Lesson 1: Creating Tables

The basic analogy for a database *table* is that it is like a single worksheet within a spreadsheet. As you work with a worksheet, you enter information in rows and columns. Well-designed spreadsheets generally have column headers that give you an idea of the kind of data you find in a column. The problem with working with data in a worksheet, however, is that it doesn't enforce any structure. You can place any type of data in any column without limitation.

Database tables also store rows of data in columns. And each of these columns has a name associated with it to provide an easy way to reference a particular piece of data. But what sets a table and database apart from a Microsoft Office Excel worksheet or spreadsheet, for example, is the strict enforcement of the data that you can enter in a column. SQL Server enforces this data structure by using data types as well as properties that you can add to further define a column. In this lesson, you will learn how to make the best choices when defining data types and properties for columns, how to create a table, and then how to assign appropriate permissions to allow access to a table.

After this lesson, you will be able to:

- Specify column details including data type.
- Specify the filegroup.
- Implement a table.
- Assign permissions to a role for tables.

Estimated lesson time: 30 minutes

Understanding Data Types

Data types limit the type of data that you can store in a column and, in some cases, even limit the range of possible values in the column. The data type that you choose for a column is the most critical decision that you make within your database. If you choose a data type that is too restrictive, applications cannot store the data they are supposed to process, leading to a large design effort. If you choose too broad a data type, however, you wind up consuming more space than necessary on disk and in memory, which can create a resource and performance issue.

When selecting a data type for a column, you should choose the data type that allows all the data values that you expect to be stored while doing so in the least amount of

space possible. SQL Server data types fall into seven general categories, which Table 3-1 describes.

Table 3-1 Seven Categories of SQL Server Data Types

Data Type Category	General Purpose
Exact numeric	Stores precise numbers either with or without decimals
Approximate numeric	Stores numeric values with or without decimals
Monetary	Stores numeric values with decimal places; used specifically for currency values with up to four decimal places
Date and time	Stores date and time information and enables special chronological enforcement, such as rejecting a value of February 30
Character	Stores character-based values of varying lengths
Binary	Stores data in a strict binary (0 and 1) representation
Special purpose	Complex data types that require specialized handling, such as XML documents or globally unique identifiers (GUIDs)

Let's look at each of these data type categories to see how you can use the different data types to provide the basic definition of each column in a table. You use these data types when defining permanent tables, temporary tables, table variables, and variables. There are few restrictions to the data types that you can use in stored procedures, triggers, and functions.

MORE INFO Data type definitions

For more detailed information about each data type, including explicit storage details and restrictions, see the SQL Server 2005 Books Online topics "Data Types (Transact-SQL)" and "Data Type Conversion (Database Engine)." SQL Server 2005 Books Online is installed as part of SQL Server 2005. Updates for SQL Server 2005 Books Online are available for download at *www.microsoft.com/technet/prodtechnol/sql/2005/downloads/books.mspx*.

Exact Numeric Data Types

You use exact numeric data types to store numbers that have zero or more decimal places. You can manipulate the numbers that you store in these data types by using any mathematical operation without requiring any special handling. The storage is also precisely defined, so any data stored in these data types returns and calculates to the same value on either an Intel or an AMD processor architecture. Table 3-2 lists the exact numeric data types that SQL Server supports.

Table 3-2 Exact Numeric Data Types

Data Type	Storage	Value Range	Purpose
bigint	8 bytes	−2E63 to 2E63 −1	Stores very large whole numbers that can be positive or negative
int	4 bytes	−2E31 to 2E31 −1	Stores whole numbers that can be positive or negative
smallint	2 bytes	−32,768 to 32,767	Stores whole numbers that can be positive or negative
tinyint	1 byte	0 to 255	Stores a small range of positive whole numbers
decimal(p,s)	5–17 bytes depending on the precision	−10E38 + 1 to 10E38 −1	Stores decimals up to a maximum of 38 places
numeric(p,s)	5–17 bytes depending on the precision	−10E38 + 1 to 10E38 −1	Functionally equivalent to decimal, and can be used interchangeably with decimal

The *decimal* and *numeric* data types accept parameters to complete the data type definition. These parameters define the precision and scale for the data type. For example, *decimal*(12,4) defines a decimal value that can have up to 12 total digits, with four of those digits after the decimal.

The most common data types from this group are *int* and *decimal*. You can use a *decimal* data type to store integer values, but doing so requires extra bytes of storage per row and should not be used for this purpose.

Although *int* data types can store both positive and negative numbers, the negative portion is very rarely used. The *int* data types are commonly used—and commonly misused. If the range of values you plan to store in a column do not exceed 32,767, you can save two bytes for every row by using *smallint* instead of *int*. If the values are going to range only from 0 to 255, you can save three bytes for every row by using *tinyint*.

IMPORTANT Space utilization

Saving two or three bytes of storage per row doesn't seem like a lot compared to the 250+ GB hard drives that you can now purchase for a few hundred dollars, pounds, euros, yen, or whatever currency you are working with. However, hard disk storage is a minor concern. If you store 1 million rows of data in a table, which is very common, the bytes per row saved would add up to 2 or 3 MB. Although that does not sound like much, consider that you also save that much space in memory if a user executes a query that returns all the rows in the table. You also save thousands of processor cycles at the same time.

The space issue becomes even larger when you join two tables together. Joining two *int* columns together consumes eight bytes of memory as well as the corresponding calculation on the processor. If both tables hold 1 million rows and need to be read completely, the operation consumes about 8 MB of memory space. If you could have stored the data in a *smallint* or *tinyint* column instead, the memory savings for this query would be 4–6 MB. And that is the savings for only a single query. Consider what would happen if thousands of queries are being processed against the database, and you can see how one or two bytes of savings per row based on the data type you use can quickly make the difference between an environment with good performance and one with very poor performance.

Approximate Numeric Data Types

Approximate numeric data types can store decimal values. However, data stored in a *float* or *real* data type is exact only to the precision specified in the data type definition. Any digits to the right are not guaranteed to be stored exactly. For example, if you stored 1.00015454 in a data type defined as *float*(8), the column is guaranteed to return only 1.000154 accurately. SQL Server rounds off any digits further to the right when it stores the data. Therefore, calculations involving these data types compound rounding errors. Transferring databases containing tables with these data types between Intel and AMD processors also introduces errors. Table 3-3 lists SQL Server's approximate numeric data types.

Table 3-3 Approximate Numeric Data Types

Data Type	Storage	Value Range	Purpose
float(p)	4 or 8 bytes	−2.23E308 to 2.23E308	Stores large, floating point numbers that exceed the capacity of a decimal data type
real	4 bytes	−3.4E38 to 3.4E38	Still valid, but replaced by float to meet the SQL-92 standard

The *float* data types accept a parameter in the definition that determines the number of digits to store precisely. For example, a *float*(8) column precisely stores seven digits, and anything exceeding that is subject to rounding errors.

Because of the imprecision associated with these data types, they are rarely used. You should consider using *float* only in cases in which an exact numeric data type is not large enough to store the values.

Monetary Data Types

Monetary data types are designed to store currency values with four decimal places of precision. Table 3-4 lists SQL Server's monetary data types.

Table 3-4 Monetary Data Types

Data Type	Storage	Value Range	Purpose
money	8 bytes	−922,337,203,685,477.5808 to 922,337,203,685,477.5807	Stores large currency values
smallmoney	4 bytes	−214,748.3648 to 214,748.3647	Stores small currency values

The *smallmoney* data types are rarely defined in databases, even though this data type is the most accurate choice for many applications that deal with products and orders. It is much more common for these databases to incorrectly use the *money* data type and waste four bytes of storage for each row stored.

Although *money* and *smallmoney* data types are designed to store currency values, they are rarely used in financial applications. Instead, these applications use a *decimal*

data type because they need to perform accurate calculations to 6, 8, and even 12 decimal places.

Date and Time Data Types

In storing data, nothing generates more controversy than figuring out how to store dates and times. Some applications need to store only a date. Other applications need to store only a time. And still other applications need to store both dates and times together. Unfortunately, SQL Server stores this type of data only together as both a date and a time—for example, 2006-03-14 20:53:36.153, which is the precise millisecond on the system clock when I started writing this sentence. Table 3-5 lists SQL Server's date and time data types.

Table 3-5 Date and Time Data Types

Data Type	Storage	Value Range	Purpose
datetime	8 bytes	January 1, 1753, through December 31, 9999, with an accuracy of 3.33 milliseconds	Stores large date and time values.
smalldatetime	4 bytes	January 1, 1900, through June 6, 2079, with an accuracy of 1 minute	Stores a smaller range of date and time values

The *datetime* and *smalldatetime* data types are stored internally as integers. The *datetime* data type is stored as a pair of four-byte integers, which together represent the number of milliseconds since midnight on January 1, 1753. The first four bytes store the date, and the second four bytes store the time. The *smalldatetime* data type is stored as a pair of two-byte integers, which together represent the number of minutes since midnight on January 1, 1900. The first two bytes store the date, and the second two bytes store the time.

Character Data Types

To store character data, you select one of the data types designed for this purpose. Each one consumes either one or two bytes of storage for each character, depending on whether the data type uses American National Standards Institute (ANSI) encoding or Unicode encoding.

Before looking at the character data types, let's look briefly at the background behind the ANSI and Unicode encodings. To handle the wide variety of languages in the world, computer technologists needed a way to store the many different characters of a language in a standard format. So, the ANSI standards body developed an encoding standard that required eight bits to represent the range of letters. The only problem was that every character could not be specified within a single eight-bit encoding. Thus, dozens of character sets were created that specified the acceptable characters for a given encoding. This approach worked well until you started transferring data between systems that used different character sets. If a character in one encoding did not exist in a different encoding, it was lost in the translation process. In addition to the encoding-translation issues, the eight-bit encoding couldn't capture several languages.

These problems led to the creation of the Unicode standard. The Unicode standard uses 2 bytes to represent each character. This extra space meant that all the character sets in use in the ANSI standard could be eliminated. Now, each unique character could be expressed within a single encoding schema. And because with Unicode there's just one encoding scheme, no encoding translation is necessary when transferring data between systems set for different languages. This makes character data completely transportable. The only downside is that Unicode data types require two bytes to store each character, so Unicode data types require twice as much space as their ANSI counterparts.

Unicode data types are preceded with an *n*. For example, *nchar* is the Unicode counterpart to the *char* data type, which uses the ANSI encoding. When defining a character data type, you specify the maximum number of bytes the column is allowed to store. For example, a *char*(10) can store a maximum of 10 characters because each character requires one byte of storage, whereas an *nchar*(10) can store a maximum of five characters because each Unicode character requires two bytes of storage. Table 3-6 lists SQL Server character data types.

Table 3-6 Character Data Types

Data Type	Storage	Number of Characters	Purpose
char(n)	1–8,000 bytes	Maximum of 8,000 characters	ANSI data type that is fixed width
nchar(n)	2–8,000 bytes	Maximum of 4,000 characters	Unicode data type that is fixed width

Table 3-6 Character Data Types

Data Type	Storage	Number of Characters	Purpose
varchar(n)	1–8,000 bytes	Maximum of 8,000 characters	ANSI data type that is variable width
varchar(max)	Up to 2 GB	Up to 1,073,741,824 characters	ANSI data type that is variable width
nvarchar(n)	2–8,000 bytes	Maximum of 4,000 characters	Unicode data type that is variable width
nvarchar(max)	Up to 2 GB	Up to 536,870,912 characters	Unicode data type that is variable width
text	Up to 2 GB	Up to 1,073,741,824 characters	ANSI data type that is variable width
ntext	Up to 2 GB	Up to 536,870,912 characters	Unicode data type that is variable width

Why are there so many character data types that appear to be equivalent to each other? The differences in the data types might be subtle, but they can be important. A *char* data type, either ANSI or Unicode, is a fixed-width data type. Therefore, it consumes the same amount of storage space regardless of the number of characters that are stored in the column. For example, a *char*(30) column consumes 30 bytes of storage space regardless of whether you store one character or 30 characters in the column. Any unused space is padded with spaces up to the maximum storage specified for the column. However, a *varchar*(30) column consumes only one byte for each character that is stored in the column.

The *text* and *ntext* data types are designed to store large amounts of character-based data. However, *text* and *ntext* columns aren't allowed with many operations. For example, you cannot use them with an equality operator or join them together. Many system functions also cannot use *text* and *ntext* data types.

Because of these limitations, SQL Server 2005 introduced the *varchar*(max) and *nvarchar*(max) data types. These data types combine the capabilities of both *text*/*ntext*

and *varchar/nvarchar* data types. They can store up to 2 GB of data and do not have any restrictions on the operations that you can perform with them or on the functions you can use them with.

Binary Data Types

There are many times when you need to store binary data. So SQL Server provides three data types that let you store various amounts of binary data in a table. Table 3-7 lists SQL Server's binary data types.

Table 3-7 Binary Data Types

Data Type	Storage	Purpose
binary(n)	1–8,000 bytes	Stores fixed-size binary data
varbinary(n)	1–8,000 bytes	Stores variable-size binary data
varbinary(max)	Up to 2 GB	Stores variable-size binary data
image	Up to 2 GB	Stores variable-size binary data

You use the *binary* data types essentially to store files within SQL Server. You use the *binary/varbinary* data types for storing small files, such as a group of 4 KB or 6 KB files containing a variety of data in native format.

The most popular data type within this group is the *image* data type. This data type has an unfortunate name; it is not used exclusively to store images, such as a library of pictures from a recent vacation. Although you can store pictures in an *image* data type, you can also use this data type to store Word, Excel, PDF, and Visio documents. You can store any file that is 2 GB or less in size in an *image* data type. One of the most famous implementations of this data type is the TerraServer project, which is a multi-terabyte database of terrestrial images that you can access at *www.terraserver.com*.

The *varbinary*(max) data type is new to SQL Server 2005. It can store the same amount of data as an *image* data type, and you can use it with all the operations and functions that you can use with *binary/varbinary* data types.

Specialized Data Types

In addition to the preceding standard data types, SQL Server provides seven additional data types for very specific purposes. Table 3-8 describes these specialized data types.

Table 3-8 Specialized Date Types

Data Type	Purpose
bit	Stores a 0, 1, or *null*. Used for basic "flag" values. TRUE is converted to 1, and FALSE is converted to 0.
timestamp	An automatically generated value. Each database contains an internal counter that designates a relative time counter not associated with an actual clock. A table can have only one *timestamp* column, which is set to the database timestamp when the row is inserted or modified.
uniqueidentifier	A 16-bit GUID used to globally identify a row across databases, instances, and servers.
sql_variant	Can change the data type based on the data that is stored within it. Stores a maximum of 8,000 bytes.
cursor	Used by applications that declare cursors. Contains a reference to the cursor that can be used for operations. This data type cannot be used in a table.
table	Used to hold a result set for subsequent processing. This data type cannot be used for a column. The only time you use this data type is when declaring table variables in triggers, stored procedures, and functions.
Xml	Stores an XML document of up to 2 GB in size. You can specify options to force only well-formed documents to be stored in the column.

CAUTION sql_variant: just say no

The *sql_variant* data type, new in SQL Server 2005, is a dangerous data type that, in my opinion, should never have been added to SQL Server. This data type enables you to declare a column or variable without having to decide what type of data will be stored in it. The *sql_variant* data type then automatically "converts" itself into the type of data that is written into it.

Databases are useful because all data is explicitly declared and explicitly typed. By allowing a data type that has no defined type, all kinds of data-mismatch issues can arise. We *very strongly* recommend that you never use *sql_variant*.

MORE INFO sql_variant

For more information about the *sql_variant* data type, see the SQL Server 2005 Books Online article "sql_variant (Transact-SQL)."

Quick Check

- What are the six categories of standard data types that you can use to define columns in tables, and what is the general purpose of each category?

Quick Check Answer

- *Exact numeric data types* store precise integer or decimal values.
- *Approximate numeric data types* store floating-point numbers.
- *Monetary data types* store currency accurate to four decimal places.
- *Datetime* data types store dates and times.
- *Character data types* store text values.
- *Binary data types* store binary streams, normally files.

Nullability

The second characteristic of any column definition is whether it requires a value to be stored. Databases have a special construct called a *null* that you can use to denote the absence of a value—something similar to "unknown" or "not applicable." A *null* is not a value, nor does it consume storage. The best way to understand this construct is to look at an example.

Let's say that you are designing a table to store addresses of your company's customers. You have decided that each address can have up to three lines for the street address. Each address can also have a city, a state or province, a postal code, and a country. So you create a table that contains seven columns. Not every customer needs all three address lines to capture the street address, so one or two of these columns are not necessary for some addresses. Some customers live in countries that do not have states or provinces, so this column is also not necessary for every customer. In addition, when users input addresses, they might not know the postal code of certain customers, but they still need to be able to save all of the data that is known. These issues create a basic dilemma. You could stick a dummy value in the columns that either don't have values or the values aren't known when the data was entered. However, inserting dummy data can cause even more problems because you are adding invalid

data to your table—data that might be seen and used by an employee or customer. Generally, you would have users just omit the data. Because the data was not explicitly specified, it is either unknown or not applicable. In the database, the column would be *null* to designate this unknown state.

When you define columns, you can specify whether or not *nulls* are allowed. If you disallow *nulls*, a user is required to specify a value for the column.

Note that because it is impossible for the absence of something to equal the absence of something—in other words, one *null* cannot equal another *null*—you cannot use a null in comparisons.

MORE INFO Nulls

For more details about *nulls*, see the SQL Server 2005 Books Online article "Null Values."

Identity

When defining columns, you also have the ability to specify a special identify property for a single column in a table. Defining a column with the identity property causes SQL Server to generate an automatically incrementing number. The identity property takes two parameters: seed and increment. The seed value designates the starting value that SQL Server uses. The increment value specifies what number SQL Server adds to this starting value when generating each successive value. This property is equivalent to autonumber or autoincrement values in other languages.

You can use the identity property with the exact numeric data types: *bigint*, *int*, *smallint*, *tinyint*, *decimal*, and *numeric*. If you use *decimal* or *numeric* data types with the identity property, you must define them with 0 decimal places.

Computed Columns

You can also create a special type of column called a *computed* column, which contains a computation involving one or more other columns in the table.

By default, the computed column contains a definition for the computation but does not physically store data by default. When the data is returned, the computation is applied to return a result.

However, you can force a computed column to physically store data by using the *PERSISTED* keyword. This keyword causes the computation to occur when the row is inserted or modified, and the result of the computation is then physically stored in the table.

Creating a Table

Now that you have seen all the column details you can specify to define the structure of a table, you are ready to actually create a table. You can create three different types of tables in SQL Server: permanent, temporary, and table variables.

MORE INFO Normalization, naming conventions, and table design

Normalization, naming conventions, and various table-design methods are beyond the scope of this book. For information about these topics, see *MCITP Self-Paced Training Kit (Exam 70-443): Designing a Database Server Infrastructure by Using Microsoft SQL Server 2005*, Microsoft Press, 2007.

Permanent Tables

To create a table, you use the *CREATE TABLE* Transact-SQL command. The general syntax of this command is as follows:

```
CREATE TABLE
    [ database_name . [ schema_name ] . | schema_name . ] table_name
        ( { <column_definition> | <computed_column_definition> }
        [ <table_constraint> ] [ ,...n ] )
    [ ON { partition_scheme_name ( partition_column_name ) | filegroup
        | "default" } ]
    [ { TEXTIMAGE_ON { filegroup | "default" } ]
[ ; ]
```

To execute this command, you must be a member of the sysadmin fixed server role, a member of the database owner fixed database role, or have been granted the *CREATE TABLE* permission. When you use this command, you create a table in the database that can be accessed by any user with the appropriate permissions.

The *ON* clause specifies where the table will reside on physical storage. If you do not specify a filegroup, SQL Server creates the table on the default filegroup.

Using our earlier example, you could use the *CREATE TABLE* command to create the *CustomerAddress* table as follows:

```
CREATE TABLE dbo.CustomerAddress
(AddressLine1              varchar(30)    NOT NULL,
AddressLine2               varchar(30)    NULL,
AddressLine3               varchar(30)    NULL,
City                       varchar(50)    NOT NULL,
StateProvinceID            int            NULL,
PostalCode                 char(10)       NULL,
CountryID                  int            NULL)
```

This table definition specifies the following:

- The table will be created in the *dbo* schema.
- A minimum of one address line that has a maximum of 30 characters must be specified for every customer. The storage space consumed will be equal to the number of characters in the column.
- One or two optional address lines can be specified, each holding up to 30 characters and consuming storage space equal to the number of characters in the column.
- A customer record must have a city specified; the City column can hold a value up to 50 characters in length and consumes storage equal to the number of characters in the column.
- A customer can have an optional state/province specified. The column consumes four bytes of storage and contains an integer value.
- A customer can have an optional postal code specified. Each row consumes 10 bytes of storage.
- A customer can have an optional country specified. The column consumes four bytes of storage and contains an integer value.

Although the preceding table definition accurately captures the necessary data, you might have noticed a few problems. A customer might have one or more home addresses, one or more business addresses, and one or more shipping addresses. A customer might also want to designate a particular address as the primary address. So you might be tempted to add a lot of additional columns to handle these situations. But that would be thinking in terms of a spreadsheet, not a database. Instead, you can simply add a column to the table that designates the type of address and a column to designate the primary address, as the following example shows:

```
CREATE TABLE dbo.CustomerAddress
(AddressType              char(4)        NOT NULL,
PrimaryAddressFlag        bit            NOT NULL,
AddressLine1              varchar(30)    NOT NULL,
AddressLine2              varchar(30)    NULL,
AddressLine3              varchar(30)    NULL,
City                      varchar(50)    NOT NULL,
StateProvinceID           int            NULL,
PostalCode                char(10)       NULL,
CountryID                 int            NULL)
```

For now, we will ignore the questions concerning the StateProvinceID and CountryID columns because we will cover them in the next lesson on constraints.

But there is still one other problem with this table definition. We are capturing addresses, but we have no way of knowing which address corresponds with which customer. To complete the table structure and allow an address to be associated with a customer, we need to add one more column to the table: the CustomerAddressID *int* column, defined with the identity property. The complete table definition is as follows:

```
CREATE TABLE dbo.CustomerAddress
(CustomerAddressID          int             IDENTITY(1,1),
AddressType                 char(4)         NOT NULL,
PrimaryAddressFlag          bit             NOT NULL,
AddressLine1                varchar(30)     NOT NULL,
AddressLine2                varchar(30)     NULL,
AddressLine3                varchar(30)     NULL,
City                        varchar(50)     NOT NULL,
StateProvinceID             int             NULL,
PostalCode                  char(10)        NULL,
CountryID                   int             NULL)
```

NOTE Deleting tables

You use the *DELETE* command to remove rows from a table. And to remove an entire table, you use the *DROP TABLE* command. To execute this command, you must be a member of the sysadmin fixed server role, a member of the database owner fixed database role, or the owner of the table.

Temporary Tables

Temporary tables, as their name suggests, are temporary table structures. Temporary tables can be either global or local and can be created by any user. All temporary tables are created in the *tempdb* database.

A local temporary table is visible only to the user who created the table and only within the connection that was used to create the table. Local temporary tables are automatically dropped when the connection they are associated with is closed. You create a local temporary table by using the *CREATE TABLE* command and prepending a pound sign (#) to the table name.

The following example shows the command to create the earlier *CustomerAddress* table as a local temporary table:

```
CREATE TABLE #CustomerAddress
(CustomerAddressID          int             IDENTITY(1,1),
AddressType                 char(4)         NOT NULL,
PrimaryAddressFlag          bit             NOT NULL,
AddressLine1                varchar(30)     NOT NULL,
```

```
AddressLine2              varchar(30)    NULL,
AddressLine3              varchar(30)    NULL,
City                      varchar(50)    NOT NULL,
StateProvinceID           int            NULL,
PostalCode                char(10)       NULL,
CountryID                 int            NULL)
```

A global temporary table, in contrast, is visible to any user within the SQL Server instance. Global temporary tables are dropped when the last connection accessing the table is closed. You create a global temporary table by using the *CREATE TABLE* command and prepending two pound signs (##) to the table name, as the following example shows:

```
CREATE TABLE ##CustomerAddress
(CustomerAddressID        int            IDENTITY(1,1),
AddressType               char(4)        NOT NULL,
PrimaryAddressFlag        bit            NOT NULL,
AddressLine1              varchar(30)    NOT NULL,
AddressLine2              varchar(30)    NULL,
AddressLine3              varchar(30)    NULL,
City                      varchar(50)    NOT NULL,
StateProvinceID           int            NULL,
PostalCode                char(10)       NULL,
CountryID                 int            NULL)
```

BEST PRACTICES **Cleaning up**

Everything you read related to programming should have one recurring theme: "If you create it, you should delete it." This mantra applies to all temporary objects that you ever create. If you create a temporary table, you should drop it when you no longer need it. This allows resources to be reclaimed and ensures that structures are not left hanging around. You should never rely on a connection being closed to clean up any temporary tables, particularly because many applications use connection pools in which the connections are never closed. Explicitly dropping a temporary table after you finish using it ensures that you never receive any errors because of attempting to create the temporary table a second time.

Table Variables

Table variables provide an alternative to temporary tables and can be used in functions, triggers, and stored procedures. Instead of storing the table and all data within the table in the *tempdb* database on disk, a table variable and all associated data is stored in memory. However, if the amount of data placed into the table variable causes it to require more storage space than is available in memory, the overflow will be spooled to disk within *tempdb*.

Table variables are local to the function, trigger, or stored procedure they were created in and are automatically deallocated when the object is exited.

You create the customer address table as a table variable by declaring the table as a variable, which you denote by prepending the table name with the @ character, as follows:

```
DECLARE @CustomerAddress TABLE
(CustomerAddressID         int            IDENTITY(1,1),
AddressType                char(4)        NOT NULL,
PrimaryAddressFlag         bit            NOT NULL,
AddressLine1               varchar(30)    NOT NULL,
AddressLine2               varchar(30)    NULL,
AddressLine3               varchar(30)    NULL,
City                       varchar(50)    NOT NULL,
StateProvinceID            int            NULL,
PostalCode                 char(10)       NULL,
CountryID                  int            NULL)
```

Assigning Permissions

Now that you've created your table, you need to provide permissions for users to access it. As you learned in Chapter 2, "Configuring SQL Server 2005," all objects in SQL Server are secured. Furthermore, SQL Server does not provide any access unless permission has been explicitly granted.

A member of the sysadmin fixed server role has already been granted unlimited rights to any object within the SQL Server instance, so a member of this role can perform any operation on a table. A member of the database owner fixed database role has already been granted permission to perform any operation on any object within the database that is owned, so a member of this role can perform any operation on a table. Additionally, the owner of a table has already been granted explicit authority to perform any operation against a table that he or she owns. All other users must be assigned permissions to work with a table.

BEST PRACTICES Security assignments

Security best practices dictate that you never grant permissions directly to a user. Therefore, you should add a Microsoft Windows login to a Windows group and the Windows group as a login to SQL Server. You then add this group as a user in a database. Next, create roles in a database corresponding to various job functions, and assign database users to the appropriate role. Finally, assign security permissions on objects in the database to the database role. It is assumed that for all examples regarding security, you are implementing security best practices.

There are seven permissions that you can assign for a table, as listed in Table 3-9.

Table 3-9 Table Permissions

Permission	Purpose
CREATE TABLE	Gives the authority to create any table in the database.
ALTER TABLE	Gives the authority to change the structure of any table in the database.
SELECT	Allows rows to be retrieved from the specified table.
INSERT	Allows rows to be inserted in a specified table. Requires the *SELECT* permission to be granted as well.
UPDATE	Allows rows to be modified in a specified table. Requires the *SELECT* permission to be granted as well.
DELETE	Allows rows to be deleted from a specified table. Requires the *SELECT* permission to be granted as well.
REFERENCES	Used with foreign key constraints; to be discussed in the next lesson.

You can use the special keyword *ALL* to grant every permission shown in the table to a specified role. However, you should always explicitly list each permission that you will allow. The general statement to assign permissions is the following:

```
GRANT { ALL [ PRIVILEGES ] }
      | permission [ ( column [ ,...n ] ) ] [ ,...n ]
      [ ON [ class :: ] securable ] TO principal [ ,...n ]
      [ WITH GRANT OPTION ] [ AS principal ]
```

The *ON* clause specifies the object that you are granting permission to, whereas the *TO* clause specifies the database role the permissions are assigned to.

For tables, it is possible to grant permissions on a subset of the columns in the table. There is no facility to grant permissions to a subset of rows in a table.

The *WITH GRANT* option enables you to grant permissions to a role whose members can then grant permissions to other users or roles. You should never use this option because it takes control of security out of the hands of the owner of the table.

For the *CustomerAddress* table, the command to grant *SELECT*, *INSERT*, *UPDATE*, and *DELETE* permissions to a role is as follows:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON CustomerAddress TO <database role>
```

PRACTICE Create a Table

In this practice, you will create three additional tables—*Customer*, *StateProvince*, and *Country*—for use with the *CustomerAddress* table we created in this lesson.

NOTE If you didn't create the *CustomerAddress* table

The instructions for creating the CustomerAddress table are earlier in this lesson, under the heading "Permanent Tables."

The *Customer* table will contain the customer name, a value for the customer's credit line, a value for the customer's outstanding balance, a computation for available credit, and the date the customer record was created. The *StateProvince* table will contain a text-based column that will store a list of the valid states or provinces recognized by this company. The *Country* table will contain a text-based column that will store a list of the valid countries. Remember to create a column to reference each of the rows the same way we did with the *CustomerAddress* table.

NOTE Database context

This practice can be done in either the *AdventureWorks* database or another database of your choice.

1. Launch SQL Server Management Studio (SSMS), connect to your instance, and then open a new query window.
2. Construct a *CREATE TABLE* statement for the *Customer* table as follows:

```
CREATE TABLE dbo.Customer
(CustomerID               int            IDENTITY(1,1),
CustomerName              varchar(50)    NOT NULL,
CreditLine                smallmoney     NULL,
OutstandingBalance        smallmoney     NULL,
AvailableCredit AS (CreditLine - OutstandingBalance),
CreationDate              datetime       NOT NULL)
```

3. Construct a *CREATE TABLE* statement for the *StateProvince* table as follows:

```
CREATE TABLE dbo.StateProvince
(StateProvinceID          int            IDENTITY(1,1),
StateProvince             varchar(50)    NOT NULL)
```

4. Construct a *CREATE TABLE* statement for the *Country* table, as follows:

```
CREATE TABLE dbo.Country
(CountryID          int           IDENTITY(1,1),
Country             varchar(50)   NOT NULL)
```

Lesson Summary

- Tables, the building blocks for every database, store all the data in SQL Server.
- To provide the necessary structure to a table, you must choose between the available *numeric*, *text*, *datetime*, and *binary* data types so that data can be properly stored.
- You can also define special properties for columns to allow *nulls*, define a column as a unique identifier column, and allow a column to store a computation or computed data.
- After a table is defined, you must grant permissions on the table to allow users to retrieve and manipulate data.

Lesson Review

The following questions are intended to reinforce key information presented in this lesson. The questions are also available on the companion CD if you prefer to review them in electronic form.

NOTE Answers

Answers to these questions and explanations of why each answer choice is right or wrong are located in the "Answers" section at the end of the book.

---

1. Which data type would you use to store up to 2 GB of text data and still be able to query and manipulate it by using standard functions and operators?
   - **A.** *text*
   - **B.** *varbinary*
   - **C.** *varchar*(max)
   - **D.** *varchar*

Lesson 2: Implementing Constraints

Designing a database is really an exercise in implementing business rules. You might not have realized it, but the entire first lesson implemented a variety of business rules. For example, in Lesson 1, we implemented a business rule stating that a customer can have more than one address, but an address is not valid unless there is at least one address line and a city.

Constraints provide a second level of business-rule implementation by preventing users from entering data into tables that is outside the allowed boundaries. Examples of this type of business rule include one that prohibits a customer's credit line from exceeding $50,000 and one that prevents users from entering countries that do not exist in a standardized list.

This lesson explains the six types of constraints that you can create to enforce business rules and shares best practices for when to implement each type of constraint.

After this lesson, you will be able to:

- Implement constraints.
- Specify the scope of a constraint.
- Create a new constraint.

Estimated lesson time: 20 minutes

Check Constraints

You use check constraints to limit the range of possible values in a column or to enforce specific patterns for data. All check constraints must evaluate to a Boolean True/False and cannot reference columns in another table.

You can create check constraints at two different levels:

- *Column-level* check constraints are applied only to the column and cannot reference data in another other column.
- *Table-level* check constraints can reference any column within a table but cannot reference columns in other tables.

The most basic constraint compares the data in a column to a specified value—for example, *CHECK CreditLine <= 50000*. You can create any number of check constraints separated by *AND*, *OR*, or *NOT* to create more complex conditions.

You can also use check constraints to enforce patterns within data. Using a check constraint this way, you might enforce the pattern that an EmployeeID is required to start with an uppercase letter, followed by three digits and then six additional letters. Another example is to require an e-mail address to contain, in order, any number of characters or digits, an @ symbol, a number of characters or digits, a period (.), and then either three characters or two characters with a period (.) plus two more characters.

The wildcard characters for pattern matching are the underscore (_), which designates one value that can be a character, number, or special character; and a percent symbol (%), which designates any number of characters, numbers, or special characters. For example, a table-level check constraint to validate an e-mail address might look like this:

```
CONSTRAINT chkEmail CHECK (Email like '%@%.[a-z][a-z][a-z]' or Email like '%@%.[a-z]
[a-z].[a-z][a-z]')
```

A column-level check constraint for the EmployeeID looks like this:

```
CHECK (EmployeeID like '[A-Z][0-9][0-9][0-9][A-Z][A-Z][A-Z][A-Z][A-Z][A-Z]')
```

MORE INFO Constraints and pattern matching

Creating pattern matches can become complex. For more information about the allowed operators and wildcards, see the SQL Server 2005 Books Online topics "CHECK Constraints" and "CREATE RULE (Transact-SQL)."

Rules

You define check constraints within the table definition and cannot reuse them. Rules provide the same functionality as check constraints, except that you create them as a separate object.

Because rules are not associated with a specific table or column when you create them, they cannot reference columns or tables in their definition. Instead, you use variables as placeholders. Rules provide the same features and complex comparisons via *AND*, *OR*, and *NOT* as check constraints and allow pattern matching.

The following examples show the previous two check constraints implemented as rules:

```
CREATE RULE EmailValidator
AS
@value like '%@%.[a-z][a-z][a-z]' or @value like '%@%.[a-z][a-z].[a-z][a-z]';
```

```
CREATE RULE EmployeeIDValidator
AS
@column like '[A-Z][0-9][0-9][0-9][A-Z][A-Z][A-Z][A-Z][A-Z][A-Z]';
```

After defining a rule, you then bind it to columns or user-defined data types by using the *sp_bindrule* system stored procedure.

MORE INFO Binding rules

For complete information about binding rules to columns or user-defined data types, see the SQL Server 2005 Books Online article "CREATE RULE (Transact-SQL)."

Default Constraints

Another mechanism for enforcing a business rule in a table is a default constraint, which enables SQL Server to write a value to a column when the user doesn't specify a value. Common uses for a default constraint are when a "typical" value or very "common" value exists for a column, but that value is not necessarily the only possible choice. For example, let's say the company we have been creating tables for is a retail store located in Grand Prairie, TX. Most customers have an address with a city of Grand Prairie. However, customers might still come into the store from nearby Arlington or Irving.

You can add a default constraint to the City column in the *CustomerAddress* table using the following example:

```
CREATE TABLE dbo.CustomerAddress
(CustomerAddressID          int             IDENTITY(1,1),
AddressType                 char(4)         NOT NULL,
PrimaryAddressFlag          bit             NOT NULL,
AddressLine1                varchar(30)     NOT NULL,
AddressLine2                varchar(30)     NULL,
AddressLine3                varchar(30)     NULL,
City                        varchar(50)     NOT NULL DEFAULT 'Grand Prairie',
StateProvinceID             int             NULL,
PostalCode                  char(10)        NULL,
CountryID                   int             NULL)
```

Unique Constraints

A unique constraint prohibits a column or combination of columns from allowing duplicate values. You might use a unique constraint to enforce a business rule stating that each customer name must be unique.

You can add a unique constraint to the CustomerName column in the *Customer* table by using the following:

```
CREATE TABLE dbo.Customer
(CustomerID                int            IDENTITY(1,1),
CustomerName               varchar(50)    NOT NULL UNIQUE NONCLUSTERED,
CreditLine                 smallmoney     NULL,
OutstandingBalance         smallmoney     NULL,
AvailableCredit AS (CreditLine - OutstandingBalance),
CreationDate               datetime       NOT NULL)
```

NOTE Clustered and nonclustered indexes

A unique constraint is physically implemented in the database as a unique index. Indexes can be either clustered or nonclustered. Within this chapter, we are explicitly avoiding the discussion of indexes, including clustered and nonclustered indexes. Chapter 4, "Creating Indexes," covers these topics in detail.

Primary Key Constraints

Your choice of primary key constraint is critical in creating a sound structure for a table. A *primary key* defines the column or combination of columns that allow a row to be uniquely identified.

MORE INFO Primary key choice

Choosing the columns for a primary key is beyond the scope of this book, as is the discussion of whether a primary key should have business meaning or be implemented as an internal database structure. For details on these topics, see *MCITP Self-Paced Training Kit (Exam 70-443): Designing a Database Server Infrastructure by Using Microsoft SQL Server 2005*, Microsoft Press, 2007.

You implement a primary key on the StateProvinceID column of the *StateProvince* table as follows:

```
CREATE TABLE dbo.StateProvince
(StateProvinceID           int            IDENTITY(1,1)  PRIMARY KEY,
StateProvince              varchar(50)    NOT NULL)
```

Foreign Key Constraints

You use foreign key constraints to implement a concept called referential integrity. *Foreign keys* ensure that the values that can be entered in a particular column exist in a specified table. Users cannot enter values in this column that do not exist in the specified table.

For example, the *CustomerAddress* table should be allowed to specify only valid values for the StateProvince column. Providing a valid list of states and provinces for a user to select from and enforcing the range of available values ensures that data is not only consistent but also valid.

To enforce referential integrity on the StateProvince column in the *CustomerAddress* table, you could use the following code, which uses the *REFERENCES* keyword:

```
CREATE TABLE dbo.CustomerAddress
(CustomerAddressID         int            IDENTITY(1,1),
AddressType                char(4)        NOT NULL,
PrimaryAddressFlag         bit            NOT NULL,
AddressLine1               varchar(30)    NOT NULL,
AddressLine2               varchar(30)    NULL,
AddressLine3               varchar(30)    NULL,
City                       varchar(50)    NOT NULL DEFAULT 'Grand Prairie',
StateProvinceID            int            NULL REFERENCES dbo.StateProvince(StateProvinceID),
PostalCode                 char(10)       NULL,
CountryID                  int            NULL)
```

Or you could use the following code, which uses the *FOREIGN KEY* keyword:

```
CREATE TABLE dbo.CustomerAddress
(CustomerAddressID         int            IDENTITY(1,1),
AddressType                char(4)        NOT NULL,
PrimaryAddressFlag         bit            NOT NULL,
AddressLine1               varchar(30)    NOT NULL,
AddressLine2               varchar(30)    NULL,
AddressLine3               varchar(30)    NULL,
City                       varchar(50)    NOT NULL DEFAULT 'Grand Prairie',
StateProvinceID            int            NULL FOREIGN KEY (StateProvinceID)
 REFERENCES dbo.StateProvince(StateProvinceID),
PostalCode                 char(10)       NULL,
CountryID                  int            NULL)
```

When you add a foreign key to a table, it not only enforces the values that can be used in a column but it also enforces a dependency chain. You cannot drop a foreign key table unless you do one of the following first:

- Drop the table that references it.
- Remove the foreign key constraint with an *ALTER TABLE* statement.

For example, you could not drop the *StateProvince* table without either dropping the *CustomerAddress* table first or removing the foreign key constraint from the *CustomerAddress* table.

IMPORTANT Referencing tables

For a foreign key to work, it must be able to uniquely identify each row in the referenced table. Therefore, you must create a primary key on the column that is used to enforce referential integrity.

Foreign Keys vs. Check Constraints

A foreign key constraint is really nothing more than a check constraint with a list of allowed values. So the question becomes, when should you use a check constraint and when should you use a foreign key?

You should use check constraints when you need to validate patterns, perform calculations to compare against, or use comparison operators such as >, <, >=, and so on.

You should always use foreign keys when you need to validate the column against a list of acceptable values. Even if the list contains only one or two values, you should still implement it as a foreign key.

If you implement a list validation as a check constraint, whenever you want to add a new value to the list, you have to modify the table structure by using an *ALTER TABLE* command. By implementing the list as a foreign key, you simply insert the new value into the table.

Using a foreign key for list validation also leads to a maintainable design. When a database is initially designed, you might not know the list of acceptable values. Or the list might be completely valid at the time it was created, but five years later, the list of valid values might have changed. Application developers can easily add a maintenance screen into an application to allow one or more designated users to modify the list of allowed values, and the foreign key constraint prevents a value from being removed from the table if it has been used. Adding a new value to the table then becomes a simple action performed by a user instead of becoming a request to the database administrator (DBA) team, as would happen if the list were in a check constraint.

Quick Check

- What are the six types of constraints, and what purpose does each serve?

Quick Check Answer

- *Check constraints* restrict the allowable values in a column.
- *Rules* implement the same functionality as check constraints but are implemented as objects separate from a specific table, so a rule can be created once and used in many places.
- A *default constraint* causes a value to be entered into a column when one is not specified by a user.
- A *unique constaint* ensures that duplicate values do not exist in a column or combination of columns.
- A *primary key* ensures that each row in a table can be uniquely identified by the column or a combination of specified columns. Only one primary key can exist on a table, whereas multiple unique constraints can be created.
- A *foreign key* forces a column to allow only values that exist in a referenced table.

PRACTICE Implement Constraints

In this practice, you will apply a variety of constraints to the *Customer*, *CustomerAddress*, *StateProvince*, and *Country* tables so that they more closely match what you would see in an actual production environment.

1. If necessary, launch SSMS, connect to your instance, and open a new query window.
2. Before you begin this exercise, drop all the tables that you created previously by using the following batch:

```
DROP TABLE dbo.CustomerAddress;
DROP TABLE dbo.Customer;
DROP TABLE dbo.Country;
DROP TABLE dbo.StateProvince;
```

NOTE Errors

If you receive any errors when executing the preceding batch, you can ignore them. Any error you might receive will say something like "could not drop table because it does not exist." Chapter 9, "Creating Functions, Stored Procedures, and Triggers," explains how to write batches that contain error checking and handling.

3. Re-create the *Country* and *StateProvince* tables with primary keys, as follows:

```
CREATE TABLE dbo.StateProvince
(StateProvinceID          int           IDENTITY(1,1)  PRIMARY KEY CLUSTERED,
StateProvince             varchar(50)   NOT NULL);

CREATE TABLE dbo.Country
(CountryID          int           IDENTITY(1,1) PRIMARY KEY CLUSTERED,
Country             varchar(50)   NOT NULL);
```

4. Create a new table for the list of allowed address types, as follows:

```
CREATE TABLE dbo.AddressType
(AddressTypeID      tinyint       IDENTITY(1,1) PRIMARY KEY CLUSTERED,
AddressType         varchar(20)   NOT NULL);
```

5. Create the *CustomerAddress* table with a primary key and enforce referential integrity for the StateProvinceID, CountryID, and AddressType columns, as follows:

```
CREATE TABLE dbo.CustomerAddress
(CustomerAddressID        int           IDENTITY(1,1) PRIMARY KEY CLUSTERED,
AddressType               char(4)       NOT NULL FOREIGN KEY (AddressType) REFERENCE
S dbo.AddressType(AddressTypeID),
PrimaryAddressFlag        bit           NOT NULL,
AddressLine1              varchar(30)   NOT NULL,
AddressLine2              varchar(30)   NULL,
AddressLine3              varchar(30)   NULL,
City                      varchar(50)   NOT NULL,
StateProvinceID           int           NULL FOREIGN KEY (StateProvinceID) REFERENCE
S dbo.StateProvince(StateProvinceID),
PostalCode                char(10)      NULL,
CountryID                 int           NULL FOREIGN KEY (CountryID) REFERENCES dbo.
Country(CountryID));
```

NOTE Data type mismatches

You should have received an error message when trying to create this table. Before reading on, can you explain why?

The AddressType column is defined as a *char*(4), but the foreign key references an integer column in the *AddressType* table. A character value cannot be implicitly converted to a *tinyint* for comparison. Although the column name in the *CustomerAddress* table does not have to match the column name in the *AddressType* table, the data types must be compatible. However, for consistency and readability, the columns names should match.

---

6. Fix the error by redefining the *CustomerAddress* table, as follows:

```
CREATE TABLE dbo.CustomerAddress
(CustomerAddressID        int           IDENTITY(1,1) PRIMARY KEY CLUSTERED,
AddressTypeID             tinyint       NOT NULL FOREIGN KEY (AddressTypeID) REFEREN
CES dbo.AddressType(AddressTypeID),
PrimaryAddressFlag        bit           NOT NULL,
AddressLine1              varchar(30)   NOT NULL,
```

```
AddressLine2              varchar(30)    NULL,
AddressLine3              varchar(30)    NULL,
City                      varchar(50)    NOT NULL,
StateProvinceID           int            NULL FOREIGN KEY (StateProvinceID)
 REFERENCES dbo.StateProvince(StateProvinceID),
PostalCode                char(10)       NULL,
CountryID                 int            NULL FOREIGN KEY (CountryID) REFERENCES
 dbo.Country(CountryID));
```

7. Create the *Customer* table with a primary key, enforcing no duplicate customer names and a credit line between 0 and 50,000. Default the available balance to 0, and default the creation date to the current date and time, as follows:

```
CREATE TABLE dbo.Customer
(CustomerID               int            IDENTITY(1,1) PRIMARY KEY CLUSTERED,
CustomerName              varchar(50)    NOT NULL UNIQUE NONCLUSTERED,
CreditLine                smallmoney     NULL CHECK (CreditLine >= 0 AND CreditLine <
= 50000),
OutstandingBalance        smallmoney     NULL DEFAULT 0,
AvailableCredit AS (CreditLine - OutstandingBalance),
CreationDate              datetime       NOT NULL DEFAULT getdate());
```

8. Our customer minidatabase is looking pretty good at this point, but there is one problem. Customers can be entered, and addresses can be entered, but there is no way to associate a customer to an address. Create a table that provides an association between the *Customer* and *CustomerAddress* tables, as follows:

```
CREATE TABLE dbo.CustomerToCustomerAddress
(CustomerID               int      NOT NULL FOREIGN KEY (CustomerID) REFERENCES
 dbo.Customer(CustomerID),
CustomerAddressID         int      NOT NULL FOREIGN KEY (CustomerAddressID)
 REFERENCES dbo.CustomerAddress(CustomerAddressID),
CONSTRAINT PK_CustomerToCustomerAddress PRIMARY KEY CLUSTERED(CustomerID,
 CustomerAddressID));
```

NOTE Cross-reference tables

The *CustomerToCustomerAddress* table is generally referred to as a *cross-reference* table. You could have linked the *Customer* and *CustomerAddress* tables together by adding a CustomerID column to the *CustomerAddress* table. However, the cross-reference table allows flexibility in the design and minimizes the amount of data that needs to be stored. For example, you could have multiple customers at the same address, such as with multiple people in the same household. If the CustomerID column were added to the *CustomerAddress* table, each customer at the same address would require you to duplicate the address in the *CustomerAddress* table. However, the cross-reference table allows you to associate a single row in the *CustomerAddress* table with one or more customers. The opposite is also true: you can associate a single customer with multiple addresses.

Lesson Summary

- You use constraints to enforce additional business rules within a table.
- You can use constraints to ensure that duplicate values cannot be entered into a column or that a column can allow only values that meet a specified condition.
- You can use constraints to enforce complex pattern matching such as the Vehicle Identification Number (VIN) that is used to uniquely identify every vehicle.
- You can also create constraints to ensure that a value cannot be entered in one table unless it already exists in another table, for example, not allowing an address to be entered unless a customer already exists for the address.

Lesson Review

The following questions are intended to reinforce key information presented in this lesson. The questions are also available on the companion CD if you prefer to review them in electronic form.

NOTE Answers

Answers to these questions and explanations of why each answer choice is right or wrong are located in the "Answers" section at the end of the book.

---

1. Which of the following objects can you use in a check constraint? (Choose all that apply.)
   - **A.** System function
   - **B.** Stored procedure
   - **C.** User-defined function (UDF)
   - **D.** View