Every time I teach the T-SQL code for creating databases, tables, keys, and constraints, I am asked the same question, “Can’t you just do this in the GUI tool?” The answer is an unequivocal “Yes!” Therefore, the next question usually follows quite shortly behind, “Then why are we spending all this time learning stuff I’ll never use?” The answer is just as unequivocal — you will use the regular syntax on a quasi-regular basis. The reality is you probably won’t actually write the code from scratch that often, but you’ll verify and edit it on the majority of all larger database projects you work on — that means that you had better know how it works.

In this chapter, you’ll study the syntax for creating your own tables. You’ll also take a look at how to make use of the SQL Management Studio to help with this (after you know how to do it for yourself).

However, before you learn too much about the actual statements that create tables and other objects, I need to digress far enough to deal with the convention for a fully qualified object name and, to a lesser extent, object ownership.
OBJECT NAMES IN SQL SERVER

In all the queries that you’ve been performing so far in this book, you’ve seen simple naming at work. I’ve had you switch the active database in the Query Analyzer before running any queries, and that has helped your queries to work. How? Well, SQL Server looks at only a very narrow scope when trying to identify and locate the objects you name in your queries and other statements. For example, you’ve been providing only the names of tables without any additional information, but there are actually four levels in the naming convention for any SQL Server table (and any other SQL Server object for that matter). A fully qualified name is as follows:

\[\text{[ServerName.[DatabaseName.[SchemaName.]]ObjectName]}\]

You must provide an object name whenever you are performing an operation on that object, but all parts of the name to the left of the object name are optional. Indeed, most of the time, they are not needed and are therefore left off. Still, before you start creating objects, it’s a good idea to get a solid handle on each part of the name. So let’s move from the object name left.

Schema Name (aka Ownership)

If you want to separate your database objects into logical groups, you can create schemas to house them. For small databases this is rarely necessary, but for more complicated systems it’s sometimes desirable either from an organizational perspective or to simplify security. Either way, it’s completely up to the architect whether you’re going to use them.

If you’re utilizing schemas (most older databases do not, but their use on newer databases is somewhat more common), you may need to indicate which schema your object is in. It is entirely possible to have two objects with the same name that reside in different schemas. If you want to access an object that is not in your default schema (set on a login-by-login basis), you’ll need to specifically state the schema name of your object. For example, let’s look at what has to be one of the worst uses of schemas I’ve ever seen — the AdventureWorks database you’ve already been using — and take a look at a query that gets a list of employees and what city they live in:

```
SELECT e.NationalIDNumber, p.FirstName, p.LastName, City
FROM HumanResources.Employee e
INNER JOIN Person.Address pa on pa.AddressID = a.AddressID
```

This example makes use of four tables spread across two schemas. If one of the two schemas involved — HumanResources and Person — happened to be the default schema, you could have left that schema name off when naming tables in that schema. In this case, all schemas were specifically named to be on the safe side.
NOTE This is another time I’m going to get on the consistency soapbox. If you’re going to use the schema features at all, I highly recommend using two-part naming (schema and table name) in all of your queries. It is far too easy for a change to be made to a user’s default schema or to some other alias that would make assumptions about the default invalid. If you’re not utilizing different schemas in your database design, it’s fine to leave them off (and make your code a fair amount more readable in the process), but keep in mind there may be a price to pay if later you start using schemas and suddenly have to update all your old code to two-part naming.

A Little More About Schemas

The ANSI/ISO standard for SQL has had the notion of what has been called a schema for quite some time now. SQL Server has had that same concept in place all along, but used to refer to it differently (and, indeed, had a different intent for it even if it could be used the same way). So, what you see referred to in SQL Server 2012 and other databases such as Oracle as “schema” was usually referred to as “owner” in SQL Server 2000 and prior.

The notion of the schema used to be a sticky one. While the problems of schema are still non-trivial, Microsoft has added some new twists to make them much easier to deal with. If, however, you need to deal with backward compatibility to prior versions of SQL Server, you'll need to either avoid the new features or use pretty much every trick they have to offer — and that means ownership (as it was known in prior versions) remains a significant hassle.

There were always some people who liked using ownership in their older designs, but I was definitely not one of them. For now, the main thing to know is that what is now “schema” is something that overlaps with an older concept called “ownership,” and you may see both terms in use. Schema also becomes important in dealing with some other facets of SQL Server such as Notification Services.

Let’s focus, for now, on what a schema is and how it works.

For prior releases, ownership (as it was known then) was actually a great deal like what it sounds — it was recognition, right within the fully qualified name, of who “owned” the object. Usually, this was either the person who created the object or the database owner (more commonly referred to as the dbo — I’ll get to describing the dbo shortly). Things still work in a similar fashion, but the object is assigned to a schema rather than an owner. Whereas an owner related to one particular login, a schema can now be shared across multiple logins, and one login can have rights to multiple schemas. The schema feature in SQL Server 2012 is now purely organizational and shouldn’t be confused with ownership any longer. Users can still own schemas or have a default schema, but the schema is not the same thing as the owner.

By default, only users who are members of the sysadmin system role, or the db_owner or db_ddladmin database roles, can create objects in a database.
Individual users can also be given the right to create certain types of database and system objects. If such individuals do indeed create an object, then, by default, that object will be assigned to whatever schema is listed as default for that login.

**NOTE** The roles mentioned here are just a few of many system and database roles that are available in SQL Server 2012. Roles have a logical set of permissions granted to them according to how that role might be used. When you assign a particular role to someone, you are giving that person the ability to have all the permissions that the role has.

The Default Schema: dbo

Whoever creates the database is considered to be the “database owner,” or dbo. Any objects that a dbo creates within that database shall be listed with a schema of dbo rather than that person’s individual username.

To explain, let me begin with a counterexample. Say that I am an everyday user of a database, my login name is MySchema, and I have been granted CREATE TABLE authority to a given database. If I create a table called MyTable, the owner-qualified object name would be MySchema.MyTable. Note that, because the table has a specific owner, any user other than me (remember, I’m MySchema here) of MySchema.MyTable would need to provide the two part (schema-qualified) name in order for SQL Server to resolve the table name.

To continue, another user whose login is Group1Member is using the system. Group1Member is a member of a Windows domain group called Group1, which has a default schema of Group1. Using exactly the same CREATE TABLE script as I used (under MySchema), Group1Member creates a new table. The new table is called Group1.MyTable, and is distinct from MySchema.MyTable.

**NOTE** Just because a feature is there doesn’t mean it should be used! Giving CREATE authority to individual users is nothing short of nightmarish. Trying to keep track of who created what, when, and for what reason becomes nearly impossible. In short, keep CREATE access limited to the members of the sysadmins or db_owner security roles.

Now, let’s say that there is also a user with a login name of Fred. Fred is the database owner (as opposed to just any member of db_owner). If Fred creates a table called MyTable using an identical CREATE statement to that used by MySchema and Group1Member, the two-part table name will be dbo.MyTable. In addition, as dbo also happens to be the default owner, any user could just refer to Fred’s table as MyTable.

**NOTE** Until SQL Server 2012, a Windows group could not be assigned a default schema, so don’t look for that in previous versions. Use this feature when you want a schema to follow group membership rather than individual users.
It’s worth pointing out that members of the sysadmin role (including the sa login) always alias to the dbo. That is, no matter who actually owns the database, a member of sysadmin will always have full access as if it were the dbo, and any objects created by a member of sysadmin will, unless explicitly defined otherwise, show ownership belonging to the dbo. In contrast, objects created by members of the db_owner database role do not default to dbo as the default schema — they will be assigned to whatever that particular user has set as the default schema (it could be anything). Weird but true!

NOTE Microsoft keeps making the use of schemas easier and more consistent, and that’s perfectly in keeping with both ANSI standards and user requests (including mine). Anything that makes a feature easier to deal with is okay in my book, but it doesn’t change the considerations about when you’d actually want to use different schemas in your design. Rather, it just lowers the price of admission.

The addition of schemas adds complexity to your database no matter what you do. While they can address organizational problems in your design, those problems can usually be dealt with in other ways that produce a much more user-friendly database. In addition, schemas, while an ANSI/ISO-compliant notion, are not supported in the same way across every major RDBMS product. This means using schemas is going to have an impact on you if you’re trying to write code that can support multiple platforms.

The Database Name

The next item in the fully qualified naming convention is the database name. Sometimes you want to retrieve data from a database other than the default, or current, database. Indeed, you may actually want to JOIN data from across databases. A database-qualified name gives you that ability. For example, if you were logged in with AdventureWorks as your current database, and you wanted to refer to the Orders table in the Accounting database you’ll be building later in the chapter, you could refer to it by Accounting.dbo.Orders. Because dbo is the default schema, you could also use Accounting..Orders. If a schema named MySchema owns a table named MyTable in MyDatabase, you could refer to that table as MyDatabase.MySchema.MyTable. Remember that the current database (as determined by the USE command or in the dropdown box if you’re using the SQL Server Management Studio) is always the default, so, if you want data from only the current database, you do not need to include the database name in your fully qualified name.

Naming by Server

In addition to naming other databases on the server you’re connected to, you can also “link” to another server. Linked servers give you the capability to perform a JOIN across multiple servers — even different types of servers (SQL Server, Oracle, DB2, Access — just about anything with an OLE DB provider). You’ll see a bit more about linked servers later in the book, but for now, just realize that there is one more level in the naming hierarchy, that it lets you access different servers, and that it works pretty much like the database and ownership levels work.

Now, let’s add to the previous example. While creating a linked server is a bit advanced (you’ll see that in Chapter 21), let’s take it as read that there’s a linked server called MyServer. If you want to retrieve
information from that server, from a database called MyDatabase, and a table called MyTable owned by MySchema, the fully qualified name would be MyServer.MyDatabase.MySchema.MyTable.

Reviewing the Defaults

So let’s look one last time at how the defaults work at each level of the naming hierarchy from right to left:

- **Object Name:** There isn’t a default — you must supply an object name.
- **Ownership:** You can leave this off, in which case it will resolve first using the current user’s name, and then, if the object name in question doesn’t exist with the current user as owner, it will try the dbo as the owner.
- **Database Name:** This can also be left off unless you are providing a Server Name — in which case you must provide the Database Name for SQL Servers (other server types vary depending on the specific kind of server).
- **Server Name:** You can provide the name of a linked server here, but most of the time you’ll just leave this off, which will cause SQL Server to default to the server you are logged in to.

If you want to skip the schema name, but still provide information regarding the database or server, you must still provide the extra “.” for the position where the owner would be. For example, if you are logged in using the AdventureWorks database on your local server, but want to refer to the Sales.Customer table in the AdventureWorks database on a linked server called MyOtherServer, you could refer to that table by using MyOtherServer.AdventureWorks..Customer. Because you didn’t provide a specific schema name, it will assume that either the default schema for the user ID that is used to log on to the linked server or the dbo (in that order) is the schema of the object you want (in this case, Customer). Since the Customer table is not part of the dbo schema, the users would need to have a default schema of Sales or they would get an error that the object was not found. In general, I recommend explicitly naming the schema of the object you want to reference.

THE CREATE STATEMENT

It would be pretty nifty if you could just wave a wand, mumble some pseudo-Latin, and have your database spring into being exactly as you want it to. Unfortunately, creating things isn’t quite that simple. You need to provide a well-defined syntax in order to create the objects in your database (and any wand-waving is entirely optional). To do that, you’ll make use of the CREATE statement.

Let’s look at the full structure of a CREATE statement, starting with the utmost in generality. You’ll find that all the CREATE statements start out the same, and then get into the specifics. The first part of the CREATE statement will always look like this:

```
CREATE <object type> <object name>
```

This will be followed by the details that vary by the nature of the object you’re creating.

CREATE DATABASE

For this part of things, I’m going to introduce you to a database called Accounting that you will use when you start to create tables. You and Accounting are going to become acquainted over the next
several chapters, and you can start by creating that database here. The most basic syntax for the
CREATE DATABASE statement looks like this:

```
CREATE DATABASE <database name>
```

**NOTE** It's worth pointing out that when you create a new object, no one can
access it except for the person who created it, the system administrator, and the
database owner (which, if the object created was a database, is the same as the
person who created it). This allows you to create things and make whatever
adjustments you need to make before you explicitly allow access to your object.

It's also worth noting that you can use the CREATE statement only to create
objects on the local server (adding a specific server name doesn't work).

This yields a database that looks exactly like your model database (the model database was discussed
in Chapter 1). In reality, what you want is almost always different, so let’s look at a fuller (though
still not entirely complete) syntax listing:

```
CREATE DATABASE <database name>
[ ON [PRIMARY]
  ([NAME = '<logical file name'>,]
   FILENAME = '<file name'>
   [, SIZE = <size in kilobytes, megabytes, gigabytes, or terabytes>]
   [, MAXSIZE = size in kilobytes, megabytes, gigabytes, or terabytes>
   [, FILEGROWTH = <kilobytes, megabytes, gigabytes, or terabytes|percentage>]])
[ LOG ON
  ([NAME = '<logical file name'>,]
   FILENAME = '<file name'>
   [, SIZE = <size in kilobytes, megabytes, gigabytes, or terabytes>]
   [, MAXSIZE = size in kilobytes, megabytes, gigabytes, or terabytes>
   [, FILEGROWTH = <kilobytes, megabytes, gigabytes, or terabytes|percentage>]])
[ CONTAINMENT = OFF|PARTIAL ]
[ COLLATE <collation name> ]
[ FOR ATTACH [WITH <service broker>] [ FOR ATTACH_REBUILD_LOG]
  WITH DB_CHAINING ON|OFF | TRUSTWORTHY ON|OFF
  [AS SNAPSHOT OF <source database name>]
[;]
```

Keep in mind that some of the preceding options are mutually exclusive (for example, if you're creating
for attaching, most of the options other than file locations are invalid). There’s a lot there, so let’s
break down the parts.

**CONTAINMENT**

Contained databases are a new feature in SQL Server 2012. They allow you to deploy a database
with fewer dependencies on the target SQL instance. For example, you can assign permissions to a
Windows user without creating a SQL Server login for that user, granting you more independence
when it comes time to move your database from server to server. Because this is a fairly advanced concept, it’s beyond the scope of this book to cover it in its entirety. Suffice it to say it’s out there and that the default setting for CONTAINMENT is OFF.

**ON**

ON is used in two places: to define the location of the file where the data is stored, and to define the same information for where the log is stored. You’ll notice the PRIMARY keyword there — this means that what follows is the primary (or main) filegroup in which to physically store the data. You can also store data in what are called secondary filegroups — the use of which is outside the scope of this title. For now, stick with the default notion that you want everything in one file.

**NOTE**

SQL Server allows you to store your database in multiple files; furthermore, it allows you to collect those files into logical groupings called filegroups. The use of filegroups is a fairly advanced concept and is outside the scope of this book.

**NAME**

This one isn’t quite what it sounds like. It is a name for the file you are defining, but only a logical name — that is, the name that SQL Server will use internally to refer to that file. You use this name when you want to resize (expand or shrink) the database and/or file.

**FILENAME**

This one is what it sounds like — the physical name on the disk of the actual operating system file in which the data and log (depending on what section you’re defining) will be stored. The default here (assuming you used the simple syntax you looked at first) depends on whether you are dealing with the database itself or the log. By default, your file will be located in the \Data subdirectory under your main \Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER\MSSQL directory (or whatever you called your main SQL Server directory if you changed it at install). If you’re dealing with the physical database file, it will be named the same as your database with an .mdf extension. If you’re dealing with the log, it will be named the same as the database file but with a suffix of _Log and an .ldf extension. You are allowed to specify other extensions if you explicitly name the files, but I strongly encourage you to stick with the defaults of mdf (database) and ldf (log file). As a side note, secondary files have a default extension of .ndf.

Keep in mind that while FILENAME is an optional parameter, it is optional only as long as you go with the extremely simple syntax (the one that creates a new database based on the model database) that I introduced first. If you provide any of the additional information, you must include an explicit file name — be sure to provide a full path.

**SIZE**

No mystery here. It is what it says — the size of the database. By default, the size is in megabytes, but you can make it kilobytes by using KB instead of MB after the numeric value for the size, or go bigger.
by using GB (gigabytes) or even TB (terabytes). Keep in mind that this value must be at least as large as
the model database is and must be a whole number (no decimals) or you will receive an error. If you do
not supply a value for `SIZE`, the database will initially be the same size as the model database.

**MAXSIZE**

This one is still pretty much what it sounds like, with only a slight twist compared to the `SIZE`
parameter. SQL Server has a mechanism to allow your database to automatically allocate additional
disk space (to grow) when necessary. `MAXSIZE` is the maximum size to which the database can grow.
Again, the number is, by default, in megabytes, but like `SIZE`, you can use KB, GB, or TB to use
different increment amounts. The slight twist is that there is no firm default. If you don’t supply a
value for this parameter, there is considered to be no maximum — the practical maximum becomes
when your disk drive is full.

If your database reaches the value set in the `MAXSIZE` parameter, your users will start getting errors
back saying that their `INSERT`s can’t be performed. If your log reaches its maximum size, you will
not be able to perform any logged activity (which is most activities) in the database. Personally, I
recommend setting up what is called an *alert*. You can use alerts to tell you when certain conditions
exist (such as a database or log that’s almost full). You’ll see how to create alerts in Chapter 21.

**WHAT VALUE SHOULD MAXSIZE BE?**

I recommend that you always include a value for `MAXSIZE`, and that you make it at
least several megabytes smaller than would fill up the disk. I suggest this because a
completely full disk can cause situations where you can’t commit any information
to permanent storage. If the log was trying to expand, the results could potentially
be disastrous. In addition, even the operating system can occasionally have prob-
lems if it runs completely out of disk space — be aware of this if your file shares a
disk with your OS.

This brings up a larger point about disk usage. You’ll hear more from me later on
this topic when you read about performance, but the disk is your biggest bottleneck
most of the time. You should place your files wisely for more reasons than just filling
space. Best practices suggest that, if at all possible, you should keep your database
files on a separate volume than your OS files. Furthermore, for performance reasons,
it’s a good idea (again, in an ideal world) to keep any files with lots of concurrent
usage on different disks. Indexes can be stored separately from their tables, logs
from database files, and any other time you see contention for disk resources.

One more thing — if you decide to follow my advice on this issue, be sure to keep
in mind that you may have multiple databases on the same system. If you size each
of them to take up the full size of the disk less a few megabytes, you will still have
the possibility of a full disk (if they all expand). There really isn’t any one “right”
answer for this scenario — you just need to prioritize your space according to likely
usage, monitor your database sizes more closely, and set up alerts in Windows
Server to notify you of low disk space situations.
FILEGROWTH

Whereas SIZE sets the initial size of the database, and MAXSIZE determines just how large the database file could get, FILEGROWTH essentially determines just how fast it gets to that maximum. You provide a value that indicates by how many bytes (in KB, MB, GB, or TB) at a time you want the file to be enlarged. Alternatively, you can provide a percentage value by which you want the database file to increase. With this option, the size will go up by the stated percentage of the current database file size. Therefore, if you set a database file to start out at 1GB with a FILEGROWTH of 20 percent, the first time it expands it will grow to 1.2GB, the second time to 1.44, and so on.

LOG ON

The LOG ON option allows you to establish that you want your log to go to a specific set of files and where exactly those files are to be located. If this option is not provided, SQL Server creates the log in a single file and defaults it to a size equal to 25 percent of the data file size. In most other respects, it has the same file specification parameters as the main database file does.

NOTE

It is highly recommended that you store your log files on a different drive than your main data files. Doing so prevents the log and main data files from competing for I/O off the disk and provides additional safety should one hard drive fail.

COLLATE

This one has to do with the issue of sort order, case sensitivity, and sensitivity to accents. When you installed your SQL Server, you decided on a default collation, but you can override this at the database level (and, as you’ll see later, also at the column level).

FOR ATTACH

You can use this option to attach an existing set of database files to the current server. The files in question must be part of a database that was, at some point, properly detached using sp_detach_db. This deprecates the older sp_attach_db functionality and has the advantage of access to as many as 32,000 files — sp_attach_db is limited to 16.

If you use FOR ATTACH, you must complete the ON PRIMARY portion of the file location information. Other parts of the CREATE DATABASE parameter list can be left off as long as you are attaching the database to the same file path they were in when they were originally detached.

WITH DB CHAINING ON|OFF

Hmmm. How to address this one in a beginning-type way . . . Well, suffice it to say this is a toughie, and is in no way a “beginning” kind of concept. With that in mind, here’s the abridged version of what this relates to.

As previously mentioned, the concept of “schemas” didn’t really exist in early versions of SQL Server. Instead, there was the notion of “ownership.” One of the bad things that could happen with
ownership was an *ownership chain*. This was a situation in which person A was the owner of an object, and then person B became the owner of an object that depended on person A’s object. You could have person after person create objects depending on other people’s objects, and there was a complex weave of permission issues based on this. An ownership chain that gets too complicated can make it difficult to implement any kind of sensible security model, as the weave of rights required to grant permissions on an object becomes a Gordian knot.

This switch is about respecting such ownership chains when they cross databases (person A’s object is in DB1, and person B’s object is in DB2). Turn it on, and cross database ownership chains work — turn it off, and they don’t. Avoid such ownership chains as if they were the plague — if you find yourself thinking ownership chaining is a good idea, imagine unraveling that knot later, and reconsider.

**TRUSTWORTHY**

This switch adds an extra layer of security around access to system resources and files outside of the SQL Server context. For example, you may run a .NET assembly that touches files on your network — if so, you must identify the database that the assembly is part of as being trustworthy.

By default, this is turned off for security reasons — be certain you understand exactly what you’re doing and why before you set this to on.

**Building a Database**

At this point, you should be ready to begin building your database. The following is the statement to create it, but keep in mind that the database itself is only one of many objects that you’ll create on your way to a fully functional database:

```sql
CREATE DATABASE Accounting
ON
  (NAME = 'Accounting',
   FILENAME = 'C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER\MSSQL\DATA\AccountingData.mdf',
   SIZE = 10,
   MAXSIZE = 50,
   FILEGROWTH = 5)
LOG ON
  (NAME = 'AccountingLog',
   FILENAME = 'C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER\MSSQL\DATA\AccountingLog.ldf',
   SIZE = 5MB,
   MAXSIZE = 25MB,
   FILEGROWTH = 5MB);
GO
```

Now is a good time to start learning about some of the informational utilities that are available with SQL Server. You saw `sp_help` in Chapter 4, but in this case, you’ll try running a command called
sp_helpdb. This one is especially tailored for database structure information, and often provides better information if you’re more interested in the database itself than in the objects it contains. 

**sp_helpdb** takes one parameter — the database name:

```
EXEC sp_helpdb 'Accounting'
```

This actually yields you two separate result sets. The first, shown in the following table, is based on the combined (data and log) information about your database:

<table>
<thead>
<tr>
<th>Name</th>
<th>db_size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Owner</td>
<td>dbid</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>DB_SIZE</th>
<th>OWNER</th>
<th>DBID</th>
<th>CREATED</th>
<th>STATUS</th>
<th>COMPATIBILITY _LEVEL</th>
</tr>
</thead>
</table>

**NOTE** The actual values you receive for each of these fields may vary somewhat from mine. For example, the DBID value will vary depending on how many databases you’ve created and in what order you’ve created them. The various status messages will vary depending on what server options were in place at the time you created the database, as well as any options you changed for the database along the way.
Note that the `db_size` property is the *total* of the size of the database and the size of the log.

The second provides specifics about the various files that make up your database — including their current size and growth settings (shown in Table 5-1):

### Table 5-1: Files That Make Up the Database

<table>
<thead>
<tr>
<th>NAME</th>
<th>FILEID</th>
<th>FILENAME</th>
<th>FILEGROUP</th>
<th>SIZE</th>
<th>MAXSIZE</th>
<th>GROWTH</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accounting</td>
<td>1</td>
<td>C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER\MSSQL\DATA\Accounting Data.mdf</td>
<td>PRIMARY</td>
<td>10240 KB</td>
<td>51200 KB</td>
<td>5120 KB</td>
<td>data only</td>
</tr>
<tr>
<td>AccountingLog</td>
<td>2</td>
<td>C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER\MSSQL\DATA\Accounting Log.ldf</td>
<td>NULL</td>
<td>5120 KB</td>
<td>25600 KB</td>
<td>5120 KB</td>
<td>log only</td>
</tr>
</tbody>
</table>

After you create tables and insert data, the database will begin to grow automatically on an as-needed basis.

**CREATE TABLE**

The first part of creating a table is pretty much the same as creating any object — remember that line I showed you? Well, here it is again:

```
CREATE <object type> <object name>
```

Because a table is what you want, you can be more specific:

```
CREATE TABLE Customers
```

With `CREATE DATABASE`, you could have stopped with just these first three keywords, and it would have built the database based on the guidelines established in the model database. With tables,
however, there is no model, so you need to provide some more specifics in the form of columns, data types, and special operators.

Let’s look at more extended syntax:

```
CREATE TABLE [database_name.[owner].]table_name
(<column name> <data type>
[[DEFAULT <constant expression>]
 | [[IDENTITY [(seed, increment) [NOT FOR REPLICATION]]]]
 | ROWGUIDCOL]
 | COLLATE <collation name>]
 | NULL|NOT NULL]
 | <column constraints>]
 | [column_name AS computed_column_expression]
 | [<table_constraint>]
 |,...n]
)
|ON {<filegroup>|DEFAULT}]
| [TEXTIMAGE_ON {<filegroup>|DEFAULT}]
```

Now that’s a handful — and it still has sections taken out of it for simplicity’s sake! As usual, let’s look at the parts, starting with the second line (you’ve already seen the top line).

**Table and Column Names**

What’s in a name? Frankly — a lot. You may recall that one of my first soapbox diatribes was back in Chapter 2 and was about names. I promised then that it wouldn’t be the last you heard from me on the subject, and this won’t be either.

The rules for naming tables and columns are, in general, the same rules that apply to all database objects. The SQL Server documentation will refer to these as the *rules for identifiers*, and they are the same rules you observed at the end of Chapter 1. The rules are actually pretty simple; what I want to touch on here, though, are some notions about how exactly to name your objects — not specific rules governing what SQL Server will and won’t accept for names, but how to go about naming your tables and columns so that they are useful and make sense.

There are a ton of different “standards” out there for naming database objects — particularly tables and columns. My rules are pretty simple:

- **Capitalization:** For each word in the name, capitalize the first letter and use lowercase for the remaining letters.
- **Name length:** Keep the name short, but make it long enough to be descriptive.
- **Limit the use of abbreviations:** The only acceptable use of abbreviations is when the chosen abbreviation is recognized by everyone. Examples of abbreviations I use include “ID” to take the place of identification, “No” or “Num” (pick one) to take the place of number, and “Org” to take the place of organization. Keeping your names of reasonable length requires you to be more cavalier about your abbreviations sometimes, but keep in mind that, first and foremost, you want clarity in your names.
Linking tables: When building tables based on other tables (usually called linking or associate tables), you should include the names of all parent tables in your new table name. For example, say you have a movie database where many stars can appear in many movies. If you have a Movies table and a Stars table, you may want to tie them together using a table called MoviesStars.

Eliminate spacing between words: When you have two words in the name, do not use any separators (run the words together) — use the fact that you capitalize the first letter of each new word to determine how to separate words.

I can’t begin to tell you the battles I’ve had with other database people about naming issues. For example, you’ll find that a good many people believe that you should separate the words in your names with an underscore (_). Why don’t I do it that way? Well, it’s an ease of use issue. Underscores present a couple of different problems:

First, many people have a difficult time typing an underscore without taking their hand away from the proper keyboard position — this leads to lots of typos.

Second, in documentation it is not uncommon to run into situations where the table or column name is underlined. Underscores are, depending on the font, impossible to see when the text is underlined — this leads to confusion and more errors.

Finally (and this is a nit-pick), it’s just more typing.

NOTE You also have the option to separate the words in the name using a regular space. If you recall my very first soapbox diatribe back in Chapter 1, you’ll know that isn’t really much of an option — it is extremely bad practice and creates an unbelievable number of errors. It was added to facilitate Access upsizing, and I continue to curse the people who decided to put it in — I’m sure they were well-meaning, but they are now part of the cause of much grief in the database world.

This list is certainly not set in stone; rather it is just a Reader’s Digest version of the rules I use when naming tables. I find that they save me a great deal of grief. I hope they’ll do the same for you.

NOTE Consistency, consistency, consistency. Every time I teach, I always warn my class that it’s a word I’m going to repeat over and over, and in no place is it more important than in naming. If you have to pick one rule to follow, pick a rule that says that whatever your standards are, make them just that: standard. If you decide to abbreviate for some reason, abbreviate that word every time (the same way). Regardless of what you’re doing in your naming, make it apply to the entire database consistently — consider having a standards document or style guide to make sure other developers utilize the same rules you do. This will save a ton of mistakes, and it will save your users time in terms of how long it takes for them to get to know the database.
Data Types

There isn’t much to this — the data types are as I described them in Chapter 2. You just need to provide a data type immediately following the column name — there is no default data type.

DEFAULT

I’ll cover this in much more detail in the chapter on constraints, but for now, suffice to say that this is the value you want to use for any rows that are inserted without a user-supplied value for this particular column. The default, if you use one, should immediately follow the data type.

IDENTITY

The concept of an identity value is very important in database design. I will cover how to use identity columns in some detail in the chapters on design. What is an identity column? Well, when you make a column an identity column, SQL Server automatically assigns a sequenced number to this column with every row you insert. The number that SQL Server starts counting from is called the seed value, and the amount that the value increases or decreases by with each row is called the increment. The default is for a seed of 1 and an increment of 1, and most designs call for it to be left that way. As an example, however, you could have a seed of 3 and an increment of 5. In this case, you would start counting from 3, and then add 5 each time for 8, 13, 18, 23, and so on.

An identity column must be numeric, and, in practice, it is almost always implemented with an integer or bigint data type.

The usage is pretty simple; you simply include the identity keyword right after the data type for the column. An identity option cannot be used in conjunction with a default constraint. This makes sense if you think about it — how can there be a constant default if you’re counting up or down every time?

NOTE It’s worth noting that an identity column works sequentially. That is, once you’ve set a seed (the starting point) and the increment, your values only go up (or down if you set the increment to a negative number). There is no automatic mechanism to go back and fill in the numbers for any rows you may have deleted. If you want to fill in blank spaces like that, you need to use SET IDENTITY_INSERT ON, which allows you to turn off the identity process for inserts from the current connection. (Yes, turning it “on” turns it off — that is, you are turning on the ability to insert your own values, which has the effect of turning off the automatic value.) This can, however, create havoc if you’re not careful or if people are still trying to use the system as you do this, so tread carefully.

The most common use for an identity column is to generate a new value for use as an identifier for each row — that is, you commonly utilize identity columns to create a primary key for a table. Keep in mind, however, that an identity column and a primary key are completely separate notions — that is, just because you have an identity column doesn’t mean that the value is unique (for example,
you can reset the seed value and count back up through values you’ve used before). **IDENTITY** values are typically used as the **PRIMARY KEY** column, but they don’t have to be used that way.

**NOTE** If you’ve come from the Access world, you’ll notice that an **IDENTITY** column is much like an **AutoNumber** column. The major difference is that you have a bit more control over it in SQL Server.

**NOT FOR REPLICATION**

This one is very tough to deal with at this point, so I am, at least in part, going to skip most of the details. Replication is a notion that’s not covered in this book, so “not for replication” is going to lack context.

**NOTE** Briefly, replication is the process of automatically doing what, in a very loose sense, amounts to copying some or all of the information in your database to some other database. The other database may be on the same physical machine as the original, or it may be located remotely.

The **NOT FOR REPLICATION** parameter determines whether a new identity value for the new database is assigned when the column is published to another database (via replication), or whether it keeps its existing value. There is a lot more to this when you start learning about SQL Server 2012’s “Always On” capabilities, but that’s more of an administrative than a programming topic. If you want to learn more, I suggest *Professional Microsoft SQL Server 2012 Administration* by Adam Jorgensen.

**ROWGUIDCOL**

This is also replication related and, in many ways, is the same in purpose to an identity column. You’ve already seen how using an identity column can provide you with an easy way to make sure that you have a value that is unique to each row and can, therefore, be used to identify that row. However, this can be a very error-prone solution when you are dealing with replicated or other distributed environments.

Think about it for a minute — while an identity column will keep counting upward from a set value, what’s to keep the values from overlapping on different databases? Now, think about when you try to replicate the values such that all the rows that were previously in separate databases now reside in one database — uh oh! You now have duplicate values in the column that is supposed to uniquely identify each row!

Over the years, the common solution for this was to use separate seed values for each database you were replicating to and from. For example, you may have database A that starts counting at 1, database B starts at 10,000, and database C starts at 20,000. You can now publish them all into the
same database safely — for a while. As soon as database A has more than 9,999 records inserted into it, you’re in big trouble.

“Sure,” you say, “why not just separate the values by 100,000 or 500,000?” If you have tables with a large amount of activity, you’re still just delaying the inevitable — that’s where a \texttt{ROWGUIDCOL} comes into play.

What is a \texttt{ROWGUIDCOL}? Well, it’s quite a bit like an identity column in that it is usually used to uniquely identify each row in a table. The difference is to what lengths the system goes to make sure that the value used is truly unique. Instead of using a numerical count, SQL Server instead uses what is known as a \textit{unique identifier} (in fact, GUID stands for Globally Unique Identifier). While an identity value is usually (unless you alter something) unique across time, it is not unique across space. Therefore, you can have two copies of your table running, and can have them both assigned identical identity values for what are different rows. While this is just fine to start with, it causes big problems when you try to bring the rows from both tables together as one replicated table. A unique identifier, sometimes still referred to as a \textit{GUID}, is unique across both space and time.

\textbf{NOTE} GUIDs (or, more commonly today, UUIDs — which look the same and do a better job at performing the same task) are in widespread use in computing today. For example, if you check the registry, you’ll find tons of them. A GUID is a 128-bit value — for you math types, that’s 38 zeros in decimal form. If I generated a GUID every second, it would, theoretically speaking, take me millions of years to generate a duplicate given a number of that size.

GUIDs are generated using a combination of information — each of which is designed to be unique in either space or time. When you combine them, you come up with a value that is guaranteed, statistically speaking, to be unique across space and time.

There is a Windows API call to generate a GUID in normal programming, but, in addition to the \texttt{ROWGUIDCOL} option on a column, SQL has a special function to return a GUID — it is called the \texttt{NEWID()} function, and can be called at any time.

\textbf{COLLATE}

This works pretty much just as it did for the \texttt{CREATE DATABASE} command, with the primary difference being in terms of scope (here, you define at the column level rather than the database level).

\textbf{NULL/NOT NULL}

This one is pretty simple — it states whether or not the column in question accepts \texttt{NULL} values. The default, when you first install SQL Server, is to set a column to \texttt{NOT NULL} if you don’t specify nullability. There is, however, a very large number of settings that can affect this default and change its behavior. For example, setting a value by using the \texttt{sp_dbcmaplevel} stored procedure or setting ANSI-compliance options can change this value.
Column Constraints

There’s a whole chapter coming up on constraints, so I won’t spend that much time on them here. Still, it seems like a good time to review the question of what column constraints are — in short, they are restrictions and rules that you place on individual columns about the data that can be inserted into that column.

For example, if you have a column that’s supposed to store the month of the year, you might define that column as being of type `tinyint` — but that wouldn’t prevent someone from inserting the number 54 in that column. Because 54 would give you bad data (it doesn’t refer to a month), you might provide a constraint that says that data in that column must be between 1 and 12. You’ll see how to do this in the next chapter.

Computed Columns

You can also have a column that doesn’t have any data of its own, but whose value is derived on the fly from other columns in the table. If you think about it, this may seem odd because you could just figure it out at query time, but really, this is something of a boon for many applications.

For example, let’s say that you’re working on an invoicing system. You want to store information about the quantity of an item you have sold, and at what price. It used to be fairly commonplace to go ahead and add columns to store this information, along with another column that stored the extended value (price times quantity). However, that leads to unnecessary wasting of disk space and maintenance hassles when the totals and the base values get out of synch. With a computed column, you can get around that by defining the value of the computed column as price multiplied by quantity.

Let’s look at the specific syntax:

```
<column name> AS <computed column expression>
```

The first item is a little different; it provides a column name to go with the value. This is simply the alias that you’re going to use to refer to the value that is computed, based on the expression that follows the `AS` keyword.

Next comes the computed column expression. The expression can be any normal expression that uses either literals or column values from the same tables. Therefore, in the example of price and quantity, you might define this column as:

```
ExtendedPrice AS Price * Quantity
```
For an example using a literal, let’s say that you always charge a fixed markup on your goods that is 20 percent over your cost. You could simply keep track of cost in one column, and then use a computed column for the ListPrice column:

```
ListPrice AS Cost * 1.2
```

Pretty easy, eh? There are a few caveats and provisos though:

- You cannot use a subquery, and the values cannot come from a different table.
- You cannot directly specify the data type of a computed column; it is implicitly of whatever type the expression produces. That said, you can use `CAST` or `CONVERT` as part of your expression to explicitly impose a type on the result.
- In SQL Server 2000 and earlier, you could not use a computed column as any part of any key (primary, foreign, or unique) or with a default constraint. For SQL Server 2005 and later, you can now use a computed column in constraints (you must flag the computed column as persisted if you do this, however).
- Special steps must be taken if you want to create indexes on computed columns. You’ll learn about these steps when you explore indexing in Chapter 9.

There will be plenty of specific examples of how to use computed columns a little later in this chapter.

---

**NOTE** Even years after computed columns were added to the product, I’m still rather surprised that I don’t hear much debate about the use of them. Rules for normalization of data say that we should not have a column in our table for information that can be derived from other columns — that’s exactly what a computed column is!

I’m glad the religious zealots of normalization haven’t weighed in on this one much, as I like computed columns as something of a compromise. By default, you aren’t storing the data twice, and you don’t have issues with the derived values not agreeing with the base values because they are calculated on the fly directly from the base values. However, you still get the end result you wanted. Note that if you index the computed column, you are indeed actually storing the data (you have to for the index). This, however, has its own benefits when it comes to read performance.

This isn’t the way to do everything related to derived data, but it sure is an excellent helper for many situations.

---

**Table Constraints**

Table constraints are quite similar to column constraints, in that they place restrictions on the data that can be inserted into the table. What makes them a little different is that they may be based on more than one column.
Again, I will be covering these in the constraints chapter (Chapter 6), but examples of table-level constraints include `PRIMARY` and `FOREIGN KEY` constraints, as well as `CHECK` constraints.

**NOTE** Okay, so why is a `CHECK` constraint a table constraint? Isn’t it a column constraint because it affects what you can place in a given column? The answer is that it’s both. If it is based on solely one column, it meets the rules for a column constraint. If, however (as `CHECK` constraints can be), it is dependent on multiple columns, you have what would be referred to as a table constraint.

**ON**

Remember when you were dealing with database creation, and I said you could create different filegroups? Well, the `ON` clause in a table definition is a way of specifically stating on which filegroup (and, therefore, physical device) you want the table located. You can place a given table on a specific physical device or, as you will want to do in most cases, just leave the `ON` clause out, and it’s placed on whatever the default filegroup is (which is the `PRIMARY` unless you’ve set it to something else). You will be looking at this usage extensively in the Chapter 9, which deals with indexes and performance tuning.

**TEXTIMAGE_ON**

This one is basically the same as the `ON` clause you just looked at, except that it lets you move a very specific part of the table to yet a different filegroup. This clause is valid only if your table definition has large column(s) in it, including the following:

- `text` or `ntext`
- `image`
- `xml`
- `varchar(max)` or `nvarchar(max)`
- `varbinary(max)`
- Any CLR user-defined type columns (including `geometry` and `geography`)

When you use the `TEXTIMAGE_ON` clause, you move only the BLOB information into the separate filegroup — the rest of the table stays either on the default filegroup or with the filegroup chosen in the `ON` clause.

**TIP** There can be some serious performance increases to be had by splitting your database into multiple files, and then storing those files on separate physical disks. When you do this, it means you get the I/O from both drives. Major discussion of this is outside the scope of this book, but keep this in mind as something to gather more information about should you run into I/O performance issues.
 Creating a Table

All right, you’ve seen plenty; you’re ready for some action, so now you can build a few tables.

When you started this section, you looked at the standard create syntax of:

```
CREATE <object type> <object name>
```

And then you moved on to a more specific start (indeed, it’s the first line of the statement that creates the table) and created a table called Customers:

```
CREATE TABLE Customers
```

Your Customers table is going to be the first table in a database you will be putting together to track your company’s accounting. You’ll be looking at designing a database in a couple of chapters, but you can go ahead and get started on your database by building a couple of tables to learn the create table statement. You’ll look at most of the concepts of table construction in this section, but I’ll save a few for later on in the book. That being said, let’s get started building the first of several tables.

You’ll want to add in a USE <database name> line prior to the create code to ensure that when you run the script, the table is created in the proper database. You can then follow up that first line, which you’ve already seen, with a few columns.

```
USE Accounting
CREATE TABLE Customers
```
```
  (CustomerNo    int         IDENTITY  NOT NULL,
   CustomerName  varchar(30)                NOT NULL,
   Address1      varchar(30)                NOT NULL,
   Address2      varchar(30)                NOT NULL,
   City          varchar(20)                NOT NULL,
   State         char(2)                    NOT NULL,
   Zip           varchar(10)                NOT NULL,
```
The CREATE Statement

```sql
Contact       varchar(25)                NOT NULL,
Phone         char(15)                   NOT NULL,
FedIDNo       varchar(9)                 NOT NULL,
DateInSystem  date              NOT NULL
)
```

This is a somewhat simplified table compared to what you would probably use in real life, but there’s plenty of time to change it later (and you will).

Once you’ve built the table, you’ll want to verify that it was indeed created, and that it has all the columns and types that you expect. To do this, you can make use of several commands, but perhaps the best is one that will seem like an old friend before you’re done with this book: `sp_help`. The syntax is simple:

```sql
EXEC sp_help <object name>
```

To specify the table object that you just created, try executing the following code:

```sql
EXEC sp_help Customers
```

You use the `EXEC` command in two different ways. This rendition executes a stored procedure — in this case, a system stored procedure. You’ll see the second version later when you are dealing with advanced query topics and stored procedures in Chapters 11 and 12.

**NOTE** Technically speaking, you can execute a stored procedure by simply calling it (without using the `EXEC` keyword). The problem is that this works only if the `sproc` being called is the first statement of any kind in the batch. Just having `sp_help Customers` would have worked in the place of the previous code, but if you tried to run a `SELECT` statement before it — it would blow up on you. Not using `EXEC` leads to very unpredictable behavior and should be avoided.

Try executing the command, and you’ll find that you get back several result sets, one after another. The information retrieved includes separate result sets for:

- Table name, schema, type of table (system versus user), and creation date
- Column names, data types, nullability, size, and collation
- The identity column (if one exists), including the `initial` seed and increment values
- The `ROWGUIDCOL` (if one exists)
Filegroup information
Index names (if any exist), types, and included columns
Constraint names (if any), types, and included columns
Foreign key (if any) names and columns
The names of any schema-bound views (more on this in Chapter 10) that depend on the table

Now that you’re certain that your table was created, let’s take a look at creating yet another table — the Employees table. This time, let’s talk about what you want in the table first, and then you can try to code the CREATE script for yourself.

The Employees table is another fairly simple table. It should include information on:

- The employee’s ID — this should be automatically generated by the system
- First name
- Optionally, middle initial
- Last name
- Title
- Social Security number
- Salary
- The previous salary
- The amount of the last raise
- Date of hire
- Date terminated (if there is one)
- The employee’s manager
- Department

Start by trying to figure out a layout for yourself.

Before we start looking at this together, let me tell you not to worry too much if your layout isn’t exactly like mine. There are as many database designs as there are database designers — and that all begins with table design. We all can have different solutions to the same problem. What you want to look for is whether you have all the concepts that need to be addressed. That being said, let’s take a look at one way to build this table.

There is at least one special column here. The EmployeeID is to be generated by the system and therefore is an excellent candidate for either an identity column or a ROWGUIDCOL. There are several reasons you might want to go one way or the other between these two, but I’ll go with an identity column for a couple of reasons:

- It’s going to be used by an average person. (Would you want to have to remember a GUID?)
- It incurs lower overhead.
You’re now ready to start constructing your script. Here’s the start of mine:

```sql
CREATE TABLE Employees
(
    EmployeeID int IDENTITY NOT NULL,
    FirstName  varchar(25)            NOT NULL,
    MiddleInitial char(1)                NULL,
    LastName   varchar(25)            NOT NULL,
)
```

For this column, the `NOT NULL` option has essentially been chosen by virtue of using an `IDENTITY` column. You cannot allow `NULL` values in an `IDENTITY` column. Note that, depending on your server settings, you will, most likely, still need to include `NOT NULL` explicitly (if you leave it to the default you may get an error depending on whether the default allows `NULLs`).

Next up, you want to add in your name columns. I usually allow approximately 25 characters for names. Most names are far shorter than that, but I’ve bumped into enough that were rather lengthy (especially because hyphenated names have become so popular) that I allow for the extra room. In addition, I make use of a variable-length data type for two reasons:

- To recapture the space of a column that is defined somewhat longer than the actual data usually is (retrieve blank space)
- To simplify searches in the `WHERE` clause — fixed-length columns are padded with spaces, which requires extra planning when performing comparisons against fields of this type

The exception in this case is the middle initial. Because you really need to allow for only one character here, recapture of space is not an issue. Indeed, a variable-length data type would actually use more space in this case, because a `varchar` needs not only the space to store the data, but also a small amount of overhead space to keep track of how long the data is. In addition, ease of search is not an issue because if you have any value in the field at all, there isn’t enough room left for padded spaces.

**NOTE** For the code that you write directly in T-SQL, SQL Server automatically adjusts to the padded spaces issue — that is, an `xx` placed in a `char(5)` is treated as being equal (if compared) to an `xx` placed in a `varchar(5)` — this is not, however, true in your client APIs such as SqlNativeClient and ADO.NET. If you connect to a `char(5)` in ADO.NET, an `xx` will evaluate to `xx` with three spaces after it — if you compare it to `xx`, it will evaluate to `False`. An `xx` placed in a `varchar(5)`, however, will automatically have any trailing spaces trimmed, and comparing it to `xx` in ADO.NET will evaluate to `True`.

Because a name for an employee is a critical item, I will not allow any `NULL` values in the first and last name columns. The middle initial is not nearly so critical (indeed, some people in the United States don’t have a middle name at all, while my wife has two, and I hear that it’s not uncommon for Brits to have several), so let’s allow a `NULL` for that field only:

```sql
FirstName  varchar(25)            NOT NULL,
MiddleInitial char(1)                NULL,
LastName   varchar(25)            NOT NULL,
```
Next up is the employee’s title. You must know what they are doing if you’re going to be cutting them a paycheck, so you can also make this a required field:

```
Title            varchar(25)            NOT NULL,
```

In that same paycheck vein, you must know their Social Security number (or similar identification number outside the United States) in order to report for taxes. In this case, you can use a `varchar` and allow up to 11 characters, as these identification numbers are different lengths in different countries. If you know your application is going to require only SSNs from the United States, you’ll probably want to make it `char(11)` instead:

```
SSN              varchar(11)            NOT NULL,
```

You must know how much to pay the employees — that seems simple enough — but what comes next is a little different. When you add in the prior salary and the amount of the last raise, you get into a situation where you could use a computed column. The new salary is the sum of the previous salary and the amount of the last raise. The `Salary` amount is something that you might use quite regularly — indeed you might want an index on it to help with ranged queries, but for various reasons I don’t want to do that here (I’ll talk about the ramifications of indexes on computed columns in Chapter 9), so I’m going to use `LastRaise` as my computed column:

```
Salary           money                  NOT NULL,
PriorSalary      money                  NOT NULL,
LastRaise AS Salary - PriorSalary,
```

If you hired them, you must know the date of hire — so that will also be required:

```
HireDate         date                   NOT NULL,
```

Note that I’ve chosen to use a `date` data type rather than the older standard `datetime` to save space. The `datetime` data type stores both date and time information down to fractions of a second. However, because you’re primarily interested in the date of hire, not the time, the `date` will meet your needs and take up half the space.

---

**NOTE** `NOT NULL` does not mean not empty. Were you to encounter a fellow with no last name — let’s take the musician, Sting, for example — you could add an entry with an empty string (`' '`) for a last name if you wished. This is a normal situation, where you design for the rule and handle the exception. You could build in enough flexibility to perfectly handle every possible situation, but is this unwieldy design worth it, given the rarity of the case? As always with SQL, it depends on your exact situation, but in general I’ll repeat — design for the rule and handle the exception.
The date of termination is something you may not know (I’d like to think that some employees are still working for you), so you’ll need to leave it nullable:

```sql
TerminationDate date NULL,
```

You absolutely want to know who the employees are reporting to (somebody must have hired them!) and what department they are working in:

```sql
ManagerEmpID int NOT NULL,
Department varchar(25) NOT NULL
```

So, just for clarity, let’s look at the entire script required to create this table:

```sql
USE Accounting

CREATE TABLE Employees
(
    EmployeeID int IDENTITY NOT NULL,
    FirstName varchar(25) NOT NULL,
    MiddleInitial char(1) NULL,
    LastName varchar(25) NOT NULL,
    Title varchar(25) NOT NULL,
    SSN varchar(11) NOT NULL,
    Salary money NOT NULL,
    PriorSalary money NOT NULL,
    LastRaise AS Salary - PriorSalary,
    HireDate date NOT NULL,
    TerminationDate date NOT NULL,
    ManagerEmpID int NOT NULL,
    Department varchar(25) NOT NULL
)
```

Again, I would recommend executing `sp_help` on this table to verify that the table was created as you expected.

**THE ALTER STATEMENT**

Okay, so now you have a database and a couple of nice tables — isn’t life grand? If only things always stayed the same, but they don’t. Sometimes (actually, far more often than you would like), you’ll get requests to *change* a table rather than re-create it. Likewise, you may need to change the size, file locations, or some other feature of your database. That’s where the `ALTER` statement comes in.
Much like the `CREATE` statement, the `ALTER` statement almost always starts out the same:

```
ALTER <object type> <object name>
```

This is totally boring so far, but it won’t stay that way. You’ll see the beginnings of issues with this statement right away, and things will get really interesting (read: convoluted and confusing!) when you deal with this even further in the next chapter (when you deal with constraints).

**ALTER DATABASE**

Let’s get right into it by taking a look at changing your database. You’ll actually make a couple of changes just so you can see the effects of different things and how their syntax can vary.

Perhaps the biggest trick with the `ALTER` statement is to remember what you already have. With that in mind, let’s take a look again at what you already have:

```
EXEC sp_helpdb Accounting
```

Notice that I didn’t put the quotation marks in this time as I did when you used this stored proc earlier. That’s because this system procedure, like many of them, accepts a special data type called `sysname`. As long as what you pass in is a name of a valid object in the system, the quotes are optional for this data type.

So, the results should be just like they were when you created the database (see the next two tables):

<table>
<thead>
<tr>
<th>NAME</th>
<th>DB_SIZE</th>
<th>OWNER</th>
<th>DBID</th>
<th>CREATED</th>
<th>STATUS</th>
<th>COMPATIBILITY_LEVEL</th>
</tr>
</thead>
</table>
And...

Let's say you want to change things a bit. For example, let's say that you know that you are going to be doing a large import into your database. Currently, your database is only 15MB in size — that doesn’t hold much these days. Because you have Autogrow turned on, you could just start your import, and SQL Server would automatically enlarge the database 5MB at a time. Keep in mind, however, that it’s actually a fair amount of work to reallocate the size of the database. If you were inserting 100MB worth of data, the server would have to deal with that reallocation at least 16 times (at 20MB, 25MB, 30MB, and so on). Because you know that you’re going to be getting up to 100MB of data, why not just do it in one shot? To do this, you would use the `ALTER DATABASE` command.

The general syntax looks like this:

```
ALTER DATABASE <database name>
  ADD FILE
    ([NAME = <'logical file name'>,]
    FILENAME = <'file name'>
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>FILEID</th>
<th>FILENAME</th>
<th>FILEGROUP</th>
<th>SIZE</th>
<th>MAXSIZE</th>
<th>GROWTH</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accounting</td>
<td>1</td>
<td>c:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER\MSSQL\DATA\Accounting Data.mdf</td>
<td>PRIMARY</td>
<td>10240BK</td>
<td>5120KB</td>
<td>5120KB</td>
<td>data only</td>
</tr>
<tr>
<td>AccountingLog</td>
<td>2</td>
<td>c:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER\MSSQL\DATA\Accounting-Log.ldf</td>
<td>NULL</td>
<td>5120KB</td>
<td>25600KB</td>
<td>5120KB</td>
<td>log only</td>
</tr>
</tbody>
</table>

Copyright ©2012 John Wiley & Sons, Inc.
The reality is that you will very rarely use all that stuff — sometimes I think Microsoft just puts it there for the sole purpose of confusing the heck out of us (just kidding!).

So, after looking at all that gobbledygook, let’s just worry about what you need to expand your database to 100MB:

```sql
ALTER DATABASE Accounting
MODIFY FILE
(NAME = Accounting,
 SIZE = 100MB)
```

Note that, unlike when you created your database, you don’t get any information about the allocation of space — instead, you get the rather non-verbose:

```
The command(s) completed successfully.
```

Gee — how informative. . . . So, you’d better check on things for yourself (see the text two tables):

```sql
EXEC sp_helpdb Accounting
```
### The ALTER Statement

<table>
<thead>
<tr>
<th>NAME</th>
<th>DB_SIZE</th>
<th>OWNER</th>
<th>DBID</th>
<th>CREATED</th>
<th>STATUS</th>
</tr>
</thead>
</table>

And

<table>
<thead>
<tr>
<th>NAME</th>
<th>FILEID</th>
<th>FILENAME</th>
<th>FILE-GROUP</th>
<th>SIZE</th>
<th>MAXSIZE</th>
<th>GROWTH</th>
<th>USAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accounting</td>
<td>1</td>
<td>c:\Program Files\Microsoft SQL Server\MSSQL11. MSSQLSERVER\MSSQL\DATA\Accounting-Data.mdf</td>
<td>PRIMARY</td>
<td>102400 KB</td>
<td>102400 KB</td>
<td>5120 KB</td>
<td>data only</td>
</tr>
<tr>
<td>AccountingLog</td>
<td>2</td>
<td>c:\Program Files\Microsoft SQL Server\MSSQL11. MSSQLSERVER\MSSQL\DATA\Accounting-Log.ldf</td>
<td>NULL</td>
<td>5120 KB</td>
<td>25600 KB</td>
<td>5120 KB</td>
<td>log only</td>
</tr>
</tbody>
</table>
Things pretty much work the same for any of the more common database-level modifications you'll make. The permutations are, however, endless. The more complex filegroup modifications and the like are outside the scope of this book, but if you need more information on them, I recommend one of the more administrator-oriented books out there (and there are a ton of them).

**Option and Termination Specs**

SQL Server has a few options that can be set with an `ALTER DATABASE` statement. Among these are database-specific defaults for most of the `SET` options that are available (such as `ANSI_PADDING`, `ARITHABORT` — handy if you’re dealing with indexed or partitioned views), state options (for example, single user mode or read-only), and recovery options. The effects of the various `SET` options are discussed where they are relevant throughout the book. This new `ALTER` functionality simply gives you an additional way to change the defaults for any particular database.

SQL Server also has the ability to control the implementation of some of the changes you are trying to make on your database. Many changes require that you have exclusive control over the database — something that can be hard to deal with if other users are already in the system. SQL Server gives you the ability to gracefully force other users out of the database so that you can complete your database changes. The strength of these actions ranges from waiting a number of seconds (you decide how long) before kicking other users out, all the way up to immediate termination of any option transactions (automatically rolling them back). Relatively uncontrolled (from the client’s perspective) termination of transactions is not something to be taken lightly. Such an action is usually in the realm of the database administrator. As such, I will consider further discussion out of the scope of this book.

**ALTER TABLE**

A far, far more common need is the situation where you need to change the makeup of one of your tables. This can range from simple things like adding a new column to more complex issues such as changing a data type.

Let’s start by taking a look at the basic syntax for changing a table:

```
ALTER TABLE table_name
{ [ALTER COLUMN <column_name>
   { [ <schema of new data type>].<new_data_type>
   [(precision [, scale])|max |
   <xml schema_collection>
   [COLLATE <collation_name>]
   [NULL|NOT NULL]
   |[ADD|DROP] ROWGUIDCOL | PERSISTED]

   |ADD
   <column_name> <data_type>
   |[DEFAULT <constant_expression>]
   |[IDENTITY [((<seed>, <increment>) [NOT FOR REPLICATION]]
   [ROWGUIDCOL]
   [COLLATE <collation_name>]
   [NULL|NOT NULL]
   [<column_constraints>]
   |[<column_name> AS <computed_column_expression>]
|ADD
```
The ALTER Statement

As with the CREATE TABLE command, there’s quite a handful there to deal with.

So let’s start an example of using this by looking back at the Employees table in the Accounting database:

```
EXEC sp_help Employees
```

For the sake of saving a few trees, I’m going to edit the results that I show here in the next table to just the part you care about — you’ll actually see much more than this:

<table>
<thead>
<tr>
<th>COLUMN_NAME</th>
<th>TYPE</th>
<th>COMPUTED</th>
<th>LENGTH</th>
<th>PREC</th>
<th>SCALE</th>
<th>NULLABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>EmployeeID</td>
<td>int</td>
<td>no</td>
<td>4</td>
<td>10</td>
<td>0</td>
<td>no</td>
</tr>
<tr>
<td>FirstName</td>
<td>varchar</td>
<td>no</td>
<td>25</td>
<td></td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>MiddleInitial</td>
<td>char</td>
<td>no</td>
<td>1</td>
<td></td>
<td></td>
<td>yes</td>
</tr>
</tbody>
</table>
Let's say that you've decided you'd like to keep previous employer information on your employees (probably so you know who will be trying to recruit the good ones back!). That just involves adding another column, and really isn't all that tough. The syntax looks much like it did with the `CREATE TABLE` statement except that it has obvious alterations to it:

```
ALTER TABLE Employees
ADD
  PreviousEmployer   varchar(30)   NULL
```

Not exactly rocket science — is it? Indeed, you could have added several additional columns at one time if you had wanted to. It would look something like this:

```
ALTER TABLE Employees
ADD
  DateOfBirth     date       NULL,  
  LastRaiseDate   date       NOT NULL 
    DEFAULT '2008-01-01'
```

(continued)

<table>
<thead>
<tr>
<th>COLUMN_NAME</th>
<th>TYPE</th>
<th>COMPUTED</th>
<th>LENGTH</th>
<th>PREC</th>
<th>SCALE</th>
<th>NULLABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastName</td>
<td>varchar</td>
<td>no</td>
<td>25</td>
<td></td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>Title</td>
<td>varchar</td>
<td>no</td>
<td>25</td>
<td></td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>SSN</td>
<td>varchar</td>
<td>no</td>
<td>11</td>
<td></td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>Salary</td>
<td>money</td>
<td>no</td>
<td>8</td>
<td>19</td>
<td>4</td>
<td>no</td>
</tr>
<tr>
<td>PriorSalary</td>
<td>money</td>
<td>no</td>
<td>8</td>
<td>19</td>
<td>4</td>
<td>no</td>
</tr>
<tr>
<td>LastRaise</td>
<td>money</td>
<td>yes</td>
<td>8</td>
<td>19</td>
<td>4</td>
<td>yes</td>
</tr>
<tr>
<td>HireDate</td>
<td>date</td>
<td>no</td>
<td>3</td>
<td></td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>TerminationDate</td>
<td>date</td>
<td>no</td>
<td>3</td>
<td></td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>ManagerEmpID</td>
<td>int</td>
<td>no</td>
<td>4</td>
<td>10</td>
<td>0</td>
<td>no</td>
</tr>
<tr>
<td>Department</td>
<td>varchar</td>
<td>no</td>
<td>25</td>
<td></td>
<td></td>
<td>no</td>
</tr>
</tbody>
</table>

Let's say that you’ve decided you’d like to keep previous employer information on your employees (probably so you know who will be trying to recruit the good ones back!). That just involves adding another column, and really isn’t all that tough. The syntax looks much like it did with the `CREATE TABLE` statement except that it has obvious alterations to it:

```
ALTER TABLE Employees
ADD
  PreviousEmployer   varchar(30)   NULL
```

Not exactly rocket science — is it? Indeed, you could have added several additional columns at one time if you had wanted to. It would look something like this:

```
ALTER TABLE Employees
ADD
  DateOfBirth     date       NULL,  
  LastRaiseDate   date       NOT NULL 
    DEFAULT '2008-01-01'
```

**NOTE** Notice the `DEFAULT` I slid in here. You haven’t really looked at these yet (they are in the next chapter), but I wanted to use one here to point out a special case. If you want to add a `NOT NULL` column after the fact, you have the issue of what to do with rows that already have `NULL` values. I have shown one solution to that here by providing a default value. The default is then used to populate the new column for any row that is already in your table.
Before you go away from this topic for now, take a look at what you’ve added:

```sql
EXEC sp_help Employees
```

<table>
<thead>
<tr>
<th>COLUMN_NAME</th>
<th>TYPE</th>
<th>COMPUTED</th>
<th>LENGTH</th>
<th>PREC</th>
<th>SCALE</th>
<th>NULLABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>EmployeeID</td>
<td>int</td>
<td>no</td>
<td>4</td>
<td>10</td>
<td>0</td>
<td>no</td>
</tr>
<tr>
<td>FirstName</td>
<td>varchar</td>
<td>no</td>
<td>25</td>
<td></td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>MiddleInitial</td>
<td>char</td>
<td>no</td>
<td>1</td>
<td></td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>LastName</td>
<td>varchar</td>
<td>no</td>
<td>25</td>
<td></td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>Title</td>
<td>varchar</td>
<td>no</td>
<td>25</td>
<td></td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>SSN</td>
<td>varchar</td>
<td>no</td>
<td>11</td>
<td></td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>Salary</td>
<td>money</td>
<td>no</td>
<td>8</td>
<td>19</td>
<td>4</td>
<td>no</td>
</tr>
<tr>
<td>PriorSalary</td>
<td>money</td>
<td>no</td>
<td>8</td>
<td>19</td>
<td>4</td>
<td>no</td>
</tr>
<tr>
<td>LastRaise</td>
<td>money</td>
<td>yes</td>
<td>8</td>
<td>19</td>
<td>4</td>
<td>yes</td>
</tr>
<tr>
<td>HireDate</td>
<td>date</td>
<td>no</td>
<td>3</td>
<td></td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>TerminationDate</td>
<td>date</td>
<td>no</td>
<td>3</td>
<td></td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>ManagerEmpID</td>
<td>int</td>
<td>no</td>
<td>4</td>
<td>10</td>
<td>0</td>
<td>no</td>
</tr>
<tr>
<td>Department</td>
<td>varchar</td>
<td>no</td>
<td>25</td>
<td></td>
<td></td>
<td>no</td>
</tr>
<tr>
<td>PreviousEmployer</td>
<td>varchar</td>
<td>no</td>
<td>30</td>
<td></td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>DateOfBirth</td>
<td>date</td>
<td>no</td>
<td>3</td>
<td></td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>LastRaiseDate</td>
<td>date</td>
<td>no</td>
<td>3</td>
<td></td>
<td></td>
<td>no</td>
</tr>
</tbody>
</table>

As you can see, all of your columns have been added. The thing to note, however, is that they all went to the end of the column list. There is no way to add a column to a specific location in SQL Server. If you want to move a column to the middle, you need to create a completely new table (with a different name), copy the data over to the new table, `DROP` the existing table, and then rename the new one.
CHAPTER 5
CREATING AND ALTERING TABLES

THE DROP STATEMENT

Performing a `<DROP>` is the same as deleting whatever object(s) you reference in your `<DROP>` statement. It’s very quick and easy, and the syntax is exactly the same for all of the major SQL Server objects (tables, views, sprocs, triggers, and so on). It goes like this:

```
DROP <object type> <object name> [, ...n]
```

Actually, this is about as simple as SQL statements get. You could drop both of your tables at the same time if you wanted:

```
USE Accounting
DROP TABLE Customers, Employees
```

This deletes them both.

**NOTE** This issue of moving columns around can get very sticky indeed. Even some of the tools that are supposed to automate this often have problems with it. Why? Well, any foreign key constraints you have that reference this table must first be dropped before you are allowed to delete the current version of the table. That means that you have to drop all your foreign keys, make the changes, and then add all your foreign keys back. It doesn’t end there, however; any indexes you have defined on the old table are automatically dropped when you drop the existing table — that means that you must remember to re-create your indexes as part of the build script to create your new version of the table — yuck!

But wait! There’s more! While you haven’t really looked at views yet, I feel compelled to make a reference here to what happens to your views when you add a column. You should be aware that even if your view is built using a `SELECT *` as its base statement, your new column will not appear in your view until you rebuild the view. Column names in views are resolved at the time the view is created for performance reasons. That means any views that have already been created when you add your columns have already resolved using the previous column list — you must either `<DROP>` and re-create the view or use an `<ALTER VIEW>` statement to rebuild it.

**TIP** Be very careful with this command. There is no “Are you sure?” kind of question that goes with this — it just assumes you know what you’re doing and deletes the object(s) in question.
The syntax is very much the same for dropping the entire database. Now go ahead and drop the Accounting database:

```
USE master

DROP DATABASE Accounting
```

You should see the following in the Results pane:

Command(s) completed successfully.

You may run into a situation where you get an error that says that the database cannot be deleted because it is in use. If this happens, check a couple of things:

- Make sure that the database that you have as current in the Management Studio is something other than the database you’re trying to drop (that is, make sure you’re not using the database as you’re trying to drop it).
- Ensure you don’t have any other connections open (using the Management Studio or `sp_who`) that are showing the database you’re trying to drop as the current database.

I usually solve the first one just as I did in the code example — I switch to using the master database. The second issue you have to check manually — I usually close other sessions down entirely just to be sure. Of course, if one of those other sessions belongs to someone else, you may want to discuss things before dropping the database.

**USING THE GUI TOOL**

You’ve just spent a lot of time learning (or perhaps slogging through) perfect syntax for creating a database and a couple of tables — that’s enough of that for a while. Let’s take a look at the graphical tool in the Management Studio that allows you to build and relate tables. From this point on, you’ll not only be dealing with code, but with the tool that can generate much of that code for you.

**Creating a Database Using the Management Studio**

If you run the SQL Server Management Studio and expand the Databases node, you should see something like Figure 5-1.
**NOTE** If you look closely at Figure 5-1, you’ll see that my Accounting database is still showing even though I dropped it in the previous example (just as you did yours). You might not wind up seeing this, depending on whether you already had the Management Studio open when you dropped the database or you opened it after you dropped the database in the Query window.

Why the difference? Well, in earlier versions of SQL Server, the tools that are now the Management Studio refreshed information, such as the available databases, regularly. Now it updates only when it knows it has a reason to (for example, you deleted something by using the Management Studio Object Explorer instead of a Query window, or perhaps you explicitly chose to refresh). The reason for the change was performance. The old 6.5 Enterprise Manager used to be a slug performance-wise because it was constantly making round trips to “poll” the server. The newer approach performs much better, but doesn’t necessarily have the most up-to-date information.

The bottom line on this is that if you see something in the Management Studio Object Explorer that you don’t expect to, try clicking in there and pressing F5 (refresh). It should then update things for you.

Now follow these steps to create a database using Management Studio:

1. Right-click the Databases node, and choose the New Database option. This pulls up the Database Properties dialog box.

2. You can fill in the information on how you want your database created. Use the same choices that you did when you created the Accounting database at the beginning of the chapter. You need to fill in:

   ➤ **The basic name and size info:** This is shown in Figure 5-2. As far as the name goes — this is pretty basic. You called it Accounting before, and because you dropped the first one you created, there’s no reason not to call it that again.
➤ The file name, size, and growth information.

**NOTE** I’ve expanded the dialog box manually to make sure you can see everything. You may see less than what’s pictured here, as the default size of the dialog box is not nearly enough to show it all — just grab a corner of the dialog box and expand it to see the additional information.

3. Click Options tab, which contains a host of additional settings, as shown in Figure 5-3.

![Figure 5-3](image)

➤ **Collation name:** You have the choice of having each database (and, indeed, individual columns if you wish) have its own collation. For the vast majority of installs, you’ll want to stick with whatever the server default was set to when the server was installed (presumably, someone had already thought this out fairly well, although if you weren’t involved in the installation, it never hurts to ask). However, you can change it for just the current database by setting it here.

**NOTE** “Why,” you may ask, “would I want a different collation?” Well, in the English-speaking world, a common need for specific collations would be that some applications are written expecting an “a” to be the same as an “A” — while others are expecting case sensitivity (“a” is not the same as “A”). In the old days, you would have to have separate servers set up in order to handle this. Another, non-English example would be dialect differences that are found within many countries of the world — even where they speak the same general language.
COMPATIBILITY LEVEL: This controls whether certain SQL Server 2012 syntax and keywords are supported. As you might expect from the name of this setting, the goal is to allow you to roll back to keywords and functional behavior that more closely matches older versions if your particular application happens to need that. For example, as you roll it back to earlier versions, some words that are keywords in later versions revert to being treated as non-keywords, and certain behaviors that have had their defaults changed in recent releases will revert to the older default.

CONTAINMENT: As I said before, containment allows you to deploy a database with fewer dependencies on the target SQL instance. For this example, leave Containment set to None, the default.

4. The remaining properties will vary from install to install, but work as I described them earlier in the chapter. Okay, given that the other settings are pretty much standard fare, go ahead and try it out. Click OK, and after a brief pause during which the database is created, you’ll see it added to the tree.

5. Expand the tree to show the various items underneath the Accounting node, and select the Database Diagrams node. Right-click it, and you’ll get a dialog indicating that the database is missing some objects it needs to support database diagramming, as shown in Figure 5-4. Click Yes.

NOTE You should only see Figure 5-4 the first time a diagram is being created for that database. SQL Server keeps track of diagrams inside special tables that it creates in your database only if you are going to actually create a diagram that will use them.

With that, you’ll get an Add Table dialog, as shown in Figure 5-5. This lets you decide which tables you want to include in your diagram — you can create multiple diagrams if you wish, potentially each covering some subsection — or submodel — of the overall database schema. In this case, you have an empty list because you just created your database and it has no tables in it yet.

6. For now, just click Close (you can’t add tables if there are no tables to add!), and you should get an empty diagram screen.

7. You can add a table either by right-clicking and choosing the appropriate option, or by clicking the New Table icon in the toolbar. When you choose New Table, SQL Server asks you for the name you want to give your new table. You then get a mildly helpful dialog box.
that lets you fill in your table one piece at a time — complete with labels for what you need to fill out, as shown in Figure 5-6.

8. I've gone ahead and filled in the columns as they were in the original Customers table, but you also need to define your first column as being an identity column. Unfortunately, you don't appear to have any way of doing that with the default grid here. To change which items you can define for your table, you need to right-click in the editing dialog box, and select Table View ➪ Modify Custom.

You then get a list of items from which you can choose, as shown in Figure 5-7. For now, just select the extra item you need — Identity and its associated elements, Seed and Increment.

9. Go back to the editing dialog box and select Table View ➪ Custom to view the identity column (see Figure 5-8). You're ready to fill in your table definition.

**NOTE** Okay, so SQL Server can be a bit temperamental on this. If you do not check the box to make this the default, SQL Server changes what your “custom” view looks like, but it does not make the custom view the active one — the result is that you won’t see the changes you made as you exit the dialog box. So, again, make sure that after changing the view, you right-click and select Table View ➪ Custom again. It should then look like Figure 5-8.
10. Once you have the table filled out, you can save the changes, and that creates your table for you.

**NOTE** This is really a point of personal preference, but I prefer to set the view down to just column names at this point. You can do this by clicking the Show icon on the toolbar or, as I prefer, by right-clicking the table and choosing Table View ➔ Column Names. I find that this saves a lot of screen real estate and makes more room for me to work on additional tables.

Now try to add in the Employees table as it was defined earlier in the chapter. The steps should be pretty much as they were for the Customers table, with just one little hitch — you have a computed column. To deal with the computed column, just select Modify Custom again (from the right-click menu), and add the Formula column. Then, simply add the proper formula (in this case, Salary-PriorSalary). When you have all the columns entered, save your new table (accepting the confirmation dialog box). Your diagram should have two tables in it (see Figure 5-9).

**NOTE** It’s very important to understand that the diagramming tool that is included with SQL Server is not designed to be everything to everyone. Presumably, because you are reading this part of this book, you are just starting out on your database journey — this tool will probably be adequate for you for a while. Eventually, you may want to take a look at some more advanced (and far more expensive) tools to help you with your database design.

**Backing into the Code: Creating Scripts with the Management Studio**

One last quick introduction before you exit this chapter — you’ll read about the basics of having the Management Studio write your scripts for you. For now, I’m going to do this as something of a quick-and-dirty introduction.

To generate scripts, follow these steps:

1. Go into the Management Studio and right-click the database for which you want to generate scripts. (In this case, you’re going to generate scripts on the Accounting database.)

2. From the pop-up menu, choose Script Database As ➔ CREATE To ➔ New Query Editor Window, as shown in Figure 5-10.
Whoa! SQL Server generates a heck of a lot more code than you saw when you created your database to begin with. Don’t panic, however — all it is doing is being very explicit in scripting major database settings rather than relying on defaults as you did when you scripted it yourself.

Note that you are not limited to scripting the database — if you want to script other objects in the database, just navigate and right-click them much the way that you right-clicked the Accounting database and, boom!, you’ve got yourself a SQL Script.

As you can see, scripting couldn’t be much easier. Once you get a complex database put together, it won’t be quite as easy as it seems in this particular demonstration, but it is a lot easier than writing it all out by hand. The reality is that it really is pretty simple once you learn what the scripting options are, and you’ll learn much more about those later in the book.

**SUMMARY**

This chapter covered the basics of the *CREATE*, *ALTER*, and *DROP* statements as they relate to creating a database and tables. There are, of course, many other renditions of these that I will cover as you continue through the book. You have also taken a look at the wide variety of options that you can use in databases and tables to have full control over your data. Finally, you have begun to see the many things that you can use the Management Studio for in order to simplify your life, and make design and scripting simpler.

At this point, you’re ready to start getting into some hardcore details about how to lay out your tables, and a discussion on the concepts of normalization and more general database design. I am, however, actually going to make you wait another chapter before you get there, so that you know about constraints and keys somewhat before hitting the design issues.
EXERCISES

1. Using the Management Studio's script generator, generate SQL for both the Customers and the Employees tables.

2. Without using the Management Studio, script a database called MyDB with a starting database size of 17MB and a starting log size of 5MB — set both the log and the database to grow in 5MB increments.

3. Create a table called Foo with a single variable length character field called Col1 — limit the size of Col1 to 50 characters.
# WHAT YOU LEARNED IN THIS CHAPTER

<table>
<thead>
<tr>
<th>TOPIC</th>
<th>CONCEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CREATE DATABASE</strong></td>
<td>The container for all your data, the database object is created using the <code>CREATE DATABASE</code> statement. You can specify the size and location of the data and log files, spread the data across multiple files, and set up a data growth strategy.</td>
</tr>
<tr>
<td><strong>CREATE TABLE</strong></td>
<td>Tables are the basic unit of data storage. <code>CREATE TABLE</code> starts similarly to <code>CREATE DATABASE</code>, but the rest of the syntax centers on specifying column names and their data types, keys, and constraints, as well as some storage options.</td>
</tr>
<tr>
<td><strong>ALTER DATABASE</strong></td>
<td>Functions similarly to <code>CREATE DATABASE</code>, but applies its changes to a database that already exist. <code>ALTER DATABASE</code> will throw an error if it’s run against a nonexistent database.</td>
</tr>
<tr>
<td><strong>ALTER TABLE</strong></td>
<td>Can be used to add or change columns, constraints, or storage on an existing table.</td>
</tr>
<tr>
<td><strong>DROP DATABASE, DROP TABLE</strong></td>
<td>Used to remove an object entirely from the system. Dropping a database or table results in loss of any data stored within.</td>
</tr>
<tr>
<td><strong>Using SSMS to create objects</strong></td>
<td>Using Management Studio, a simple dialog can be used to create a database. Creating tables can also be done through the designer GUI.</td>
</tr>
<tr>
<td><strong>Using SSMS to generate scripts</strong></td>
<td>Using SSMS, you can have SQL Server generate and save a <code>CREATE</code> script for any object in your database. This is extremely useful for source control or deployment scenarios.</td>
</tr>
</tbody>
</table>