Programming with Transact-SQL

Objectives

- Learn about the origins and uses of Transact-SQL.
- Understand how to work with data types in Transact-SQL.
- Explore built-in functions for working with nulls, numbers, strings, dates, and system information.
- Use control-of-flow statements such as IF, CASE, and GOTO.
- Learn to set up a WHILE loop.
- Use error handling with @@ERROR and RAISERROR.

Overview of Transact-SQL

Transact-SQL is the SQL Server implementation of SQL-92, a standard codified by the American National Standards Institute (ANSI) and also adopted by the International Organization for Standardization (ISO). No single vendor has fully implemented every part of the ANSI/ISO standard, and each vendor has added its own proprietary extensions to the language, so you'll find plenty of things in Transact-SQL that you won't find in other database products.

The SQL language came about as a result of the work that Dr. E. F. Codd did back in the sixties on his Relational Database Model. The first version of the language was known as SEQUEL. It was then completely rewritten in the seventies, and eventually became known as SQL, because it turned out that the acronym SEQUEL had already been trademarked. The original "sequel" pronunciation has stuck to this day—SQL Server is still widely referred to as *sequel* server, although some purists insist that the language name should be pronounced ess-que-ell. However you pronounce it, SQL-92 has been relatively well received and is the most widely supported standard today.

Transact-SQL Extensions

Transact-SQL includes some very useful extensions to the SQL-92 standard that add procedural capabilities, making Transact-SQL more like a programming language. There are control-of-flow features, such as IF-ELSE syntax and WHILE loops, as well as support for variables, parameters, and user-defined functions. Like other programming languages, Transact-SQL also supports built-in functions for manipulating strings, numbers, date/time information, and returning system information.

Although Transact-SQL has programming language features, you'd never want to use Transact-SQL to replace a programming language. There is no user interface, and its programming constructs are very limited. The main advantage to programming in Transact-SQL is that your routines execute on the server. Transact-SQL provides the building blocks for all your views, stored procedures, user-defined functions, and triggers. Performing as much processing as possible in Transact-SQL improves performance because less data has to traverse the network for processing on the client.

You can break down Transact-SQL into two main categories:

- **Data Definition Language (DDL),** lets you create and modify objects in your database. The main commands used are CREATE, ALTER, and DROP. The SQL Server Enterprise Manager uses DDL behind the scenes to create and modify objects.

- **Data Manipulation Language (DML),** lets you work with your data. SELECT, INSERT, UPDATE, DELETE, and TRUNCATE are all part of DML.

A group of Transact-SQL statements can be organized into batches.

Batches

A batch is a collection of SQL statements that are submitted to SQL Server to be processed as a single unit. SQL Server compiles a single execution plan for each batch, where all the optimized steps needed to perform the statements are built into the plan. If there is a compile error in one of the statements, none of the statements in the batch will be executed.

You can use the Query Analyzer to create ad hoc batches by separating each batch with the GO statement. The GO statement isn't actually a part of the Transact-SQL language itself. It causes any statements preceding it to be treated as a single batch, and separates those statements from any statements following it.

See TSQL.sql

The following example shows two SQL DDL statements creating a table and a view, which are required to be processed as separate batches. If you attempt to process them as a single unit, you will receive a compile error:

```
CREATE TABLE tblTest
(
  ID int NULL,
  TestName varchar(50) NOT NULL
)
GO

CREATE VIEW vwTest
AS
   SELECT * FROM tblTest
GO
```

Variables

Transact-SQL variables and parameters work the same way in Transact-SQL as they do in any programming language. They must be declared with a data type (this is not optional) by using the DECLARE keyword. Local variables are always preceded with the @ symbol. You can use either SET or SELECT to assign a value to a variable:

```
DECLARE @local varchar(12)
SET @local = 'Local Phone: '
--SELECT @local = 'Local Phone: '

SELECT LastName, FirstName, @local + HomePhone AS Phone
FROM tblEmployee
ORDER BY LastName, FirstName
```

Delimiters and Operators

Transact-SQL has its own specific delimiters and operators, which may vary slightly from other database languages you might be used to (such as Microsoft Access). Table 1 lists the Transact-SQL operators and delimiters with a brief description of each.

Delimiter/Operator	Description
'	String and Date delimiter
+	Addition and Concatenation operator
-	Subtraction
*	Multiplication
/	Division
=	Equals
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<> or !=	Not equal to (!= is non-SQL-92 standard)
!<	Not less than (non-SQL-92 standard)
!>	Not greater than (non-SQL-92 standard)
_	Single-character wildcard
%	Modulo operator and Multiple-character wildcard
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
ALL	TRUE if all of a set of comparisons are TRUE
AND	TRUE if both Boolean expressions are TRUE
ANY, SOME	TRUE if any of a set of comparisons are TRUE
BETWEEN	TRUE if the operand is within a range
EXISTS	TRUE if a subquery contains any rows
IN	TRUE if the operand is equal to one of a list of expressions
LIKE	TRUE if the operand matches a pattern
NOT	Reverses the value of any other Boolean operator
OR	TRUE if either Boolean expression is TRUE

Table 1. Transact-SQL operators and delimiters.

Transact-SQL and Data Types

Many programming languages, such as VBA, are quite forgiving about implicitly converting data types in expressions and computations. For example, the following code snippet from VBA is perfectly legal, since you are allowed to concatenate a string and a number:

```
strMsg = "The result is: " & (2+2)
```

In the above case, the act of concatenating the product of the expression (2+2) to a string causes VBA to perform an implicit data type conversion of the integer to a string. In Transact-SQL, you must explicitly perform data type conversions yourself, using CAST or CONVERT.

CAST and CONVERT

The CAST and CONVERT functions are similar, but have slightly different syntax, with CAST being the ANSI synonym for CONVERT. Here's the syntax for each one:

```
CAST (expression AS data_type)
```

```
CONVERT (data_type [(length)], expression [, style])
```

The following listing shows using CAST and CONVERT to perform explicit data type conversion:

```
DECLARE @msg varchar(20)
SELECT @msg = 'The result is: ' +
  CAST((2+2) AS varchar)
PRINT @msg

DECLARE @msg varchar(20)
SELECT @msg = 'The result is: ' +
  CONVERT(varchar, (2+2))
PRINT @msg
```

NOTE The PRINT statement, like the GO statement, is specific to the Query Analyzer and is not part of the Transact-SQL language. It is used mainly for debugging purposes, not for printing to a printer or returning information to a client application.

The CONVERT function provides options to format the expression in its optional *style* argument. This is most useful for formatting datetime columns. The following examples use a style argument of 1 and 101 respectively:

```
SELECT CONVERT(varchar(12), GETDATE(), 1)
SELECT CONVERT(varchar(12), GETDATE(), 101)
```

The style 1 formats the year as two-digit, the style 101 formats the year as four-digit:

```
10/04/00
10/04/2000
```

The complete style argument list is found in Books Online under the *CAST and CONVERT* topic.

STR

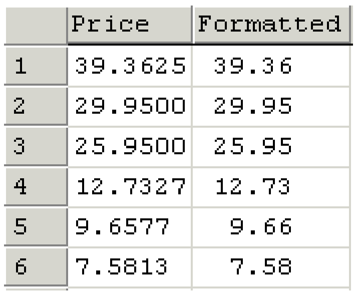
The STR function returns a string from a numeric expression:

```
STR(float_expression[, length[, decimal]])
```

The optional length and decimal parameters offer more flexibility than CAST or CONVERT when converting decimal data to character data, giving you explicit control over formatting. The following query uses STR to create a character string six characters long (with padding at the beginning of the string), and two decimal places:

```
SELECT Price, STR(Price, 6, 2) AS Formatted
FROM tblProduct
ORDER BY Price DESC
```

Figure 1 shows the first few rows of the output.



	Price	Formatted
1	39.3625	39.36
2	29.9500	29.95
3	25.9500	25.95
4	12.7327	12.73
5	9.6577	9.66
6	7.5813	7.58

Figure 1. Using STR to convert a number to a string.

Using Built-in Functions

This chapter isn't going to cover all of the functions available in Transact-SQL—you can find a complete listing in Books Online by searching for the topic *Functions* under the Transact-SQL subset. Transact-SQL functions fall under the following main categories as listed in Table 2.

Category	**Description**
Configuration	Returns information about the current configuration.
Cursor	Returns information about cursors.
Date and Time	Performs an operation on a date and time input value and returns either a string, numeric, or date and time value.
Mathematical	Performs a calculation based on input values provided as parameters to the function, and returns a numeric value.
Metadata	Returns information about the database and database objects.
Security	Returns information about users and roles.
String	Performs an operation on a string (char or varchar) input value and returns a string or numeric value.
System	Performs operations and returns information about values, objects, and settings in SQL Server.
System Statistical	Returns statistical information about the system.
Text and Image	Performs an operation on a text or image input values, and returns information about the value.

Table 2. Transact-SQL built-in functions.

The following sections explore some of the more commonly used functions.

Working with Nulls

Working with null values is a common problem when you allow columns to contain nulls. Nulls in aggregates are ignored, but this isn't necessarily always the behavior you want.

IS NULL

To test for a null value in a column, use IS NULL. This is a good habit to get into because using the equality operator, =, doesn't always work as expected—it depends on your ANSI NULL settings, which can be set differently by a connection and may differ from the database defaults. The following statement selects all of the employees who do not have a phone number listed (HomePhone is null):

```
SELECT LastName, HomePhone
FROM tblEmployee
WHERE HomePhone IS NULL
```

ISNULL

The ISNULL function allows you to replace a null with another value. Here's the same query with the null values replaced with the literal string, 'No Phone':

```
SELECT LastName, ISNULL(HomePhone, 'No Phone') AS Phone
FROM tblEmployee
WHERE HomePhone IS NULL
```

NULLIF

The NULLIF system function is just the opposite of ISNULL—it replaces actual values with nulls. This is useful for eliminating values from aggregate functions. For example, the following query computes an average price by not considering zeros:

```
SELECT AVG(NULLIF(Price, 0)) FROM tblproduct
```

You can test this by adding a new product with a price of zero and comparing the results with a query that computes the average without NULLIF.

COALESCE

The COALESCE function is used to return the first non-null expression in a series of expressions. The logic goes something like this: If expression1 is not null, then return expression1. If expression1 is null and expression2 is not null, then return expression2, and so on:

```
COALESCE(expression1, expression2 [,...n])
```

Use COALESCE when you have data stored in any one of a set of fields. For example, an Employees table has different columns for hourly wages, annual salaries, and commissions, to accommodate different categories of employees.

Each employee record will contain a value in only one of those three columns. The following expression could be used to compute employee compensation:

```
COALESCE(HourlyWage * 40 * 52,
         Salary,
         Commission * AnnualSales)
```

Functions and Numbers

Using the mathematical operators with numeric data is useful for basic calculations. You are probably already familiar with standard SQL aggregate functions like SUM and AVG. There are also functions for dealing with numbers and functions that return information about numbers and work with numbers.

IsNumeric

The IsNumeric function works just like its VBA equivalent, returning 1 if a value is numeric, and 0 if it can't. The following query uses IsNumeric on the ZipCode column of the tblCustomer table. Any null values or values that have non-numeric characters will return zero:

```
SELECT ISNUMERIC(ZipCode), ZipCode
FROM tblCustomer
```

RAND

RAND is used to generate random numbers with a value between 0 and 1. An optional seed parameter gives Transact-SQL a starting point for generating the resulting number, otherwise you'll just get the same number. If you use the same seed multiple times, you'll get the same number each time. The following example bases the random number on the current date/time, which will give a more random result than many seeds:

```
SELECT RAND(
  (DATEPART(mm, GETDATE()) * 100000 )
  + (DATEPART(ss, GETDATE()) * 1000 )
  + DATEPART(ms, GETDATE())
  )
```

ROUND

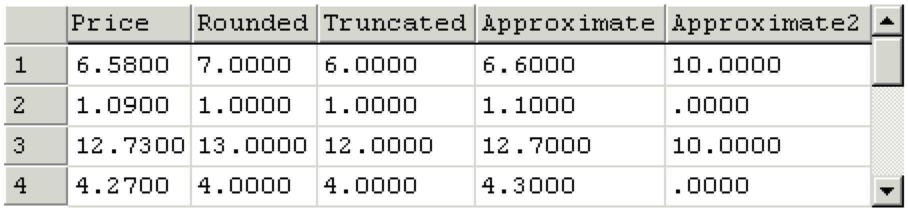
The ROUND function is used to round an expression to a specified length. The optional function argument is used to specify the type of operation. If you specify 0 (the default), the expression is rounded. Any other value causes the expression to be truncated.

```
ROUND(expression, length[, function])
```

The following example shows some of the variations you can use when using the Transact-SQL ROUND function:

```
SELECT Price,
  ROUND(Price, 0) AS Rounded,
  ROUND(Price, 0, 1) AS Truncated,
  ROUND(Price, 1) AS Approximate,
  ROUND(Price, -1) AS Approximate2
FROM tblProduct
```

Figure 2 shows the first few rows of the result set.



	Price	Rounded	Truncated	Approximate	Approximate2
1	6.5800	7.0000	6.0000	6.6000	10.0000
2	1.0900	1.0000	1.0000	1.1000	.0000
3	12.7300	13.0000	12.0000	12.7000	10.0000
4	4.2700	4.0000	4.0000	4.3000	.0000

Figure 2. The result set from using the ROUND function.

String Functions

String functions exist in all programming languages for the purpose of manipulating strings. This section will cover using some of the most common ones.

REPLACE and STUFF

REPLACE replaces all occurrences of a given string with another. The following SELECT will replace any occurrences of the string “Ten” with the string “10”:

```
SELECT Units, REPLACE(Units, 'Ten', '10')
FROM tblUnit
```

STUFF also replaces one string with another, but based on the starting position and length of the string, as shown in the following syntax:

```
STUFF (expr1, start, length, expr2)
```

The following example performs surgery on the string ‘12345’, starting at the 3rd character, excising 2 characters, and inserting the string ‘xxxx’ in place of those two characters:

```
SELECT STUFF('123456', 3, 2, 'xxxx')
```

Here are the results:

```
12xxxx56
```

LEN, LEFT, and RIGHT

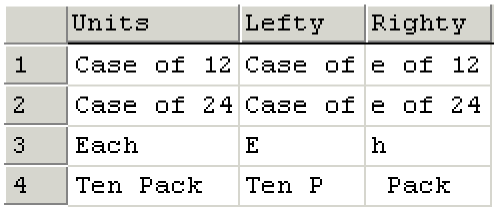
The LEN function returns the length of a string, the LEFT function returns the leftmost number of characters from a string, and the RIGHT function returns the rightmost number of characters from a string. Here’s the syntax for all three:

```
LEN (string_expression)
LEFT (string_expression, integer_numchars)
RIGHT (string_expression, integer_numchars)
```

The following example uses LEN to get the length of the Units in tblUnits, then subtracts three characters from the left, then three characters from the right:

```
SELECT Units,
  LEFT(Units, LEN(Units) -3) AS Lefty,
  RIGHT(Units, LEN(Units) -3) AS Righty
FROM tblUnit
```

Figure 3 shows the results.



	Units	Lefty	Righty
1	Case of 12	Case of	e of 12
2	Case of 24	Case of	e of 24
3	Each	E	h
4	Ten Pack	Ten P	Pack

Figure 3. Using the LEN, LEFT, and RIGHT functions.

SUBSTRING

The SUBSTRING function returns part of a string from a starting point to a specified length, or ending point. The following example uses the SUBSTRING function on the FirstName column, starting at the first character and returning one character. This is then concatenated with appropriate punctuation and the LastName column:

```
SELECT SUBSTRING(FirstName, 1, 1) + '. ' + LastName
FROM tblEmployee
```

Here's what the results look like:

```
M. Jones
S. Krash
R. Lovely
```

In that example, because only the leftmost character was being extracted, the LEFT function would also have worked. If you need to extract characters from the middle of a string, then SUBSTRING is your only choice.

CHARINDEX

Use CHARINDEX to find the starting position of an expression in a character string. The following example returns the starting position of the word “doll” in the Product column. If “doll” isn’t found, then CHARINDEX returns zero.

```
SELECT Product, CHARINDEX('doll', Product) AS Start
FROM tblProduct
```

SPACE

Use the SPACE function to add spaces, as in the following example where three spaces are concatenated between the FirstName and LastName columns.

```
SELECT FirstName + SPACE(3) + LastName
FROM tblEmployee
```

CHAR and ASCII

The CHAR function converts an ASCII code into a character. The following statement converts the ASCII value 71 to the letter “G”:

```
SELECT CHAR(71) AS Character
```

The ASCII function takes the “G” and converts it back to 71:

```
SELECT ASCII('G') as Ascii
```

LOWER and UPPER

The LOWER and UPPER functions convert strings to lowercase and uppercase, respectively:

```
SELECT UPPER(Units) AS Up,
  LOWER(Units) AS Low
FROM tblUnit
```

LTRIM and RTRIM

The LTRIM and RTRIM remove leading and trailing blanks respectively, which returns a string after removing all trailing blanks. There is no TRIM function—you need to use both together to achieve trimming both leading and trailing blanks:

```
SELECT LTRIM(RTRIM(Units)) AS Trimmed
FROM tblUnit
```

RTRIM is used frequently on char columns to strip any trailing spaces.

Date and Time Functions

Date and time functions perform operations on date and time values, returning a string, numeric, or date/time value.

GETDATE

GETDATE returns the current system date and time:

```
SELECT GETDATE() AS RightNow
```

MONTH, DAY, and YEAR

The MONTH, DAY, and YEAR functions can be used to return the parts of a date. The following SELECT statement will return the month number, the day, and the year.

```
SELECT MONTH(GETDATE()) AS ThisMonth,
  DAY(GETDATE()) AS ThisDay,
  YEAR(GETDATE()) AS this year
```

DATEPART and DATENAME

If you want to break down date and time information further, use DATEPART. Here's the syntax:

```
DATEPART(part, date)
```

The part is an abbreviation you supply indicating the part of the date you'd like returned. Table 3 shows the abbreviations you can use with DATEPART.

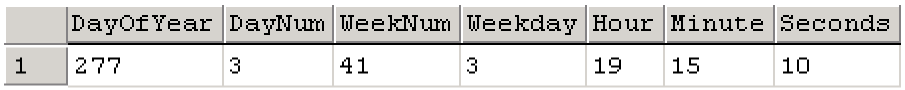
Part of Date	Abbreviations
Year	yy, yyyy
Quarter	qq, q
Month	mm, m
Day of year	dy, y
Day	dd, d
Week	wk, ww
Weekday	dw
Hour	hh
Minute	mi, n
Second	ss, s
Millisecond	ms

Table 3. DATEPART syntax.

The following query uses DATEPART to pick apart the current date:

```
SELECT
  DATEPART(dy, GETDATE()) AS DayOfYear,
  DATEPART(dd, GETDATE()) AS DayNum,
  DATEPART(ww, GETDATE()) AS WeekNum,
  DATEPART(dw, GETDATE()) AS Weekday,
  DATEPART(hh, GETDATE()) AS Hour,
  DATEPART(mi, GETDATE()) AS Minute,
  DATEPART(ss, GETDATE()) AS Seconds
```

The result set will look something like that shown in Figure 4.



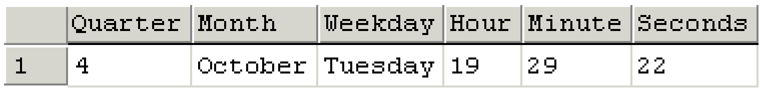
	DayOfYear	DayNum	WeekNum	Weekday	Hour	Minute	Seconds
1	277	3	41	3	19	15	10

Figure 4. The results from using DATEPART.

The DATENAME function is used to return a string representing a date part. The following query uses DATENAME instead of DATEPART to pick apart the current date:

```
SELECT
  DATENAME(qq, GETDATE()) AS Quarter,
  DATENAME(mm, GETDATE()) AS Month,
  DATENAME(dw, GETDATE()) AS Weekday,
  DATENAME(hh, GETDATE()) AS Hour,
  DATENAME(mi, GETDATE()) AS Minute,
  DATENAME(ss, GETDATE()) AS Seconds
```

The results are similar to DATEPART, as shown in Figure 5.



	Quarter	Month	Weekday	Hour	Minute	Seconds
1	4	October	Tuesday	19	29	22

Figure 5. The results from the DATENAME function.

DATEADD and DATEDIFF

DATEADD is used to perform computations on dates by adding a specified interval. The following query adds 1 to the year, month, and day:

```
SELECT
  DATEADD(yy, 1, GETDATE()) AS AddYear,
  DATEADD(mm, 1, GETDATE()) AS AddMonth,
  DATEADD(dd, 1, GETDATE()) AS AddDay
```

To subtract from a date, you use DATEADD and pass in a negative number. DATEDIFF is used if you want to calculate the length of time between two date/time values. As with DATEADD, DATEDIFF allows you to specify the unit of measure. The following query uses the Northwind database to calculate the number of days elapsed between various dates in the Northwind Orders table. The first date passed to DATEDIFF is subtracted from the second date.

```
USE Northwind
SELECT
  OrderDate, RequiredDate, ShippedDate,
  DATEDIFF(dd, OrderDate, RequiredDate) AS LeadTime,
  DATEDIFF(dd, OrderDate, ShippedDate) AS DaysToShip,
  DATEDIFF(dd, ShippedDate, RequiredDate) AS DaysEarly
FROM Orders
```

Using the @@ Functions

Functions that start with the @@ symbols are often referred to as global variables, but they're not really variables—you can't assign values to them or work with them the way you work with real variables. They're functions (and not all of them are global, either). They are used to return different kinds of information about what's going on in SQL Server.

@@ROWCOUNT

@@ROWCOUNT returns the number of rows returned or affected by a statement that selects or modifies data. It returns 0 if no values are returned by a SELECT statement or are affected by an action query. The following batch selects first the LastName values of customers living in California, and then uses @@ROWCOUNT to return the number of rows:

```
SELECT LastName, State
FROM tblCustomer
WHERE State = 'CA'
SELECT @@ROWCOUNT AS NumInCA
```

@@TRANCOUNT

The @@TRANCOUNT function is used for keeping track of things when you use explicit transactions. A BEGIN TRAN statement sets @@TRANCOUNT to 1. A ROLLBACK TRAN statement decrements @@TRANCOUNT by 1, or if @@TRANCOUNT is 1, it gets set to 0. The COMMIT TRAN statement decrements @@TRANCOUNT by 1. If you have nested transactions, @@TRANCOUNT helps you keep track of how many transactions are still pending. The following example uses the @@ROWCOUNT function to determine whether the transaction should be committed. If @@TRANCOUNT is greater than zero after the COMMIT TRAN statement, something went wrong (as it should, since Units is the primary key in the tblUnit table) and the transaction is rolled back:

```
BEGIN TRAN
  UPDATE tblUnit
    SET Units = 'Witless Update'
IF @@ROWCOUNT > 0
    COMMIT TRAN
IF @@TRANCOUNT > 0
    ROLLBACK TRAN
```

@@IDENTITY

The @@IDENTITY function is used to retrieve the new value after inserting a row in a table that has an identity column. The following example inserts a new row and then selects the new identity column value:

```
INSERT INTO tblCategory (Category)
VALUES ('Shark Lingerie')
SELECT @@IDENTITY AS NewCategoryID
```

TIP: The @@IDENTITY function can only be relied upon if it immediately follows the INSERT statement. The value returned by @@IDENTITY is always the identity value that was generated by the last statement executed in that connection. In addition, it will not work correctly if there are triggers that perform cascading inserts on other tables—you'll get the identity value from the last table that had data inserted. SQL Server 2000 introduces the new SCOPE_IDENTITY() and IDENT_CURRENT(*tablename*) functions to work around these problems. SCOPE_IDENTITY, returns the last identity value generated for any table in the current session and the current scope—values inserted by triggers aren't considered part of the current scope. IDENT_CURRENT allows you to retrieve the last identity value inserted in a specified table, regardless of the session or scope where that last insert occurred.

@@ERROR

The @@ERROR function is used to return any errors on the statement that preceded it. Error handling is covered later in this chapter.

```
UPDATE tblUnit
  SET Units = 'Witless Update'
SELECT @@ERROR AS WitlessError
```

Control of Flow

Control of flow syntax is used to branch and loop as necessary. Probably the most familiar construct in all of programming is the IF statement.

IF...ELSE

IF conditionally executes a statement if the condition is true. If it's not true, then an optional ELSE executes another statement.

```
IF ThisStatementIsTrue
  ExecuteThisStatement
ELSE
  ExecuteThisStatementInstead
```

The following example sums the prices in the products table. If the result is greater than 100, then 'High' is returned. If it's not, then 'Low' is returned.

```
IF (SELECT SUM(Price) FROM tblProduct) > 100
  SELECT 'High' AS HiSum
ELSE
  SELECT 'Low' AS LoSum
```

BEGIN…END

The preceding example had a single statement following both the IF and the ELSE statements. Note that there is no END IF in Transact-SQL. If you want multiple statements to execute conditionally, you must surround them with a BEGIN…END block.

The following example uses the same query as the preceding example, but extends it by subtracting ten cents from all prices if the sum is greater than 100, and adds ten cents to all prices if the sum is less than 100:

```
IF (SELECT SUM(Price) FROM tblProduct) > 100
  BEGIN
    UPDATE tblProduct
    SET Price = Price - .10
    SELECT 'Ten cents removed from all prices'
  END
ELSE
  BEGIN
    UPDATE tblProduct
    SET Price = Price + .10
    SELECT 'Ten cents added to all prices'
  END
```

GOTO, RETURN, and Labels

The GOTO statement is used to jump to a label. It's mostly used in error handling or to allow you to write spaghetti Transact-SQL code if you're so inclined. The following example runs one query if the MONTH function returns 3 (March), and jumps to the label if it returns some other month and executes the statements after the label.

```
IF MONTH(GETDATE()) = 3
  GOTO SomeLabel
SELECT * FROM tblCustomer

SomeLabel:
  SELECT * FROM tblProduct
```

The RETURN statement unconditionally exits and stops executing any further statements. The following example will not run the SELECT query if the month is 3. Any other month will cause all of the rows from the tblCustomer table to be retuned.

```
IF MONTH(GETDATE()) = 3
  RETURN
SELECT * FROM tblCustomer
```

CASE

The CASE expression is used in two distinct capacities: A simple CASE statement compares the results against a set of simple responses for replacement, and a searched CASE expression compares the results against a set of Boolean values to determine if a match exists. CASE is most often used within a SELECT statement to provide alternate values for the input.

```
CASE input_expression
  WHEN when_expression THEN result_expression
    [...n]
  [ELSE else_result_expression]
END
```

The following example replaces type codes in the titles table in the pubs sample database with a “friendlier” description:

```
SELECT Category =
  CASE type
    WHEN 'popular_comp' THEN 'Popular Computing'
    WHEN 'mod_cook' THEN 'Modern Cooking'
    WHEN 'business' THEN 'Business'
    WHEN 'psychology' THEN 'Psychology'
    WHEN 'trad_cook' THEN 'Traditional Cooking'
    ELSE 'N/A'
  END,
  Title, Price
FROM titles
```

VBA has a construct that is very useful for evaluating expressions in queries called the Immediate If:

```
Iif(expression, truepart, falsepart)
```

However, it's not supported in Transact-SQL. You can use CASE to provide the same functionality:

```
CASE
  WHEN Expression THEN TruePart
  ELSE FalsePart
END
```

The following query determines the average product price and then compares it against the actual price, returning whether it's above or below the average:

```
DECLARE @Avg money
SELECT @Avg = AVG(Price) FROM tblProduct
SELECT Product, Price, Ranking =
  CASE
    WHEN Price > @Avg
    THEN 'above average'
    ELSE 'below average'
  END
FROM tblProduct
GROUP BY Product, Price
```

CASE is not really a flow control construct in Transact-SQL. Rather than determining the order of execution, each CASE block is actually a single expression, and CASE expressions can be used outside of Transact-SQL batches. For example, if you wanted a computed column to concatenate first name, middle initial, and last name, without adding an extra space where middle initial was null, you could use this expression as the formula for your computed column:

```
(CASE WHEN ([MiddleInitial] IS NULL) THEN ([FirstName] + '
' + [LastName]) ELSE ([FirstName] + ' ' + [MiddleInitial]
+ ' ' + [LastName]) END)
```

WHILE

The WHILE statement is used to set up a loop. The loop will continue until the condition is met. As with the IF statement, if more than one statement is going to be used it must be contained within a BEGIN .. END wrapper:

```
WHILE Boolean_expression
  {sql_statement | statement_block}
```

In the following example the price is increased in the product table as long as the average price is less than 12 and the maximum price is not over 50.

```
WHILE (SELECT AVG(price) FROM titles) <= $12
  BEGIN
    UPDATE tblProduct
      SET Price = Price * 1.01
    IF (SELECT MAX(Price) FROM tblProduct) > $50
      BREAK
    ELSE
      CONTINUE
  END
```

Error Handling

Error handling in Transact-SQL will seem cumbersome if you're used to error handling in programming languages such as VBA. There's no such thing as an "On Error Goto" statement that allows you to have an error-handling section in your code that processes all run-time errors. All errors are handled inline as they occur, on a statement-by-statement basis. An additional complication is that, unlike VBA where every run-time error is fatal, very few Transact-SQL errors are. A fatal run-time error for SQL Server would be system wide, like running out of disk space or memory. In that situation, all bets are off and none of your statements are going to execute anyway. But what might not be fatal to SQL Server could very well be to your application—you can hit a constraint error or a data type error, and unless you specifically check for an error condition on the next line, SQL Server will just keep executing any remaining statements.

Using @@Error

As shown earlier in this chapter, the @@ERROR function is used to return the error number of the Transact-SQL error that occurred. You need to test for errors *after any and every statement that could possibly cause a run-time error*. In the following example, an attempt is made to insert a row into the tblOrder table that has an invalid CustomerID. The @@IDENTITY function is called immediately following the INSERT statement:

```
INSERT INTO tblOrder (CustomerID)
  VALUES(0)
SELECT @@IDENTITY AS NewOrder
SELECT @@ERROR AS Err
```

The problem here is that although @@IDENTITY returns a value, it's not the right one because the new row was not inserted into the table. And the @@ERROR function is called a line too late—it returns zero since the fact that @@IDENTITY returned a bogus value isn't even considered a run-time error.

To fix the procedure, the @@ERROR function has to come immediately after the INSERT statement. Then an IF statement branches according to whether or not an error occurred on the insert:

```
INSERT INTO tblOrder (CustomerID)
  VALUES(0)
IF @@ERROR > 0
  SELECT 'An error occurred.'
ELSE
  SELECT @@IDENTITY AS NewOrder
```

Using RAISERROR for Ad Hoc Error Messages

In the previous example, the error is trapped, but a scary error message is also returned to the client (in this case, the results pane of the Query Analyzer):

```
Server: Msg 547, Level 16, State 1, Line 1
INSERT statement conflicted with COLUMN FOREIGN KEY
constraint 'CustomerOrder'. The conflict occurred in
database 'Shark', table 'tblCustomer', column
'CustomerID'.
```

Using RAISERROR allows you to generate your own errors and pass them back to the client. Depending on the severity level the error will also be written to the Windows NT event log. Here's the syntax:

```
RAISERROR ({msg_id | msg_str}{, severity, state}
    [, argument
        [,...n]] )
    [WITH option[,...n]]
```

When you call RAISERROR, @@ERROR returns the SQL Server error number passed as the *msg_id* argument. Error numbers for user-defined error messages should be greater than 50,000. Ad hoc messages, which are simply user-defined error messages, raise an error of 50,000. The maximum value for msg_id is 2,147,483,647 (2 (31) - 1). The state parameter doesn't mean anything to SQL Server, and it can be any arbitrary number you assign, with values ranging from 1 to 127. However, the severity level (see Table 4) determines how SQL Server handles the error and whether or not it gets logged in the Windows NT/Windows 2000 event log. You can set any severity level for your own custom ad hoc error messages.

Severity Level	**Meaning**
0 – 10	Informational messages
15	Warnings
16 and higher	Errors
19 and higher	Errors are automatically written to the SQL Server and Windows NT/Windows 2000 event logs
20 or higher	Fatal errors (can only be set by a system administrator)

Table 4. Severity levels for RAISERROR.

NOTE Use the WITH LOG option to cause an error of any severity to be written to the log. Only members of sysadmin can use WITH LOG.

The previous example has been rewritten to return a more understandable error message to the client that an invalid CustomerID has been supplied:

```
INSERT INTO tblOrder (CustomerID)
  VALUES(0)
IF @@ERROR > 0
  RAISERROR ('Invalid CustomerID supplied.', 16, 1)
ELSE
  SELECT @@IDENTITY AS NewOrder
```

You can also use RAISERROR in situations that aren't strictly defined as errors. The following example raises and logs an error if the prices are dropped too low:

```
DECLARE @avg money
SELECT @avg = AVG(Price) FROM tblProduct

IF @avg < 15
  RAISERROR ('Average price too low--error time.', 16, 1)
ELSE
  PRINT 'Prices OK, no error'
```

Summary

- Transact-SQL is the SQL Server extension of the SQL query language and contains elements that are not part of the ANSI SQL-92 standard.
- Transact-SQL supports many programming language constructs like variables and control-of-flow statements.
- Transact-SQL does not perform implicit data type conversions. You need to use CAST or CONVERT when converting data types.
- Always use IS NULL to test for null values. There are built-in functions for working with nulls.
- The built-in functions available for use in processing numbers, strings, and dates are all fully documented in Books Online.
- Control-of-flow statements such as IF, CASE, and GOTO can be used to branch in your Transact-SQL code.
- BEGIN-END blocks are necessary if multiple statements are to be executed in any control-of-flow structure.
- WHILE loops allow you to set up a loop and process records.
- The @@ERROR system function will return the error number of any errors occurring on the statement immediately preceding it.
- RAISERROR is used to create ad hoc errors and pass back information to client applications.

(Review questions and answers on the following pages.)

Questions

1. How do you declare a variable in Transact-SQL?

2. Name two functions that can be used to convert data types in Transact-SQL.

3. What is the difference between IS NULL, ISNULL, and NULLIF?

4. What do you need in an IF…ELSE if you have multiple statements to execute for the IF and ELSE blocks?

5. When is the value of the @@ERROR function available?

6. How can you create an ad hoc error message with a more friendly text than the built-in error messages?

Answers

1. How do you declare a variable in Transact-SQL?
   <u>Use the DECLARE key word, name the variable with the @ symbol, and assign a data type: 'DECLARE @myvariable varchar(50)'.</u>

2. Name two functions that can be used to convert data types in Transact-SQL.
   <u>CAST, CONVERT, and STR</u>

3. What is the difference between IS NULL, ISNULL, and NULLIF?
   <u>The IS NULL statement is used to test for nulls, the ISNULL function converts a null to another value, and the NULLIF function converts a value to a null.</u>

4. What do you need in an IF…ELSE if you have multiple statements to execute for the IF and ELSE blocks?
   <u>A BEGIN…END block</u>

5. When is the value of the @@ERROR function available?
   <u>Only on the line immediately after the statement that caused the error</u>

6. How can you create an ad hoc error message with a more friendly text than the built-in error messages?
   <u>Use RAISERROR</u>

Lab 7: Programming with Transact-SQL

TIP: Because this lab includes a great deal of typed code, we've tried to make it simpler for you. You'll find all the code in **TSQL-Lab.sql**, in the same directory as the sample project. To avoid typing the code, you can cut/paste it from the text file instead.

Lab 7 Overview

In this lab you'll learn how to write Transact-SQL code. You'll convert data types, work with built-in functions, use control-of-flow constructs, and handle errors.

To complete this lab, you'll need to work through four exercises:

- Converting Data Types
- Using Built-In Functions
- Using Control of Flow Constructs
- Handling Errors

Each exercise includes an "Objective" section that describes the purpose of the exercise. You are encouraged to try to complete the exercise from the information given in the Objective section. If you require more information to complete the exercise, the Objective section is followed by detailed step-by-step instructions.

Converting Data Types

Objective

In this exercise, you'll create a SELECT query that concatenates the ProductID, Product and Price values from tblProduct into a single column with the literal string "Product: " appearing before the ProductID value, and the literal string "Price: " appearing before the Price value.

Things to Consider

- What is the Transact-SQL concatenation operator?
- How do you concatenate string values with numeric values?

Step-by-Step Instructions

1. Launch the Query Analyzer and make sure the Shark database is selected.
2. Create the basic SELECT statement with the columns and values you are going to use. Execute the query to make sure that it works.

```
SELECT ProductID, Product, Price
FROM tblProduct
```

3. Concatenate the values from the three columns into a single column named PriceList using the plus (+) operator and test the query.

```
SELECT ProductID + Product + Price AS ProductList
FROM tblProduct
```

4. You will receive the following error:

```
Server: Msg 245, Level 16, State 1, Line 5
Syntax error converting the varchar value 'Great White
12" Shark Case' to a column of data type int.
```

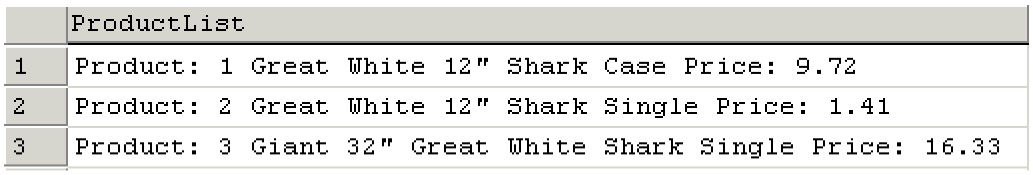
5. Use the CONVERT function to convert the numeric data types to strings so that they can be concatenated together:

```
SELECT
  CONVERT(varchar(5), ProductID)
  + Product +
  CONVERT(varchar(5), Price)
  AS ProductList
FROM tblProduct
```

6. Execute the query. You won't receive an error this time, but all of the values will be munged together. Fix this by concatenating the literal strings for Product: and Price: in front of the ProductID and Price values, respectively. Concatenate an empty space after ProductID and before Product. The query should look like the following:

```
SELECT
  'Product: ' +
  CONVERT(varchar(5), ProductID)
  + ' ' +
  + Product +
  ' Price: ' +
  CONVERT(varchar(5), Price)
  AS ProductList
FROM tblProduct
```

Figure 6 shows the first few rows of the result set.



	ProductList
1	Product: 1 Great White 12" Shark Case Price: 9.72
2	Product: 2 Great White 12" Shark Single Price: 1.41
3	Product: 3 Giant 32" Great White Shark Single Price: 16.33

Figure 6. The first few rows of the result set.

Using Built-In Functions

Objective

In this exercise, you'll work with the OrderDate in the tblOrder table to create a query that has a computed value calculating the date that is two weeks in advance of the OrderDate. You'll then format the data so that only the date value is displayed.

Things to Consider

- Which built-in function is used to calculate a future date?
- How do you create a computed value in a query?
- How do you control the display of date/time data?

Step-by-Step Instructions

1. Launch the Query Analyzer if it is not already open and create a SELECT query selecting the OrderID and OrderDate columns from tblOrder:

```
SELECT OrderID, OrderDate
FROM tblOrder
```

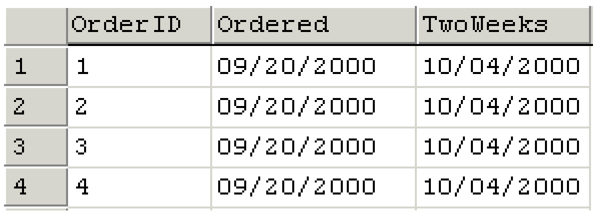
2. Create the computed value using the DATEADD function, adding two weeks to the OrderDate. Name the column TwoWeeks.

```
SELECT OrderID, OrderDate,
  DATEADD(ww, 2, OrderDate) AS TwoWeeks
FROM tblOrder
```

3. The result set looks a bit ragged because the minute and second data is also displayed. Use the CONVERT function to display only the date, and omit the display of the time value. Look up the needed values from Books Online in the CAST and CONVERT topic. The completed query should look like the following:

```
SELECT OrderID,
  CONVERT(varchar(12), OrderDate, 101), AS Ordered
  CONVERT(varchar(12), DATEADD(ww, 2, OrderDate), 101)
    AS TwoWeeks
FROM tblOrder
```

Figure 7 shows the first few rows of the result set.



	OrderID	Ordered	TwoWeeks
1	1	09/20/2000	10/04/2000
2	2	09/20/2000	10/04/2000
3	3	09/20/2000	10/04/2000
4	4	09/20/2000	10/04/2000

Figure 7. The first few rows of the result set.

Using Control of Flow Constructs

Objective

In this exercise, you'll count the number of units sold for ProductID 1, saving the value in a variable. If sales are above a certain number, you'll execute a query that selects the product information from the products table. If sales are below that number, you'll execute a query displaying sales of other products that are above the number. In each case, you'll print out a message to be displayed in the Messages tab.

Things to Consider

- How do you declare a variable?
- What data type does the variable need to be?
- How do you execute multiple statements in an IF structure?
- How do you display text in the Messages pane in the Query Analyzer?

Step-by-Step Instructions

1. Declare a variable with a data type of int. Create the SELECT query to count the number of items sold. Highlight only the SELECT statement and execute the query. Jot down the number—this is what will be stored in the variable.

```
DECLARE @count int
SELECT @count = (SELECT SUM(Quantity)
  FROM tblOrderDetail WHERE ProductID = 1)
```

2. Assuming that the sum of the quantity of items sold for Product 1 is 1,026, create the following IF statement to test if the sum is greater than 1,030. Add a PRINT statement to write to the Messages pane. Make sure to enclose the SELECT statement and the PRINT statement in a BEGIN…END block.

```
IF @count > 1030
  BEGIN
    SELECT Product, Price FROM tblProduct
    WHERE ProductID = 1
    PRINT 'This product is doing well.'
  END
```

3. Add the ELSE statement, selecting products from the tblOrderDetail table whose quantity sold exceeds the value stored in @count.

```
ELSE
  BEGIN
    SELECT SUM(Quantity) AS NumSold,
      ProductID FROM tblOrderDetail
    GROUP BY ProductID
    HAVING SUM(Quantity) > @count
    PRINT 'Other products doing better.'
  END
```

4. Execute the entire query. Check the Messages tab to make sure the message printed out correctly.

5. Modify the query by placing a different value in the “IF @count > 1030” statement, and verify that the correct results were obtained for a different condition.

Handling Errors

Objective

In this exercise, you'll create a procedure to insert a row into the tblUnit table. You'll test to see if an error occurred, and if it did, create an ad hoc error message to return that information to the client.

Things to Consider

- Which function do you use to trap errors?
- Where does this function go?
- How do you return a custom error message to the client?

Step-by-Step Instructions

1. Create the following INSERT statement to add a row to the tblUnit table.

```
INSERT INTO tblUnit(Units)
  VALUES ('Each')
```

2. Since this value already exists in tblUnit, executing the query will cause a run-time error. Declare a variable to hold the error value and use the @@ERROR function to test to see if an error occurred.

```
DECLARE @err int
INSERT INTO tblUnit(Units)
  VALUES ('Each')
SET @err = @@ERROR
```

3. Set up an IF statement to test to see if an error occurred. If it did, use RAISERROR to return an error message to the client.

```
DECLARE @err int
INSERT INTO tblUnit(Units)
  VALUES ('Each')
SET @err = @@ERROR
IF @err <> 0
  RAISERROR('This value already exists.', 16, 1)
```

4. The Messages pane should display the following information:

```
Server: Msg 2627, Level 14, State 1, Line 1
Violation of PRIMARY KEY constraint 'aaaaatblUnit_PK'.
Cannot insert duplicate key in object 'tblUnit'.
The statement has been terminated.
Server: Msg 50000, Level 16, State 1, Line 6
This value already exists.
```