FROM members
      ORDER BY memdate
   -- open cursor
   OPEN cur_mem
   -- fetch next member from cursor and increment counter
   -- stop after first five members are processed
   WHILE @counter <= 5
      BEGIN
         FETCH NEXT FROM cur_mem INTO @fname, @lname, @mdate
         PRINT 'Member ' + CONVERT(char(3), @counter) + @fname + ' '
               + @lname
         PRINT 'Member date: ' + CONVERT(char(20), @mdate, 107)
```

```
      PRINT ''
      SET @counter = @counter + 1
   END
-- close cursor
CLOSE cur_mem
-- deallocate cursor
DEALLOCATE cur_mem
```

Processing All Rows in the Cursor

In the previous example, the WHILE loop used a limit of 5 to process only the first five records in the cursor. In the following examples, you will learn a few techniques for processing all rows in the cursor—first by relying on the COUNT function to count the number of rows in the SELECT statement and then by using the @@FETCH_STATUS function to stop the loop when the fetch is no longer successful.

If you want to change the previous example to process all members in order by membership date, you can declare a variable to hold the total number of rows in the Members table. Then modify the WHILE loop condition to refer to this variable.

The revisions to the member_cursor procedure are formatted in bold. (There are no changes to the code after the ellipsis.)

```
ALTER PROCEDURE member_cursor
AS
   -- declare counter and variables for each column
   DECLARE @counter int, @fname varchar(15), @lname varchar(15),
           @mdate smalldatetime, @total_mems int
   -- initialize counter
   SET @counter = 1
   -- initialize @total_mems to total number of member records
   SELECT @total_mems = COUNT(*) FROM members
   -- declare cursor
   DECLARE cur_mem CURSOR FOR
      SELECT firstname, lastname, memdate
      FROM members
      ORDER BY memdate
   -- open cursor
   OPEN cur_mem
   -- fetch next member from cursor and increment counter
   -- stop after all members are processed
   WHILE @counter <= @total_mems
      BEGIN
   ...
```

Note: Because the SELECT statement used in the cursor declaration returns all rows from the Members table, the variable value should reflect the total number of rows in the cursor.

Using the @@FETCH_STATUS Function

Transact-SQL provides a three **cursor functions** that allow you to retrieve information about a cursor.

Function	Description
@@CURSOR_ROWS	Returns an integer representing the number of rows in the most recently opened cursor.
@@FETCH_STATUS	Returns an integer representing the status of the last FETCH statement issued against any cursor that is currently open.
CURSOR_STATUS	Returns an integer representing the status of a specific cursor. Use within a stored procedure to determine whether or not the procedure has returned a cursor and result set for a given parameter.

This lesson focuses on using the **@@FETCH_STATUS** function to check the status of the previous FETCH statement. The function returns an integer representing the status of the last FETCH issued against any open cursor. It returns three possible values:

Return Value	Description
0	Fetch was successful
-1	Fetch failed, or the row was beyond the result set
-2	Row fetched is missing

You can use this function in a loop condition to end the loop when there are no more rows to fetch, as shown below:

```
ALTER PROCEDURE member_cursor
AS
   -- declare counter and variables for each column
   DECLARE @counter int, @fname varchar(15), @lname varchar(15),
        @mdate smalldatetime
   -- initialize counter
   SET @counter = 1
   -- declare cursor
   DECLARE cur_mem CURSOR FOR
      SELECT firstname, lastname, memdate
      FROM members
      ORDER BY memdate
   -- open cursor
   OPEN cur_mem
   -- fetch first record
   FETCH NEXT FROM cur_mem INTO @fname, @lname, @mdate
   WHILE @@FETCH_STATUS = 0
      BEGIN
         PRINT 'Member ' + CONVERT(char(3), @counter) +
               @fname + ' ' + @lname
         PRINT 'Member date: ' + CONVERT(char(20), @mdate, 107)
         PRINT ''
         SET @counter = @counter + 1
```

```
      FETCH NEXT FROM cur_mem INTO @fname, @lname, @mdate
   END
-- close cursor
CLOSE cur_mem
-- deallocate cursor
DEALLOCATE cur_mem
```

Note that the loop continues as long as the @@FETCH_STATUS function returns 0 (meaning that the fetch was successful). Also note that you must do one fetch before the loop. If not, the @@FETCH_STATUS function will return -1 before the first fetch, and the statements in the loop will not execute.

Alternative: To avoid including an initial FETCH statement outside of the loop, you can create an "infinite loop" and break out of the loop when the @@FETCH_STATUS is not equal to 0. Note that the lines that are no longer necessary have been commented out and formatted in italic font.

```
ALTER PROCEDURE member_cursor
AS
   -- declare counter and variables for each column
   DECLARE @counter int, @fname varchar(15), @lname varchar(15),
           @mdate smalldatetime
   -- initialize counter
   SET @counter = 1
   -- declare cursor
   DECLARE cur_mem CURSOR FOR
      SELECT firstname, lastname, memdate
      FROM members
      ORDER BY memdate
   -- open cursor
   OPEN cur_mem
   --FETCH NEXT FROM cur_mem INTO @fname, @lname, @mdate
   WHILE 1=1
      BEGIN
         FETCH NEXT FROM cur_mem INTO @fname, @lname, @mdate
         IF @@FETCH_STATUS <> 0
            BREAK
         PRINT 'Member ' + CONVERT(char(3), @counter) +
               @fname + ' ' + @lname
         PRINT 'Member date: ' + CONVERT(char(20), @mdate, 107)
         PRINT ''
         SET @counter = @counter + 1
      END
   -- close cursor
   CLOSE cur_mem
   -- deallocate cursor
   DEALLOCATE cur_mem
```

Note that the IF test must appear after the FETCH statement but before the PRINT statements. If you test the condition before the first FETCH, the @@FETCH_STATUS function will return -1, and the procedure will exit the loop before the first fetch. If you test the condition after the PRINT statements inside the loop, the final member will be printed twice.

Also keep in mind: If you don't include any kind of condition to exit this endless loop, the procedure will keep printing the last record in the cursor.

A final note about @@FETCH_STATUS: It's important to understand that @@FETCH_STATUS is global to all cursors on a connection. This function reflects the status of the last fetch executed; you cannot specify which cursor the function acts on. This is important if you have multiple cursors open at the same time.

Using a Parameter Variable with a Cursor

You can refer to variables in the SELECT statement you use to declare the cursor. For example, in the previous procedure, you may want to add criteria to your SELECT statement, so the cursor includes only records for specific member types. Instead of hard-coding a member type into the query, you can create a parameter variable and have the user supply a member code when he runs the procedure.

```
ALTER PROCEDURE member_cursor
   --declare parameter variable for member type
   @mcode char(2) = NULL
AS
   -- declare counter and variables for each column
   DECLARE @counter int, @fname varchar(15), @lname varchar(15),
          @mdate smalldatetime
   -- initialize counter
   SET @counter = 1
   -- declare cursor
   DECLARE cur_mem CURSOR FOR
      SELECT firstname, lastname, memdate
      FROM members
      WHERE memcode = @mcode
      ORDER BY memdate
   -- open cursor
   OPEN cur_mem
...
```

Using Cursors for Reporting

Cursors may be useful for generating running totals and displaying individual records along with group subtotals. For example, assume you want to create a report that displays information about the museum collections that were donated in 2006 and 2007. The report below displays a description of all collections donated in 2006 and 2007, along with the donation date, the collection value, and a running total of the collection donations. At the end of the report, you will see a summary of the 2006 donations, 2007 donations, and total donations for 2006-2007.

```
2006-2007 DONATIONS
-----------------------------------------------

Description                    Date      Value     Total Donation
ORIGINAL STARFLEET UNIFORM     04/05/06  250.00    250.00
BLACKLIGHT POSTERS             04/15/06  100.00    350.00
TICKLE ME ELMO DOLL            05/15/06  10.00     360.00
SIGNED WOODSTOCK POSTER        06/07/06  200.00    560.00
SLOGAN BUTTONS                 07/22/06  20.00     580.00
POKEMON TRADING CARDS          07/22/06  100.00    680.00
BEANIE BABIES                  07/22/06  100.00    780.00
CLASSIC PEZ DISPENSERS         10/20/06  95.00     875.00
1950S JUKEBOX                  11/01/06  500.00    1375.00
VINTAGE TV                     11/01/06  125.00    1500.00
BETA MAX PLAYER                11/15/06  25.00     1525.00
BETA MAX MOVIES                11/15/06  50.00     1575.00
PILLBOX HAT                    01/28/07  7.50      1582.50
POODLE SKIRT                   02/26/07  50.00     1632.50
SADDLE SHOES                   02/26/07  15.00     1647.50

2006 Total Donations: 1575.00
2007 Total Donations: 72.50
Total for 2006-2007: 1647.50
```

One way to create this kind of report is to use a cursor. Cursors allow you to create running totals by incrementing a local variable after each row is processed. You can also create variables to hold subtotals for the 2006 and 2007 donations and increment them conditionally, based on the donation date for the collection.

The following procedure uses a cursor to create the donations report printed earlier.

```
CREATE PROCEDURE collections_cursor
AS
   --declare local variables
   DECLARE @descrip char(30), @don_date smalldatetime,
           @coll_val numeric(7,2), @2006_total numeric(9,2),
           @2007_total numeric (9,2), @grand_total numeric(9,2)
   --initialize subtotal and grand total variables
   SET @2006_total = 0
   SET @2007_total = 0
   SET @grand_total = 0
   --print heading
   PRINT '2006-2007 DONATIONS'
   PRINT '-----------------------------------------------'
   PRINT ''
   PRINT 'Description            Date      Value     Total Donation'
   --declare cursor
   DECLARE cur_coll CURSOR
      FOR SELECT coll_desc, donate_date, coll_value
      FROM collections
      WHERE donate_date >= '01/01/2006'
      ORDER BY donate_date
   --open cursor
   OPEN cur_coll
   --first first row
   FETCH NEXT FROM cur_coll INTO @descrip, @don_date, @coll_val
   --continue loop as long as fetch is successful
   WHILE @@FETCH_STATUS = 0
      BEGIN
         --increment 2006 subtotal if date is in 2006
         IF DATEPART(yyyy, @don_date) = 2006
            SET @2006_total = @2006_total + @coll_val
         --increment 2006 subtotal if date is in 2007
         ELSE IF DATEPART(yyyy, @don_date) = 2007
            SET @2007_total = @2007_total + @coll_val
         --increment grand total
         SET @grand_total = @2006_total + @2007_total
         --print row
         PRINT @descrip + ' ' + CONVERT(char(10), @don_date, 1)
               + CONVERT(char(10), @coll_val)
               + CONVERT(char(12), @grand_total)
         --fetch next row
         FETCH NEXT FROM cur_coll INTO @descrip, @don_date, @coll_val
      END
   --print summary
   PRINT ''
   PRINT '2006 Total Donations: ' + CONVERT(char(12), @2006_total)
   PRINT '2007 Total Donations: ' + CONVERT(char(12), @2007_total)
   PRINT 'Total for 2006-2007: ' + CONVERT(char(12), @grand_total)
   --close cursor
   CLOSE cur_coll
   --deallocate cursor
   DEALLOCATE cur_coll
```

Exercise 6: Using Basic Cursors

In this exercise, you will practice declaring and managing cursors to process records from the EMPLOYEE and JOBS tables.

1. Write a procedure that selects all tour guides from the EMPLOYEE table. Use a basic loop to fetch the data into variables and print each tour guide's first name, last name, and hourly rate in this format (although the exact hourly rate may be different from the rate shown below):

   ```
   Employee name: GINA PEREZ
   Hourly rate: $11.25
   ```

   End the loop after all rows have been processed.

2. Modify the procedure so that the user can enter a job code when he or she runs the procedure. The procedure will then process the employees who have that code.

3. Write a procedure to print a weekly payroll report for non-exempt employees (secretaries and tour guides). Display each non-exempt employee's ID, first name, last name, job code, and weekly pay, assuming that every employee worked 40 hours during the week. Additionally, include the total weekly pay for secretaries and tour guides, as well as a weekly grand total for all non-exempt employees. The finished report should look something like this (although the exact names and numbers may be different).

```
                WEEKLY EMPLOYEE PAYROLL
--------------------------------------------------

ID    Last Name       First Name      Job   Weekly Pay
A1465 PICKNEY         ROBERTA         SEC   $460.80
A5150 PEREZ           GINA            TRG   $602.80
A4115 FLOWERS         JANETTE         SEC   $402.00
A6395 CHEN            MICHAEL         TRG   $506.40
A1333 CARTER          PATRICIA        TRG   $456.00
A1568 LEVY            RAY             TRG   $456.00
A5015 RICCI           BETTY           TRG   $456.00
A7812 LEE             JUDY            TRG   $549.20

Total pay for secretaries: $862.80
Total pay for tour guides: $3026.40

Total pay for all non-exempt employees: $3889.20
```

Working with More Advanced Cursors

In the previous lesson, you practiced using relatively simple cursors to process rows from a result set one at a time. These cursors were used to retrieve data only (not to update or delete data), and the fetches were limited to retrieving the next record in the result set.

This lesson introduces a few more advanced cursors options, including using the SCROLL option to select first, last, and previous records from a cursor, as well as specific records, using absolute and relative position. You will explore updatable cursors and limit the cursor's functionality by making it read-only or restricting the updates to specific columns. You will also practice using nested cursors to loop through groups of data, as well as individual records. Finally, at the end of this lesson, you will discuss some limitations of cursors.

Using the SCROLL Option

In the previous examples, all of your FETCH statements were limited to fetching the next record from the result set. If you declare a cursor with the **SCROLL** option, you can use FETCH statements to retrieve records from the result set in a different order.

Use the following syntax to declare a cursor with the SCROLL option:

```
DECLARE cursor_name SCROLL CURSOR
   FOR select statement
```

For example:

```
DECLARE cur_mem SCROLL CURSOR
   FOR SELECT empid, hiredate, hourrate
   FROM employee
   ORDER BY hiredate
```

Now, in your FETCH statements, you can use these options to retrieve a row from the cursor:

Option	Description
FIRST	Returns the first row in the cursor.
LAST	Returns the last row in the cursor.
PRIOR	Returns the row immediately preceding the current row.
NEXT	Returns the row immediately following the current row. (This is the default.)
ABSOLUTE *n*	If *n* is positive, returns the row *n* rows from the beginning of the cursor. If *n* is negative, returns the row *n* rows before the end of the cursor. (*n* can be an integer, or you can use a variable of type int, smallint, or tinyint.)
RELATIVE *n*	If *n* is positive, returns the row *n* rows beyond the current row. If *n* is negative, returns the row *n* rows before the current row. (You can specify an integer for *n*, or you can use a variable of type int, smallint, or tinyint.)

For example, the following FETCH statement would return the sixth row from the **cur_mem** cursor:

```
FETCH ABSOLUTE 6 FROM cur_mem
```

Note: You must specify the FROM keyword with these options.

The **scrolling** procedure demonstrates how to declare a cursor with the SCROLL option and use the FETCH options described above to fetch records from the cursor:

```
CREATE PROCEDURE scrolling
AS
   DECLARE cur_emp SCROLL CURSOR
      FOR SELECT empid, hiredate, hourrate
      FROM employee
      ORDER BY hiredate
   OPEN cur_emp
   PRINT 'First record...'
   FETCH FIRST FROM cur_emp
   PRINT ''
   PRINT 'Last record...'
   FETCH LAST FROM cur_emp
   PRINT ''
   PRINT 'Previous record...'
   FETCH PRIOR FROM cur_emp
   PRINT ''
   PRINT 'Fifth record...'
   FETCH ABSOLUTE 5 FROM cur_emp
   PRINT ''
   PRINT 'Fifth record from end...'
   FETCH ABSOLUTE -5 FROM cur_emp
   PRINT ''
   PRINT 'Add 2 to current record...'
   FETCH RELATIVE 2 FROM cur_emp
   PRINT ''
   CLOSE cur_emp
   DEALLOCATE cur_emp
```

Using Cursors for Updates

Unless you specify otherwise, a cursor allows you to update or delete rows in the result set. To perform an update or delete on the most recently fetched row in a cursor, use the syntax

```
WHERE CURRENT OF cursor_name
```

In the WHERE clause of the UPDATE or DELETE statement.

Assume you want to update the minimum hourly rates for all the jobs in the JOBS table, according to these conditions:

- If the current minimum rate is less than $9.50, update the rate to $10.00 per hour.
- If the job is classified as exempt, add 3% to the current minimum rate.
- If the job is non-exempt, add 5% to the current minimum rate.

The **update_minrate** procedure uses a cursor to select and update each job in the JOB2 table (a copy of the JOBS table). Note the syntax of the UPDATE statement in the loop:

```
UPDATE job2
SET minrate = @new_min
WHERE CURRENT OF cur_job
```

The WHERE clause instructs the UPDATE to operate on the row most recently fetched from the cursor. The entire procedure is printed below:

```
CREATE PROCEDURE update_minrate
AS
   DECLARE @jcode char(3), @jclass varchar(10),
           @old_min numeric(5,2), @new_min numeric(5,2)
   DECLARE cur_job CURSOR
      FOR SELECT jobcode, class, minrate FROM job2
   OPEN cur_job
   -- fetch first record
   FETCH NEXT FROM cur_job INTO @jcode, @jclass, @old_min
   WHILE @@FETCH_STATUS = 0
      BEGIN
      -- assign new rate based on conditions
         IF @old_min < 9.5
            SET @new_min = 10
         ELSE IF @jclass = 'EXEMPT'
            SET @new_min = (@old_min * .03) + @old_min
         ELSE IF @jclass = 'NON-EXEMPT'
            SET @new_min = (@old_min * .05) + @old_min
         -- update jobs table for current record
         UPDATE job2
         SET minrate = @new_min
         WHERE CURRENT OF cur_job
         -- print old and new minimum rates
         PRINT 'Job: ' + @jcode
         PRINT 'Old minimum rate: ' + CONVERT(char(5), @old_min)
         PRINT 'New minimum rate: ' + CONVERT(char(5), @new_min)
         PRINT ''
         -- fetch next record
         FETCH NEXT FROM cur_job INTO @jcode, @jclass, @old_min
      END
   CLOSE cur_job
   DEALLOCATE cur_job
```

Restricting the Updatable Columns

In the procedure above, the cursor allows updates to any column in the SELECT statement, so you could, for example, use the cursor to update the job code or classification, causing unwanted results. If you want to restrict updates to one or more specific columns, you can specify the columns when you declare the cursor. Not only does this prevent unwanted updates, but it can help to clarify the use of the cursor.

To specify a list of updatable columns, include the **FOR UPDATE OF *column_list*** clause when you declare the cursor:

```
DECLARE cursor_name CURSOR
   FOR select statement
   FOR UPDATE OF column_list
```

In the previous procedure, you could limit updates to the Minrate column by including the clause in bold in the cursor declaration:

```
DECLARE cur_job CURSOR
   FOR SELECT jobcode, class, minrate FROM job2
   FOR UPDATE OF minrate
```

Now, if you attempt to use the cursor to update a column other than Minrate, you will see the following error message:

```
The cursor has a FOR UPDATE list and the requested column to be updated is
not in this list.
```

Declaring a Read-Only Cursor

If you want to prevent all updates and deletes, you can declare a cursor as read-only, using this syntax:

```
DECLARE cursor_name CURSOR
   FOR select statement
   FOR READ ONLY
```

If you attempt an update with a read-only cursor, you will get this error message:

```
The cursor is READ ONLY.
```

Nesting Cursors

It is possible to nest cursors to process multiple data sets. For example, you may need to process groups with the outer cursor and individual records with the inner cursor. This allows you to calculate running totals within each group and reset them for the next group.

Think back to the reporting example that asked you to calculate total donations for museum collections donated in 2006 and 2007. Using nested cursors, you can calculate running totals for every year in which a collection was donated, according to the donation dates recorded in the Collections table.

The following procedure uses two cursors: one (named **cur_years**) to retrieve each year from the Donations table, and a second (**cur_coll**) to process each collection donated in a given year.

The outer loop, which begins after the first row is fetched from the cur_years cursor, performs the following tasks:

- Reset subtotal to 0 for the current year.
- Print a subheading for the current year and column headings for the collection information.
- Declare, open, use, close, and deallocate the second cursor (cur_colls).
- Increment the grand total by adding the current subtotal to the grand total.
- Print the subtotal for the current year and the running grand total.
- Fetch the next row from the cur_years cursor.

The inner loop, which begins after the first row is fetched into the second cursor, is used to process the collections for the current year. The statements in the loop perform these tasks:

- Increment the subtotal by adding the donation value of the most recently fetched collection.
- Print the information about the current collection (description, donation date, and value), along with the running subtotal for the current year.
- Fetch the next row from the cur_coll cursor.

The entire procedure is printed below:

```
CREATE PROCEDURE nested_cursor
AS
   --declare local variables
   DECLARE @descrip char(30), @don_date smalldatetime,
           @coll_val numeric(7,2), @subtotal numeric(9,2),
           @grand_total numeric(9,2), @coll_year char(4)
   --initialize grand total variable
   SET @grand_total = 0
```

```
--print main heading
PRINT 'COLLECTION DONATIONS'
PRINT '-------------------------------------------------'
PRINT ''
--declare cursor for collection years
DECLARE cur_years CURSOR
   FOR SELECT DISTINCT DATEPART(yyyy, donate_date)
   FROM collections
   ORDER BY DATEPART(yyyy, donate_date)
--open first cursor
OPEN cur_years
--fetch first year from cur_years cursor
FETCH NEXT FROM cur_years INTO @coll_year
--outer loop, continues until no more years in cur_years cursor
WHILE @@FETCH_STATUS = 0
   BEGIN
      --initialize subtotal (reset for each year)
      SET @subtotal = 0
      --print subheadings for yearly donations
      PRINT @coll_year + ' DONATIONS'
      PRINT '-----------------'
      PRINT 'Description      Date      Value     Total Donation'
      --declare second cursor to process individual collections
      DECLARE cur_coll CURSOR
         FOR SELECT coll_desc, donate_date, coll_value
         FROM collections
         WHERE DATEPART(yyyy, donate_date) = @coll_year
         ORDER BY donate_date
      --open cursor
      OPEN cur_coll
      --fetch first row
      FETCH NEXT FROM cur_coll INTO @descrip, @don_date, @coll_val
      --inner loop continues as long as
      --there are collections for current year
      WHILE @@FETCH_STATUS = 0
         BEGIN
            --increment subtotal
            SET @subtotal = @subtotal + @coll_val
            --print row
            PRINT @descrip + ' ' + CONVERT(char(10), @don_date, 1)
               + CONVERT(char(10), @coll_val)
               + CONVERT(char(12), @subtotal)
         --fetch next row
         FETCH NEXT FROM cur_coll INTO @descrip, @don_date, @coll_val
      END
      --close second cursor
      CLOSE cur_coll
      --deallocate second cursor
      DEALLOCATE cur_coll
      --increment grand total
      SET @grand_total = @grand_total + @subtotal
```

```
        --print subtotal and grand total
        PRINT ''
        PRINT 'Subtotal for ' + @coll_year + ': $'
              + CONVERT(char(10), @subtotal)
        PRINT 'New grand total: $'
              + CONVERT(char(10), @grand_total)
        PRINT ''
        --get next year
        FETCH NEXT FROM cur_years INTO @coll_year
    END
    --close first cursor
    CLOSE cur_years
    --deallocate first cursor
    DEALLOCATE cur_years
```

It's important to note that the statements for declaring, opening, closing, and deallocating the second cursor all appear within the outer loop. This is because the DECLARE statement relies on a variable that changes (@coll_year) to populate the cursor. Since cursor variable values cannot change after a cursor is declared, you have to close, deallocate, and re-declare the cursor to repopulate the cursor based on the new variable value.

Understanding the Limitations of Cursors

Over the past two lessons, you've learned about and practiced several techniques for using cursors to process SQL Server data. While cursors can be very useful in certain circumstances, most SQL Server programmers warn against relying too heavily cursors. In many cases, you can accomplish the same task more efficiently by using other techniques, such as set-based processing.

In general, cursors are much less efficient than equivalent set-based solutions. For example, if you are performing a simple update based on some condition, you can simply run an UPDATE statement and specify the condition in a WHERE clause. You do not need to use a cursor to evaluate and update the qualifying records one at a time one. In addition to the performance penalty you experience with cursors, set-based statements are often much easier to write.

Cursors, however, may still be useful in certain scenarios—for example, when no alternate set-based solution exists or when you are processing relatively few rows. Another potential benefit of cursors relates to locking tables: Cursors allow you to lock only part of a table (without locking the whole table), so other users can continue to work with table data when you are running a batch processing job. Additionally, with cursors, you may be able to better manage the database transaction log, since only one row is processed at a time.

Exercise 7: Using Advanced Cursors (Optional)

In this exercise, you will practice working with updatable and nested cursors. This exercise uses data from the EMPLOYEE, EVAL, and JOBS tables.

1. Earlier in class, you wrote a few procedures that gave employees a raise based on an evaluation score. Evaluation scores for individual employees are stored in the **EVAL** table in your database. There is a record for most employees, but some employees do not have evaluation scores.

   Write a procedure that updates employees' hourly rates in the EMPLOYEE table based on the evaluation scores in the EVAL table. Use the following table to determine the raise percentage that corresponds to each evaluation score.

Evaluation Score	Raise
3	5%
2	3%
1	0%

   Use an updatable cursor to process each employee record. If the employee has a score listed in the EVAL table, update the employee's hourly rate in the EMPLOYEE table and display some feedback regarding the update, such as:

```
Employee ID A1465 UPDATED
Evaluation score: 3
Raise: 5.00 %
Old rate: 11.18
New rate: 11.74
```

   Make sure to limit the updates to the HOURRATE column.

   If the employee does not have a record in the EVAL table, display a message similar to the following:

```
No evaluation for B6655
```

   Note: You may want to create a copy of the EMPLOYEE table before you test your solution. The following statement copies the data from the EMPLOYEE table into a new table named EMP3:

```
SELECT * INTO emp3 FROM employee
```

(continued on the following page)

Exercise 7
(continued)

2. Write a procedure that uses nested cursors to process weekly employee pay for every job type in the JOBS table. The report should display the job code, the weekly pay for each employee in that job, and a subtotal for the job code. For weekly pay calculations, assume that every employee worked 40 hours. At the end of the report, display a grand total for all employees. The first few groups are shown below.

```
                WEEKLY EMPLOYEE PAYROLL
-----------------------------------------------------

Job code: ACC
ID    Last Name       First Name      Job   Weekly Pay
-----------------------------------------------------
B9819 HILL            ANNA            ACC   $638.00

Total pay for ACC $638.00
-----------------------------------------------

Job code: CUR
ID    Last Name       First Name      Job   Weekly Pay
-----------------------------------------------------
B1158 JACKSON         RICHARD         CUR   $1140.00
B1263 RUDOLF          OLIVIA          CUR   $1094.00
B3544 RICE            JOSEPH          CUR   $946.00
B2611 HAND            AMELIA          CUR   $992.00

Total pay for CUR $4172.00
-----------------------------------------------
```

Handling Exceptions

While it's always important to code your programs carefully in order to avoid errors, it is nearly impossible to guarantee that your programs will never raise errors. Throughout this course, you've seen examples of SQL Server error messages, where the offending statement terminates, and you have to decipher the error message so you can find and correct the problem in your stored procedure. These errors are not only disruptive, but they can sometimes be difficult to interpret—even for experienced programmers. Using the exception-handling techniques introduced in this lesson, you can customize the messages displayed and the behavior exhibited when problems inevitably arise.

Using the TRY…CATCH Construct

Microsoft significantly improved SQL Server's exception-handling capabilities with the release of SQL Server 2005. One major improvement is the introduction of the **TRY…CATCH** construct, used to trap and handle errors.

The TRY…CATCH construct consists of two code blocks: a TRY block (beginning with a **BEGIN TRY** statement and ending with an **END TRY** statement) and a CATCH block (between the keywords **BEGIN CATCH** and **END CATCH**). When a statement in the TRY block generates an error, the error is "trapped," and control is passed to the CATCH block, which can include statements for handling the error. If no error is generated in the TRY block, the CATCH block is skipped.

```
BEGIN TRY
   Statements that could generate an error
END TRY

BEGIN CATCH
   Statements to handle error
END CATCH
```

As long as the error is trapped, no SQL Server error message will be returned to the user. Instead, you can use the CATCH block to handle the exception in a customized way.

Using the TRY…CATCH construct, you can trap almost any SQL Server-defined error (as long as the severity level is not greater than 19, indicating a severe error, like a hardware failure). This technique offers much more flexibility than error-handling techniques in previous versions of SQL Server.

Consider the following procedure, which attempts to insert a new record into the Jobs table. If the INSERT statement raises an exception, control will pass to the CATCH block, which returns the message "Insert failed" to the client.

```
CREATE PROCEDURE insert_error
AS
   BEGIN TRY
      INSERT INTO jobs VALUES ('PRO', 'PROGRAMMER', 'EXEMPT', 25, 40)
      PRINT 'Insert succeeded.'
   END TRY
   BEGIN CATCH
      PRINT 'Insert failed.'
   END CATCH
```

This procedure should run successfully the first time—as long as the Jobs table doesn't already include a record with the job code PRO. The second time you run the procedure, the INSERT will fail, since the primary key on the Jobcode column prevents you from inserting duplicate records.

Using Error-Handling Functions to Retrieve Error Information

In addition to the new TRY…CATCH construct, SQL Server 2005 introduced a new set of error-handling functions to return information about an error. You can use these functions to return the components of a standard SQL Server error message, such as the error number, severity level, state, and description of the error. You can also return the name of the procedure or trigger that generated the error, as well as the line number where the error occurred.

These functions are described in the following table:

Function	Description
ERROR_NUMBER()	Returns the error number associated with the error.
ERROR_MESSAGE()	Returns the default error message associated with the error.
ERROR_SEVERITY()	Returns the error's severity level.
ERROR_STATE()	Returns the state of the error.
ERROR_PROCEDURE()	Returns the name of the stored procedure or trigger in which the error occurred. (If the error didn't occur in a stored procedure or trigger, this function returns NULL.)
ERROR_LINE()	Returns the line number where the error occurred.

You can use these functions inside a CATCH block to return information about the error.

```
ALTER PROCEDURE insert_error
AS
   BEGIN TRY
      INSERT INTO jobs VALUES ('PRO', 'PROGRAMMER', 'EXEMPT', 25, 40)
      PRINT 'Insert succeeded.'
   END TRY
   BEGIN CATCH
      PRINT 'Insert failed.'
      PRINT 'Error occurred on line ' +
            CONVERT(varchar(3), ERROR_LINE())
            + ' of the ' + ERROR_PROCEDURE() + ' procedure.'
      PRINT 'Error number: ' + CONVERT(varchar(10), ERROR_NUMBER())
      PRINT 'Error message: ' + ERROR_MESSAGE()
      PRINT 'Severity level: ' +
            CONVERT(varchar(10), ERROR_SEVERITY())
      PRINT 'Error state: ' + CONVERT(varchar(10), ERROR_STATE())
   END CATCH
```

Now, if the insert fails, you will see this message:

```
Insert failed.
Error occurred on line 4 of the insert_error procedure.
Error number: 2627
Error message: Violation of PRIMARY KEY constraint 'PK__jobs__6OA75C0F'.
Cannot insert duplicate key in object 'dbo.jobs'.
Severity level: 14
Error state: 1
```

Displaying Customized Error Messages

In addition to using SQL Server's built-in error messages, you can use the **RAISERROR** command to return a customized error message back to the application. You can customize the following components of an error message:

- **Text:** The message to display to the user. You can specify the text message as part of the RAISERROR command, or you can store the message in the sysmessages table and call it by referring to its error number.

- **Error number:** You can use any number greater than 50,000 for user-defined error messages. (You only assign an error number when you store a custom error message in the sysmessages table, as explained below.)

- **Severity level:** The severity level indicates the seriousness of the error. Assign a severity level according to the following table:

Level	Description
0-10	Information only
11-16	User-correctible errors (such as syntax, security, and data validation)
17-19	Hardware or software problem
20-25	Fatal error

- **State:** Each condition that raises an error assigns a specific state code (so two errors with the same error number might have different states if they were raised for different reasons). The RAISERROR command can raise errors with a state from 1 through 127. You should use a different state for each RAISERROR command that references the same error. This can help you troubleshoot multiple errors with the same error number and message.

As you've seen, you can use the PRINT command to return an informational or warning message to the user based on some condition (for example, if a query returns no data). The RAISERROR command offers some advantages over this alternative. First, if you use the RAISERROR command in a TRY block to raise an error with a severity level of 11 to 19, control passes to the CATCH block, so you can handle the error in the same way you would handle a SQL Server-generated error.

Note: If the severity level is 10 or lower, control does not pass to the CATCH block. You can use these levels to simply return an informational message without handling it like an error.

Another advantage of the RAISERROR command is that you can use string substitution to build the error message. This is demonstrated in the section on *Using Parameter Variables in Your Error Messages*.

Issuing Ad Hoc Messages

You can issue an "ad hoc" message by specifying the error text in the RAISERROR command. This command follows the syntax:

```
RAISERROR('text message', severity_level, state)
```

You do not specify an error number in this command, since the message is not stored permanently on the server and cannot be called again in another procedure.

For example, the following command returns an informational message "No members found" with a severity level of 10 and a state of 1.

```
RAISERROR('No members found', 10, 1)
```

The **member_error** procedure uses this RAISERROR statement in the TRY block to return a message if the user supplies a member code that does not appear in the Members table.

```
CREATE PROCEDURE member_error
   @mcode char(2)= NULL
AS
   BEGIN TRY
      IF @mcode IN (SELECT memcode FROM members)
         SELECT * FROM members
         WHERE memcode = @mcode
      ELSE
         RAISERROR('No members found', 10, 1)
      PRINT 'Thank you.'
   END TRY
   BEGIN CATCH
      PRINT 'You raised a severe error.'
   END CATCH
```

Since the severity level is 10, control doesn't pass to the CATCH block when the error is raised. This indicates that the RAISERROR command issues an informational message only; the program does not have to stop.

If you want control to pass to the CATCH block, you need to raise an error with a severity level of 11 or more.

```
ALTER PROCEDURE member_error
   @mcode char(2)= NULL
AS
   BEGIN TRY
      IF @mcode IN (SELECT memcode FROM members)
         SELECT * FROM members
         WHERE memcode = @mcode
      ELSE
         RAISERROR('No members found', 11, 1)
      PRINT 'Thank you.'
   END TRY
   BEGIN CATCH
      PRINT 'You raised a severe error.'
   END CATCH
```

Now, if you execute this procedure with an invalid member code, the error will be trapped, and control will pass to the CATCH block. The following message will be displayed:

```
You raised a severe error.
```

If you want to display information about the error when control passes to the CATCH block, you can use the error functions described earlier to capture information about the error. You can then use PRINT statements to create a custom error message (as you did earlier), or you can add a RAISERROR statement to the CATCH block to display the error information in the same format as a typical SQL Server error message.

The CATCH block in the following procedure uses variables to hold the error message, severity level, and state. The RAISERROR command is used to return this error information to the client.

```
ALTER PROCEDURE member_error
   @mcode char(2)= NULL
AS
   BEGIN TRY
      IF @mcode IN (SELECT memcode FROM members)
         SELECT * FROM members
         WHERE memcode = @mcode
      ELSE
         RAISERROR('No members found', 11, 1)
      PRINT 'Thank you.'
   END TRY
   BEGIN CATCH
      DECLARE @err_msg varchar(1000), @err_sev int,
              @err_state int
      SET @err_msg = ERROR_MESSAGE()
      SET @err_sev = ERROR_SEVERITY()
      SET @err_state = ERROR_STATE()
      PRINT 'You raised a severe error.'
      RAISERROR(@err_msg, @err_sev, @err_state)
   END CATCH
```

Now, when you run the procedure with an invalid member code, you will see the following message:

```
You raised a severe error.
Msg 50000, Level 11, State 1, Procedure member_error, Line 19
No members found
```

Note that the error message appears in the same format as a typical SQL Server-generated message.

Storing Error Messages

If you want to use the same error message repeatedly, you can add the message to the **sysmessages** data table, using the **sp_addmessage** system procedure:

```
sp_addmessage error_number, severity_level, 'message'
```

For example, the following statement will add an error with the error number 50001, the severity level 11, and the message "No members found" to the **sysmessages** table:

```
EXEC sp_addmessage 50001, 11, 'No members found'
```

Now, you can call the error in the RAISERROR command by referring to the error number:

```
RAISERROR(error_number, severity_level, error_state)
```

The statement below raises error number 50001:

```
RAISERROR(50001, 11, 1)
```

Note: In order to add or drop messages in the sysmessages table, you need to be a member of the sysadmin or serveradmin fixed user role.

Using Parameter Variables in Your Error Messages

When you add error messages to the sysmessages table, you can use parameters as placeholders in the error message and then pass in a value when you raise the error. The following statement will add a new message to the sysmessages table. The **%s** in the error message indicates that the parameter will be replaced by a string value when the error is raised.

```
EXEC sp_addmessage 50002, 11, 'No members found in %s database'
```

When you raise the error, you pass in the parameter value as a fourth argument to the RAISERROR command, such as:

```
RAISERROR(50002, 11, 1, @dbname)
```

According to this statement, the value stored in the variable @dbname will replace the %s parameter in the error message.

The entire procedure is printed below. Note that the **DB_NAME()** function is used to return the current database name, which is assigned to the @dbname variable.

```
ALTER PROCEDURE member_error
   @mcode char(2)= NULL
AS
   DECLARE @dbname varchar(128)
   SET @dbname = DB_NAME()
   BEGIN TRY
      IF @mcode IN (SELECT memcode FROM members)
         SELECT * FROM members
         WHERE memcode = @mcode
      ELSE
         RAISERROR(50002, 11, 1, @dbname)
      PRINT 'Thank you.'
   END TRY
   BEGIN CATCH
      DECLARE @err_msg varchar(1000), @err_sev int,
              @err_state int
      SET @err_msg = ERROR_MESSAGE()
      SET @err_sev = ERROR_SEVERITY()
      SET @err_state = ERROR_STATE()
      PRINT 'You raised a severe error.'
      RAISERROR(@err_msg, @err_sev, @err_state, @dbname)
   END CATCH
```

Now, if you attempt to run the procedure with an invalid member code, you will see a message like the following (where prog1sqld29 is the name of the current database).

```
You raised a severe error.
Msg 50000, Level 11, State 1, Procedure member_error, Line 21
No members found in prog1sqld29 database
```

Dropping Stored Messages

To drop a message from the sysmessages table, use the **sp_dropmessage** system procedure:

```
sp_dropmessage error_number
```

such as:

```
sp_dropmessage 50001
```

Exercise 8: Handling Exceptions

In this exercise, you will practice using the TRY…CATCH construct and the RAISERROR command to trap and handle exceptions in your stored procedures.

This exercise refers to data in the JOBS table.

1. Open the **Ex8-error.sql** file in your **Exercises and Solutions** folder. This file includes code for a stored procedure that updates the minimum pay rate for a specific job code.

   ```
   CREATE PROCEDURE minrate_error
      @jcode char(3), @minpay numeric(5,2)
   AS
      UPDATE jobs
      SET minrate = @minpay
      WHERE jobcode = @jcode
   ```

   The MINRATE column in the JOBS table includes a CHECK constraint that enforces a minimum wage of $6.50 per hour. If you attempt to enter a minimum rate less than $6.50, the update will fail and the procedure will generate an error.

   Add TRY and CATCH blocks to the procedure to trap this error and print the following message:

   ```
   Update failed.
   Error number: 547
   Error message: The UPDATE statement conflicted with the CHECK
   constraint "CK__jobs1__minrate__66603565". The conflict occurred in
   database "prog1sqld29", table "dbo.jobs", column 'minrate'.
   ```

   Note: This error number and error message are defined by SQL Server. You can use the error functions to retrieve this information.

2. Modify the procedure again to raise an exception if the user supplies an invalid job code. Use the RAISERROR command to pass control to the CATCH block and display an appropriate error message.

Appendix A: Class Tables

The following tables are included in your class database.

Collections

Column	Data Type	Length	Precision	Scale
coll_id	char	5		
coll_desc	varchar	30		
num_items	int			
exhibit_id	int			
mem_id	char	5		
donate_date	smalldatetime			
coll_value	numeric		7	2

The following data is included in the Collections table (23 rows):

```
5PEZ0    CLASSIC PEZ DISPENSERS          20   1950   GR009    2006-10-20      95.00
5JUKE    1950S JUKEBOX                    1   1950   GR009    2006-11-01     500.00
5POOD    POODLE SKIRT                     2   1950   AH119    2007-02-26      50.00
5SHOE    SADDLE SHOES                     1   1950   AH119    2007-02-26      15.00
5TV00    VINTAGE TV                       1   1950   GR009    2006-11-01     125.00
6BEAT    BEATLES RECORDS                  6   1960   TC769    2004-05-30      65.00
6PHAT    PILLBOX HAT                      1   1960   ST546    2007-01-28       7.50
6WOOD    SIGNED WOODSTOCK POSTER          1   1960   MT907    2006-06-07     200.00
6SLOG    SLOGAN BUTTONS                  20   1960   MT907    2006-07-22      20.00
6TREK    ORIGINAL STARFLEET UNIFORM       1   1960   AA910    2006-04-05     250.00
7POST    BLACKLIGHT POSTERS              15   1970   MT907    2006-04-15     100.00
7SHOE    PLATFORM HEELS                   5   1970   JJ111    2003-12-20     150.00
7SUIT    LEISURE SUITS                    2   1970   JJ111    2003-12-20      40.00
7DISC    DISCO BALL                       1   1970   LS504    2003-01-02      50.00
8ATAR    ATARI GAME SYSTEM                1   1980   AR492    2005-12-15      60.00
8GAME    ATARI GAME CARTRIDGES           20   1980   AR492    2005-12-15     100.00
8BETA    BETA MAX PLAYER                  1   1980   EJ257    2006-11-15      25.00
8MOVI    BETA MAX MOVIES                 10   1980   EJ257    2006-11-15      50.00
8SMUR    SMURF FIGURINES                 30   1980   JJ111    2003-10-12      60.00
8CABB    CABBAGE PATCH DOLLS             10   1980   JJ111    2003-02-05     400.00
9POKE    POKEMON TRADING CARDS          100   1990   JJ891    2006-07-22     100.00
9BEAN    BEANIE BABIES                   20   1990   JJ891    2006-07-22     100.00
9ELMO    TICKLE ME ELMO DOLL              1   1990   PS107    2006-05-15      10.00
```

Employee

Column	Data Type	Length	Precision	Scale
empid	char	5		
firstname	varchar	15		
midname	varchar	15		
lastname	varchar	15		
jobcode	char	3		
hiredate	smalldatetime			
hourrate	numeric		5	2

The following data is included in the Employee table (17 rows):

```
A1465     ROBERTA     JEAN        PICKNEY     SEC     2004-06-14     10.65
B2559     BIFF        NULL        ARFUSS      DIR     2002-11-01     39.25
A5150     GINA        ANN         PEREZ       TRG     2003-04-05     11.25
A4115     JANETTE     E.          FLOWERS     SEC     2006-12-12     10.05
A6395     MICHAEL     F.          CHEN        TRG     2006-03-01      9.45
B1158     RICHARD     ALONSO      JACKSON     CUR     2002-11-01     28.50
B1263     OLIVIA      NULL        RUDOLF      CUR     2002-12-06     27.35
A1333     PATRICIA    K.          CARTER      TRG     2006-06-23      8.50
B3544     JOSEPH      AARON       RICE        CUR     2006-05-15     23.65
B2611     AMELIA      JANE        HAND        CUR     2005-04-26     24.80
A1568     RAY         P.          LEVY        TRG     2007-01-30      8.50
A5015     BETTY       JO          RICCI       TRG     2007-02-04      8.50
A7812     JUDY        N.          LEE         TRG     2005-06-25     10.25
B7610     BEVERLY     SUE         WATERS      DEV     2005-02-27     22.10
B6655     DANIEL      Z.          POPOV       DEV     2005-02-27     22.10
B3678     RUTH        A.          OLIVER      TRC     2005-08-06     18.75
B9819     ANNA        M.          HILL        ACC     2006-10-12     15.95
```

Eval

Column	Data Type	Length
empid	char	5
score	int	

The following data is included in the Eval table (12 rows):

```
A1465     3
A5150     2
A6395     2
B1158     3
B1263     1
B2611     2
A1568     2
A5015     1
A7812     2
B7610     3
B3678     1
B9819     2
```

Exhibits

Column	Data Type	Length
exhibit_id	int	
room	char	3
empid	char	5

The following data is included in the Exhibits table (5 rows):

```
1950      20A       B1158
1960      20B       B1263
1970      21        B3544
1980      22        B2611
1990      22        B3544
```

Jobs

Column	Data Type	Length	Precision	Scale	Constraint
jobcode	char	3			Primary Key
jobdesc	varchar	16			
class	varchar	10			Default (EXEMPT)
minrate	numeric		5	2	Check (minrate >= 6.5)
maxrate	numeric		5	2	

The following data is included in the Jobs table (7 rows):

```
ACC       ACCOUNTANT          EXEMPT          15.00     20.00
CUR       CURATOR             EXEMPT          20.25     35.00
DEV       DEVELOPMENT         EXEMPT          17.75     26.50
DIR       DIRECTOR            EXEMPT          35.25      NULL
SEC       SECRETARY           NON-EXEMPT       9.85     15.75
TRC       TOUR COORDINATOR    EXEMPT          17.50     26.50
TRG       TOUR GUIDE          NON-EXEMPT       8.50     12.00
```

Members

Column	Data Type	Length
mem_id	char	5
firstname	varchar	15
lastname	varchar	15
memcode	char	2
memdate	smalldatetime	

The following data is included in the Members table (20 rows):

```
AC135     ANNE        COOPER      ST      2004-07-05
RP090     RICHARD     POWERS      FM      2003-08-10
AA910     AARON       ATWOOD      AD      2006-03-27
CM818     CHRIS       MARKS       ST      2005-12-17
PS107     PENELOPE    STARR       CH      2003-04-06
GR009     GRANT       REDDY       SR      2006-10-11
AH119     ALICIA      HERNANDEZ   SR      2007-02-22
GG292     GREG        GREEN       AD      2005-01-01
MB760     MARIE       BURNS       CH      2006-04-30
LS504     LOU         SMALLS      ST      2002-12-14
JJ111     JENNY       JEFFRIES    ST      2002-12-14
TC769     TODD        CHANG       FM      2004-05-15
MT907     MIKE        THOMAS      AD      2006-03-09
ST546     STACY       THURMAN     FM      2005-05-25
RR288     RONALD      ROGAN       FM      2006-09-27
EJ257     EDWARD      JACKSON     AD      2006-09-30
AR492     AARON       ROLLINS     ST      2005-12-12
CB652     CHRIS       BECKER      ST      2005-12-05
JJ891     JULIE       JOSEPH      CH      2006-09-13
RJ533     ROGER       JACKSON     ST      2006-07-22
```

Memtype

Column	Data Type	Length	Precision	Scale
memcode	char	2		
mem_descrip	varchar	10		
dues	numeric		5	2

The following data is included in the Memtype table (5 rows):

```
CH      CHILD     39.95
ST      STUDENT   49.95
AD      ADULT     54.95
SR      SENIOR    44.95
FM      FAMILY    74.95
```

Appendix B: SQL Server Date Functions

SQL Server provides a variety of date functions that will help you analyze and manipulate date data in different ways. The following functions are available in SQL Server:

GETDATE()

Returns the current system date and time. For example, the following query returns the current date and time:

```
SELECT GETDATE()
```

DATEADD(*datepart*, *number*, *date*)

Adds date and time intervals (minutes, days, weeks, etc.) to a specified date. *datepart* specifies which part of the date to increment. *number* is an integer value used to increment *datepart*. The final argument, *date*, specifies the datetime (or smalldatetime) value on which you want to add the interval.

You can use the following words or abbreviations to specify the *datepart* parameter for this function:

Datepart	Abbreviations
year	yy, yyyy
quarter	qq, q
month	mm, m
dayofyear	dy, y
day	dd, d
week	wk, ww
weekday	dw, w
hour	hh
minute	mi, n
second	ss, s
millisecond	ms

Note: These date parts are also used to specify the *datepart* parameter in the other functions discussed below.

The following query uses the DATEADD function to display all columns from the Members table, along with a calculated column (Newdate) that adds two days to the membership date:

```
SELECT *, DATEADD(dd, 2, memdate) AS newdate
FROM members
```

DATEDIFF(*datepart*, *date1*, *date2*)

Returns the number of *datepart* intervals between two dates. *date1* is subtracted from *date2*, so if *date1* is later than *date2*, the function returns a negative value.

The query below uses the DATEDIFF function to calculate how long ago (in years) each employee was hired.

```
SELECT empid, hiredate, DATEDIFF(yy, hiredate, GETDATE()) AS yrs_employed
FROM employee
```

Note: This actually subtracts the year portion of the hire date from the year portion of the current date, so the following query would return 1, even though the dates are only one day apart (but in different years).

```
SELECT DATEDIFF(yy, '2006-12-31', '2007-1-1')
```

To return the number of full years an employee has worked for the company, along with a decimal that reflects the partial years, you can calculate the number of days between two dates and divide that result by the number of days in a year (365.25). This will return a decimal value representing the number of years:

```
SELECT empid, hiredate,
       DATEDIFF(dd, hiredate, GETDATE())/365.25 AS partial_years
FROM employee
```

DATEPART(*datepart*, *date*)

Returns an integer representing the *datepart* interval of the specified date (where 1=Sunday).

The following query returns integers representing the month and weekday for the membership dates in the Members table:

```
SELECT *, DATEPART(mm, memdate) AS mem_month,
       DATEPART(dw, memdate) AS mem_weekday
FROM members
```

DATENAME(*datepart*, *date*)

Returns a character string representing a portion (*datepart*) of *date*.

You could use the DATENAME function instead of DATEPART in the previous query to display the names of the months and weekdays (instead of integers).

```
SELECT *, DATENAME(mm, memdate) AS mem_month,
       DATENAME(dw, memdate) AS mem_weekday
FROM members
```

Converting Dates to Strings

When you use the CONVERT function to change date data to a string data type, you include a style parameter to specify how the converted date will be displayed.

```
CONVERT(datatype, expression, style)
```

The first parameter is a string data type indicating how many characters to display. (Make sure to include enough characters to accommodate the date format you choose.) The second parameter is a date expression—either the GETDATE function or a date field. The third parameter is a style, indicating how the date will be displayed.

The following table lists possible values for the *style* parameter, along with the corresponding display format for January 19, 2007 at 3:14 p.m.:

Style Value	Date Format	Formatted Date
1	mm/dd/yy	01/19/07
2	yy.mm.dd	07.01.19
3	dd/mm/yy	19/01/07
4	dd.mm.yy	19.01.07
5	dd-mm-yy	19-01-07
6	dd Mon yy	19 Jan 07
7	Mon dd, yy	Jan 19, 07
8	hh:mm:ss	15:14:30
9	Mon dd, yyyy h:mm:ss:msAM/PM	Jan 19 2007 3:14:30:093PM
10	mm-dd-yy	01-19-07
11	yy/mm/dd	07/01/19
12	yymmdd	070119
13	dd Mon yyyy hh:mm:ss:ms	19 Jan 2007 15:14:30:093
14	hh:mm:ss:ms	15:14:30:093

Note: You can add 100 to any style that displays a two-digit year to format the year as a four-digit year. For example, the style 101 would format the date as *mm*/*dd*/*yyyy*, such as 01/19/2007.

Appendix C: Solutions to Exercises

Solutions to Exercise 1

1. Write a short SQL procedure to compute parking charges. The procedure should accept the number of hours parked, calculate the charge at $.75 per hour, and display the result.

```
CREATE PROCEDURE parking
   @hours int = 0
AS
   DECLARE @park_total numeric(5,2)
   SET @park_total = .75 * @hours
   PRINT 'PLEASE PAY: $' + CONVERT(char(6), @park_total)
```

   The statement below executes this procedure to calculate the cost of parking for 10 hours:

```
EXEC parking 10
```

2. Write a program that calculates the total price of a purchase, including sales tax. The procedure should accept the price of the purchase (before tax) and calculate tax based on a rate of 7.5%. Display the original purchase price, amount of tax, and total price.

```
CREATE PROCEDURE tax
   @price numeric(7,2) = 0
AS
   DECLARE @tax_amt numeric(7,2), @total_price numeric(7,2)
   SET @tax_amt = @price * .075
   SET @total_price = @price + @tax_amt
   PRINT 'Original price: ' + CONVERT(varchar(10), @price)
   PRINT 'Tax: ' + CONVERT(varchar(10), @tax_amt)
   PRINT 'Total price: ' + CONVERT(varchar(10), @total_price)
```

   To test the procedure for a $10.50 purchase:

```
EXEC tax 10.5
```

(continued on the following page)

Solutions to Exercise 1
(continued)

3. Write a procedure to compute gas mileage. The procedure requires three arguments: the odometer reading when you started your trip, the reading when you finished the trip, and how many gallons of fuel were used. The computer should then process the information and display your miles per gallon.

```
CREATE PROCEDURE mileage
   @start numeric(7,1) = 0, @finish numeric(7,1) = 0,
   @gallons numeric(6,2) = 0
AS
   DECLARE @mpg numeric(3,1)
   SET @mpg = (@finish - @start) / @gallons
   PRINT 'Your gas mileage was: ' + CONVERT(char(10) , @mpg)
```

   The following statement executes this procedure, calculating the miles per gallon for a trip that started with 100 miles on the odometer, ended with 200 miles on the odometer, and used 5 gallons of gas.

```
EXEC mileage 100, 200, 5
```

4. Given the assumption that you sleep a healthy eight hours a night, calculate and display the number of hours of your life that you have spent sleeping. Supply the date of your birth and use the system date for today's date.

```
CREATE PROCEDURE sleep
   @bday smalldatetime = 'Jan 1, 1900'
AS
   DECLARE @hours int
   SET @hours = DATEDIFF(hh, @bday, GETDATE())
   PRINT 'You have slept ' +
      CONVERT(varchar(10) , @hours/3) + ' hours'
```

   The following statement tests this procedure for someone who was born on January 1, 1980:

```
EXEC sleep '1/1/80'
```

(continued on the following page)

Solutions to Exercise 1
(continued)

Extra: The Department of Transportation and the National Society of Beekeepers need your help. The "New York Special" train is leaving New York City, heading for Los Angeles at 40 miles per hour. At the same time, the "Los Angeles Express" train is leaving Los Angeles for New York at 60 miles per hour. These two trains are on the same track, 3,000 miles apart.

During the time the trains are traveling toward each other, a bee goes back and forth between the two locomotives. As soon as the bee touches the front of one engine, it immediately turns around and flies toward the other train. The bee travels at 100 miles per hour.

Your job is to write a procedure that determines the distance the bee travels.

```
CREATE PROCEDURE bee
AS
   DECLARE @nytrain int, @latrain int, @bee_mph int, @bee_distance int
   SET @nytrain = 40
   SET @latrain = 60
   SET @bee_mph = 100
   SET @bee_distance = 3000 / ( @nytrain + @latrain ) * @bee_mph
   PRINT 'That poor little bee travels a distance of ' +
      CONVERT(varchar(10), @bee_distance) + ' miles'
```

Solutions to Exercise 2

1. Using the EMPLOYEE table, write a stored procedure that accepts an employee ID as an argument and then displays the first name, last name, and job code for the employee.

```
CREATE PROCEDURE emp_select
   @emp_id char(5) = NULL
AS
   DECLARE @fname varchar(15), @lname varchar(15), @jcode char(3)
   SELECT @fname = firstname, @lname = lastname, @jcode = jobcode
   FROM   employee
   WHERE  empid = @emp_id
   PRINT 'Employee ID: ' + @emp_id
   PRINT 'Name: ' + @fname + ' ' + @lname
   PRINT 'Job code: ' + @jcode
```

   Challenge: Instead of displaying the job code (e.g., SEC), display the associated job title (e.g., SECRETARY), using data from JOBS table.

```
CREATE PROCEDURE emp_select_challenge
   @emp_id char(5) = NULL
AS
   DECLARE @fname varchar(15), @lname varchar(15), @jdesc varchar(16)
   SELECT @fname = firstname, @lname = lastname, @jdesc = jobdesc
   FROM   employee INNER JOIN jobs
   ON employee.jobcode = jobs.jobcode
   WHERE  empid = @emp_id
   PRINT 'Employee ID: ' + @emp_id
   PRINT 'Name: ' + @fname + ' ' + @lname
   PRINT 'Job description: ' + @jdesc
```

2. Create a stored procedure to insert a new job into the JOBS table. Include one parameter variable for each column in the JOBS table, so users can supply the job information when they execute the procedure. Specify default NULL values for all columns except the job code.

```
CREATE PROCEDURE job_insert
   @jcode char(3), @jdesc varchar(16) = NULL,
   @jclass varchar(10) = NULL, @jminrate numeric(5,2) = NULL,
   @jmaxrate numeric(5,2) = NULL
AS
   INSERT INTO jobs (jobcode, jobdesc, class, minrate, maxrate)
   VALUES (@jcode, @jdesc, @jclass, @jminrate, @jmaxrate)
```

   To test the procedure:

```
EXEC job_insert 'INT', 'INTERN', 'NON-EXEMPT', 6.75, 9.75
```

(continued on the following page)

Solutions to Exercise 2
(continued)

3. Write a procedure to update the maximum salaries in the JOBS table. Include two parameter variables: one to specify the percent increase for EXEMPT jobs and one to specify the percent increase for NON-EXEMPT jobs. The default increase for EXEMPT and NON-EXEMPT jobs is 0%. After the update, display all job codes and their new maximum rates.

```
CREATE PROCEDURE increase_maxrate
   @exempt_raise numeric(5,3) = 0, @nonexempt_raise numeric(5,3) = 0
AS
-- increase maxrate for exempt jobs
   UPDATE jobs
   SET maxrate = maxrate * (1 + @exempt_raise)
   WHERE class = 'EXEMPT'
-- increase maxrate for non-exempt jobs
   UPDATE jobs
   SET maxrate = maxrate * (1 + @nonexempt_raise)
   WHERE class = 'NON-EXEMPT'
-- display jobs and new max rates
   SELECT jobcode, maxrate FROM jobs
```

Run the procedure, increasing maximum rates for EXEMPT jobs by 5% and maximum rates for NON-EXEMPT jobs by 8%.

```
EXEC increase_maxrate .05, .08
```

Solutions to Exercise 3

1. Set up a procedure that requests the user's age and then determines whether the person is old enough to drive.

```
CREATE PROCEDURE drive
   @age int = 0
AS
IF @age >= 16
   PRINT 'You are old enough to drive a car.'
ELSE
   BEGIN
      PRINT 'You are only ' + CONVERT(varchar(3), @age)
      PRINT 'You must wait ' + CONVERT(varchar(3), 16-@age) +
           ' years to drive.'
   END
```

2. Modify the previous procedure to add a condition for 15-year-olds who qualify for a learner's permit.

```
ALTER PROCEDURE drive
   @age int = 0
AS
IF @age >= 16
   PRINT 'You are old enough to drive a car.'
ELSE
   IF @age = 15
      PRINT 'You can get a learner''s permit.'
   ELSE
      BEGIN
         PRINT 'You are only ' + CONVERT(varchar(3), @age)
         PRINT 'You must wait ' + CONVERT(varchar(3), 16-@age) +
              ' years to drive.'
      END
```

3. Write a procedure that calculates a raise percentage based on an employee's score on a performance review, according to the table in the exercise.

```
CREATE PROCEDURE emp_raise
   @rating int = 2
AS
   DECLARE @raise numeric(5,3)
   IF @rating = 3
      SET @raise = .05
   ELSE
      IF @rating = 2
         SET @raise = .03
```

(continued on the following page)

Solutions to Exercise 3
(continued)

```
      ELSE
         IF @rating = 1
            SET @raise = 0
   IF @raise IS NULL
      PRINT 'Enter 1, 2, or 3 as the rating.'
   ELSE
      PRINT 'Your raise is ' + CONVERT(varchar(5), (@raise * 100)) + '%'
```

Challenge: Modify the previous procedure to update a museum employee's hourly pay rate (in the EMPLOYEE table), based on his or her evaluation rating. You should accept input for the employee ID and raise percentage and update the record associated with that ID. When you're finished with the update, display the employee's hourly rate from the EMPLOYEE table.

```
ALTER PROCEDURE emp_raise
   @emp_id char(5) = NULL, @rating int = 2
AS
   DECLARE @raise numeric(5,3), @newrate numeric(5,2)
   IF @rating = 3
      SET @raise = .05
   ELSE
      IF @rating = 2
         SET @raise = .03
      ELSE
         IF @rating = 1
            SET @raise = 0
   IF @raise IS NULL
      BEGIN
         PRINT 'Enter 1, 2, or 3 as the rating.'
         PRINT 'No change to employee''s hourly rate'
      END
   ELSE
      BEGIN
         PRINT 'Your raise is ' + CONVERT(varchar(5), (@raise * 100))
             + '%'
         -- update employee's rate in Employee table
         UPDATE employee
         SET hourrate = hourrate * (1 + @raise)
         WHERE empid = @emp_id
         PRINT 'Update complete.'
      END
   -- get rate from Employee table
   SELECT @newrate = hourrate
   FROM employee
   WHERE empid = @emp_id
   -- display hourly rate
   PRINT 'Hourly rate is: ' + CONVERT(varchar(8), @newrate)
```

(continued on the following page)

Solutions to Exercise 3
(continued)

Extra: Write a procedure that determines overtime eligibility for museum employees. Define parameter variables to get user input for employee ID and hours worked. Then determine overtime eligibility for the employee whose ID you entered, based on the factors defined in the exercise.

```
CREATE PROCEDURE overtime
   @emp_id char(5) = NULL, @hours numeric(5,2) = 0
AS
   DECLARE @jclass varchar(10)
   -- get employee's job class and assign to variable
   SELECT @jclass = class
   FROM employee INNER JOIN jobs
   ON employee.jobcode = jobs.jobcode
   WHERE empid = @emp_id
   -- assign overtime eligibility based on job class
   IF @jclass = 'NON-EXEMPT' AND @hours > 40
      PRINT 'You are eligible for overtime pay.'
   ELSE
      PRINT 'You are not eligible for overtime pay.'
```

Challenge: Now modify the previous procedure to calculate and display the employee's weekly pay, using the pay rate from the HOURRATE column in the EMPLOYEE table.

```
ALTER PROCEDURE overtime
   @emp_id char(5) = NULL, @hours numeric(5,2) = 0
AS
   DECLARE @jclass varchar(10), @rate numeric(5,2),
           @weekpay numeric(7,2)
   -- get employee's job class and hourly rate
   -- assign to variables
   SELECT @jclass = class, @rate = hourrate
   FROM employee INNER JOIN jobs
   ON employee.jobcode = jobs.jobcode
   WHERE empid = @emp_id
   -- assign overtime eligibility and calculate weekly pay
   -- based on job class & hours
   IF @jclass = 'NON-EXEMPT'
      BEGIN
         IF @hours > 40
            BEGIN
               PRINT 'You are eligible for overtime pay.'
               SET @weekpay = (40 * @rate) +
                   ((@hours - 40) * (@rate * 1.5))
            END
```

(continued on the following page)

Solutions to Exercise 3
(continued)

```
      ELSE
            BEGIN
               PRINT 'You are not eligible for overtime pay.'
               SET @weekpay = @hours * @rate
            END
      END
   ELSE
      BEGIN
         PRINT 'You are not eligible for overtime pay.'
         SET @weekpay = 40 * @rate
      END
   PRINT 'Weekly pay: ' + CONVERT(varchar(8), @weekpay)
```

Solutions to Exercise 4

1. Rewrite the procedure in Exercise 3, question 3, using a simple CASE function.

```
CREATE PROCEDURE raise_case
   @rating int = 2
AS
DECLARE @raise numeric(5,3)
SET @raise =
   CASE @rating
      WHEN 3 THEN .05
      WHEN 2 THEN .03
      WHEN 1 THEN 0
   END
IF @raise IS NULL
   PRINT 'Enter 1, 2, or 3 as the rating.'
ELSE
   PRINT 'Your raise is ' + CONVERT(char(5), (@raise * 100)) + '%'
```

2. Write a stored procedure that uses the CASE function to assign a salary grade based on an employee's hourly wage, according to the table provided in the exercise.

```
CREATE PROCEDURE salary_grade
   @hour_rate numeric(5,2) = 0
AS
   DECLARE @sal_grade char(1)
   SET @sal_grade =
      CASE
         WHEN @hour_rate < 10 THEN 'A'
         WHEN @hour_rate < 20 THEN 'B'
         WHEN @hour_rate < 30 THEN 'C'
         ELSE 'D'
      END
   PRINT 'Hourly rate: $' + CONVERT(char(6), @hour_rate)
   PRINT 'Salary grade: ' + @sal_grade
```

(continued on the following page)

Solutions to Exercise 4
(continued)

Challenge: Modify the previous procedure to have the user input an employee ID (from the EMPLOYEE table) and calculate the salary grade based on that employee's hourly rate. Consider adding a conditional statement to validate the employee ID entered by the user (and exit the procedure if the user provides an invalid ID).

```
ALTER PROCEDURE salary_grade
   @emp_id char(5) = NULL
AS
   IF @emp_id NOT IN (SELECT empid FROM employee)
      BEGIN
         PRINT 'Please enter a valid employee ID.'
         RETURN
      END
   DECLARE @hour_rate numeric(5,2), @sal_grade char(1)
   -- get employee's hourly rate from Employee table
   SELECT @hour_rate = hourrate
   FROM employee
   WHERE empid = @emp_id
   -- assign salary grade based on rate
   SET @sal_grade =
      CASE
         WHEN @hour_rate < 10 THEN 'A'
         WHEN @hour_rate < 20 THEN 'B'
         WHEN @hour_rate < 30 THEN 'C'
         ELSE 'D'
      END
   PRINT 'Hourly rate: $' + CONVERT(char(6), @hour_rate)
   PRINT 'Salary grade: ' + @sal_grade
```

Solutions to Exercise 5

1. Write a procedure that uses a loop to display a name 10 times on the screen. Use a parameter variable to supply the name to the procedure.

```
CREATE PROCEDURE names_loop
   @my_name varchar(15) = NULL
AS
   DECLARE @ctr int
   SET @ctr = 1
   WHILE @ctr <= 10
      BEGIN
         PRINT @my_name
         SET @ctr = @ctr + 1
      END
```

2. Modify the procedure from question 1 to display a number next to each line, incrementing the number each time the loop runs. Add a condition to exit the procedure after 5 iterations if the name is 5 letters long.

```
ALTER PROCEDURE names_loop
   @my_name varchar(15) = NULL
AS
   DECLARE @ctr int
   SET @ctr = 1
   WHILE @ctr <= 10
      BEGIN
         PRINT CONVERT(char(2), @ctr) + ' ' + @my_name
         IF @ctr = 5 AND LEN(@my_name) = 5
            BREAK
         SET @ctr = @ctr + 1
      END
```

3. Write a procedure that starts with the number 2 and doubles the number with each iteration until you reach 32. End the loop after you reach the number 32; then display the number of loop iterations.

```
CREATE PROCEDURE loop_32
AS
   DECLARE @num int, @ctr int
   SET @num = 2
   SET @ctr = 0
   WHILE @num <= 32
      BEGIN
         PRINT CONVERT(char(2), @num)
         SET @num = @num + @num
         SET @ctr = @ctr + 1
      END
   PRINT 'The loop executed ' + CONVERT(char(2), @ctr) + ' times.'
```

(continued on the following page)

Solutions to Exercise 5
(continued)

Extra: The budget for your tour guides has expanded, so you've decided to give all your tour guides raises. Using a loop, give each tour guide a 5% raise until the average pay for tour guides equals or exceeds $11 per hour. Display the current average tour guide rate at the beginning of the procedure, and recalculate and display the average rate after each iteration of the loop.

```
CREATE PROCEDURE trg_raise_loop
AS
   DECLARE @avg_rate numeric(7,2)
   -- get current average rate
   SELECT @avg_rate = AVG(hourrate)
   FROM employee
   WHERE jobcode = 'TRG'
   PRINT 'Initial average rate: $' + CONVERT(char(10), @avg_rate)
   -- use loop to assign 5% raises until average >= $11/hour
   WHILE @avg_rate < 11
      BEGIN
         UPDATE employee
         SET hourrate = hourrate * 1.05
         WHERE jobcode = 'TRG'
         -- calculate new average
         SELECT @avg_rate = AVG(hourrate)
         FROM employee
         WHERE jobcode = 'TRG'
         -- print new average
         PRINT 'New average rate: ' + CONVERT(char(10), @avg_rate)
      END
```

Challenge: Modify the procedure in the Extra question to include a condition to exit the loop when any one tour guide exceeds the maximum rate per hour for tour guides. (The maximum rate is included in the JOBS table.)

The following solution operates on a copy of the Employee table, named **Emp2**.

```
CREATE PROCEDURE trg_raise_loop_challenge
AS
   DECLARE @avg_rate numeric(7,2), @max_emp_rate numeric(7,2),
           @max_possible numeric(7,2)
   -- get current average and max rates
   SELECT @avg_rate = AVG(hourrate), @max_emp_rate = MAX(hourrate)
   FROM emp2
   WHERE jobcode = 'TRG'
   PRINT 'Initial average rate: $' + CONVERT(char(10), @avg_rate)
```

(continued on the following page)

Solutions to Exercise 5
(continued)

```
-- get max possible from jobs table
   SELECT @max_possible = maxrate
   FROM jobs
   WHERE jobcode = 'TRG'
   -- use loop to assign 5% raises until average >= $11/hour
   WHILE @avg_rate < 11
      BEGIN
         UPDATE emp2
         SET hourrate = hourrate * 1.05
         WHERE jobcode = 'TRG'
         -- calculate new average and max
         SELECT @avg_rate = AVG(hourrate), @max_emp_rate = MAX(hourrate)
         FROM emp2
         WHERE jobcode = 'TRG'
         -- print new average
         PRINT 'New average rate: ' + CONVERT(char(10), @avg_rate)
         -- exit if employee exceeds max possible
         IF @max_emp_rate > @max_possible
            BEGIN
               PRINT 'Employee exceeded max rate.'
               BREAK
            END
      END
```

Solutions to Exercise 6

1. Write a procedure that selects all tour guides from the EMPLOYEE table. Use a basic loop to fetch the data into variables and print each tour guide's first name, last name, and hourly rate. End the loop after all rows have been processed.

```
CREATE PROCEDURE trg_cursor
AS
   DECLARE @fname varchar(15), @lname varchar(15), @pay numeric(5,2)
   DECLARE cur_trg CURSOR FOR
      SELECT firstname, lastname, hourrate
      FROM employee
      WHERE jobcode = 'TRG'
   OPEN cur_trg
   WHILE 1=1
      BEGIN
         FETCH NEXT FROM cur_trg INTO @fname, @lname, @pay
         IF @@FETCH_STATUS <> 0
            BREAK
         PRINT 'Employee name: ' + @fname + ' ' + @lname
         PRINT 'Hourly rate: $' + CONVERT(char(8), @pay)
         PRINT ''
      END
   CLOSE cur_trg
   DEALLOCATE cur_trg
```

2. Modify the procedure so that the user can enter a job code when he or she runs the procedure. The procedure will then process the employees who have that code.

```
ALTER PROCEDURE trg_cursor
   @jcode char(3) = NULL
AS
   DECLARE @fname varchar(15), @lname varchar(15), @pay numeric(5,2)
   DECLARE cur_trg CURSOR FOR
      SELECT firstname, lastname, hourrate
      FROM employee
      WHERE jobcode = @jcode
   OPEN cur_trg
   WHILE 1=1
      BEGIN
         FETCH NEXT FROM cur_trg INTO @fname, @lname, @pay
         IF @@FETCH_STATUS <> 0
            BREAK
         PRINT 'Employee name: ' + @fname + ' ' + @lname
         PRINT 'Hourly rate: $' + CONVERT(char(8), @pay)
         PRINT ''
      END
   CLOSE cur_trg
   DEALLOCATE cur_trg
```

(continued on the following page)

Solutions to Exercise 6
(continued)

3. Write a procedure to print a weekly payroll report for non-exempt employees (secretaries and tour guides). Display each non-exempt employee's ID, first name, last name, job code, and weekly pay, assuming that every employee worked 40 hours during the week. Additionally, include the total weekly pay for secretaries and tour guides, as well as a weekly grand total for all non-exempt employees.

```
CREATE PROCEDURE nonexempt_pay_cursor
AS
   DECLARE @id char(5), @fname char(15), @lname char(15),
      @jcode char(3), @weekpay numeric(7,2),
      @totalpay numeric(9,2), @sec_pay numeric(9,2),
      @trg_pay numeric(9,2)
   SET @totalpay = 0
   SET @sec_pay = 0
   SET @trg_pay = 0
   DECLARE cur_emppay CURSOR FOR
      SELECT empid, firstname, lastname, employee.jobcode, hourrate*40
      FROM employee inner join jobs
      ON employee.jobcode = jobs.jobcode
      WHERE class = 'NON-EXEMPT'
   PRINT '               WEEKLY EMPLOYEE PAYROLL                '
   PRINT '------------------------------------------------------'
   PRINT ''
   PRINT 'ID    Last Name       First Name      Job   Weekly Pay'
   OPEN cur_emppay
   FETCH cur_emppay INTO @id, @fname, @lname, @jcode, @weekpay
   WHILE @@FETCH_STATUS = 0
      BEGIN
         IF @jcode = 'SEC'
            SET @sec_pay = @sec_pay + @weekpay
         ELSE IF @jcode = 'TRG'
            SET @trg_pay = @trg_pay + @weekpay
         SET @totalpay = @totalpay + @weekpay
         PRINT @id + ' ' + @lname + ' ' + @fname + ' ' + @jcode
              + '   $' + CONVERT(char(10), @weekpay)
         FETCH cur_emppay INTO @id, @fname, @lname, @jcode, @weekpay
      END
   PRINT ' '
   PRINT 'Total pay for secretaries: $' + CONVERT(char(12), @sec_pay)
   PRINT 'Total pay for tour guides: $' + CONVERT(char(12), @trg_pay)
   PRINT ' '
   PRINT 'Total pay for all non-exempt employees: $' +
         CONVERT(char(12), @totalpay)
   CLOSE cur_emppay
   DEALLOCATE cur_emppay
```

Solutions to Exercise 7

1. Write a procedure that updates employees' hourly rates in the EMPLOYEE table based on the evaluation scores in the EVAL table, as explained in the exercise.

   Note that this solution modifies a copy of the EMPLOYEE table, named EMP3. Also note the LEFT OUTER JOIN between the employee and evaluation tables. This ensures that all employees are included in the result set, even if they don't have an evaluation score.

```
CREATE PROCEDURE eval_raise
AS
DECLARE @eval_score int, @id char(5), @old_rate numeric(7,2),
        @new_rate numeric(7,2), @pct_raise numeric(5,2)
DECLARE cur_raise CURSOR
   FOR SELECT emp3.empid, hourrate, score
   FROM emp3 LEFT OUTER JOIN eval
   ON emp3.empid = eval.empid
   FOR UPDATE OF hourrate
OPEN cur_raise
FETCH NEXT FROM cur_raise INTO @id, @old_rate, @eval_score
WHILE @@FETCH_STATUS = 0
   BEGIN
      SET @pct_raise =
         CASE @eval_score
            WHEN 1 THEN 0
            WHEN 2 THEN .03
            WHEN 3 THEN .05
            ELSE NULL
         END
      IF @pct_raise IS NOT NULL
         BEGIN
            SET @new_rate = (@pct_raise * @old_rate) + @old_rate
            UPDATE emp3
            SET hourrate = @new_rate
            WHERE CURRENT OF cur_raise
            PRINT 'Employee ID ' + @id + ' UPDATED'
            PRINT 'Evaluation score: ' + CONVERT(char(1), @eval_score)
            PRINT 'Raise: ' + CONVERT(char(5), (@pct_raise * 100)) + '%'
            PRINT 'Old rate: ' + CONVERT(char(6), @old_rate)
            PRINT 'New rate: ' + CONVERT(char(6), @new_rate)
            PRINT ''
         END
      ELSE
         BEGIN
            PRINT 'No evaluation for ' + @id
            PRINT ''
         END
      FETCH NEXT FROM cur_raise INTO @id, @old_rate, @eval_score
   END
   CLOSE cur_raise
   DEALLOCATE cur_raise
```

(continued on the following page)

Solutions to Exercise 7
(continued)

2. Write a procedure that uses nested cursors to process weekly employee pay for every job type in the JOBS table.

```
CREATE PROCEDURE nested_emp_pay
AS
DECLARE @id char(5), @fname char(15), @lname char(15),
      @jobs_jcode char(3), @emp_jcode char(3), @weekpay numeric(7,2),
      @subtotal_pay numeric(9,2), @totalpay numeric(9,2)
DECLARE cur_jobs CURSOR FOR
   SELECT jobcode FROM jobs
PRINT '               WEEKLY EMPLOYEE PAYROLL               '
PRINT '-----------------------------------------------------'
OPEN cur_jobs
FETCH NEXT FROM cur_jobs INTO @jobs_jcode
WHILE @@FETCH_STATUS = 0
   BEGIN
      PRINT ''
      PRINT 'Job code: ' + @jobs_jcode
      SET @subtotal_pay = 0
      DECLARE cur_emppay CURSOR FOR
         SELECT empid, firstname, lastname, employee.jobcode,
               hourrate*40
         FROM employee
         WHERE jobcode = @jobs_jcode
      PRINT 'ID    Last Name       First Name      Job   Weekly Pay'
      PRINT '-----------------------------------------------------'
      OPEN cur_emppay
      FETCH cur_emppay INTO @id, @fname, @lname, @emp_jcode, @weekpay
      WHILE @@FETCH_STATUS = 0
         BEGIN
            PRINT @id + ' ' + @lname + ' ' + @fname + ' ' +
               @emp_jcode + '   $' + CONVERT(char(10), @weekpay)
            SET @subtotal_pay = @subtotal_pay + @weekpay
            FETCH cur_emppay INTO @id, @fname, @lname, @emp_jcode,
                  @weekpay
         END
      PRINT ''
      PRINT 'Total pay for ' + @jobs_jcode + ' $' +
            CONVERT(char(12), @subtotal_pay)
      PRINT '-----------------------------------------------'
      SET @totalpay = @totalpay + @subtotal_pay
      CLOSE cur_emppay
      DEALLOCATE cur_emppay
      FETCH NEXT FROM cur_jobs INTO @jobs_jcode
   END
PRINT ' '
PRINT 'Total pay for all employees: $' + CONVERT(char(12), @totalpay)
CLOSE cur_jobs
DEALLOCATE cur_jobs
```

Solutions to Exercise 8

1. Modify the procedure in the **Ex8-error.sql** file by adding TRY and CATCH blocks to the procedure to trap a violation of the check constraint on the MINRATE column.

```
CREATE PROCEDURE minrate_error
   @jcode char(3), @minpay numeric(5,2)
AS
   BEGIN TRY
      UPDATE jobs
      SET minrate = @minpay
      WHERE jobcode = @jcode
   END TRY
   BEGIN CATCH
      PRINT 'Update failed.'
      PRINT 'Error number: ' + CONVERT(varchar(10), ERROR_NUMBER())
      PRINT 'Error message: ' + ERROR_MESSAGE()
   END CATCH
```

2. Modify the procedure again to raise an exception if the user supplies an invalid job code. Use the RAISERROR command to pass control to the CATCH block and display an appropriate error message.

```
ALTER PROCEDURE minrate_error
   @jcode char(3), @minpay numeric(5,2)
AS
   BEGIN TRY
      IF @jcode IN (SELECT jobcode FROM jobs)
         UPDATE jobs
         SET minrate = @minpay
         WHERE jobcode = @jcode
      ELSE
         RAISERROR('Invalid job code', 11, 1)
   END TRY
   BEGIN CATCH
      PRINT 'Update failed.'
      PRINT 'Error message: ' + ERROR_MESSAGE()
   END CATCH
```