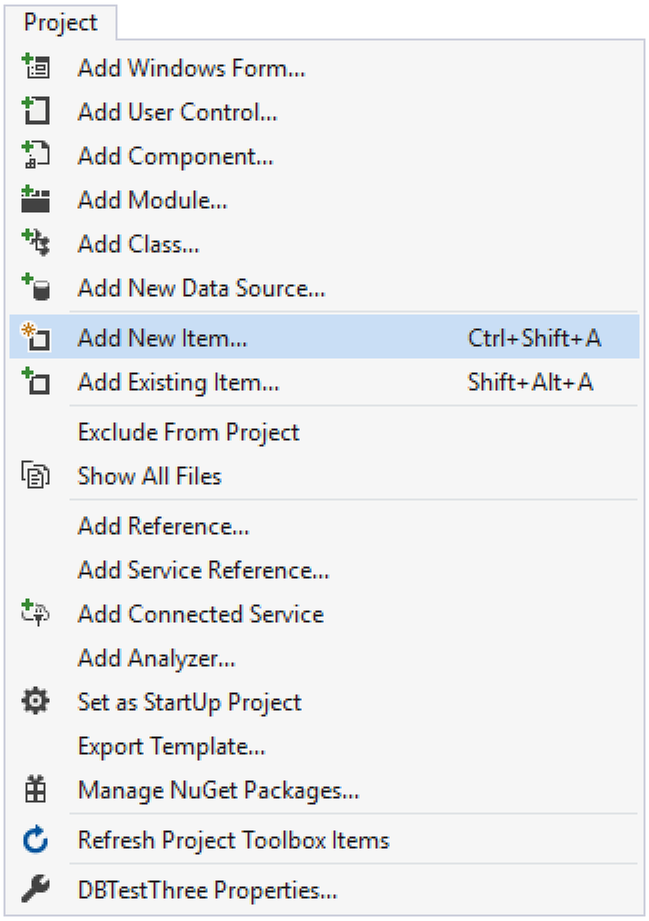
Creating the Visual Studio Database Project

In previous lessons, you learned how to set up a database with one table. In this section, you'll learn how to set up a database with more than one table, and write the code to display the results. Along the way, you'll learn about Foreign Keys, Joins, and Stored Procedures. As before, we'll use SQL Server as our database system.

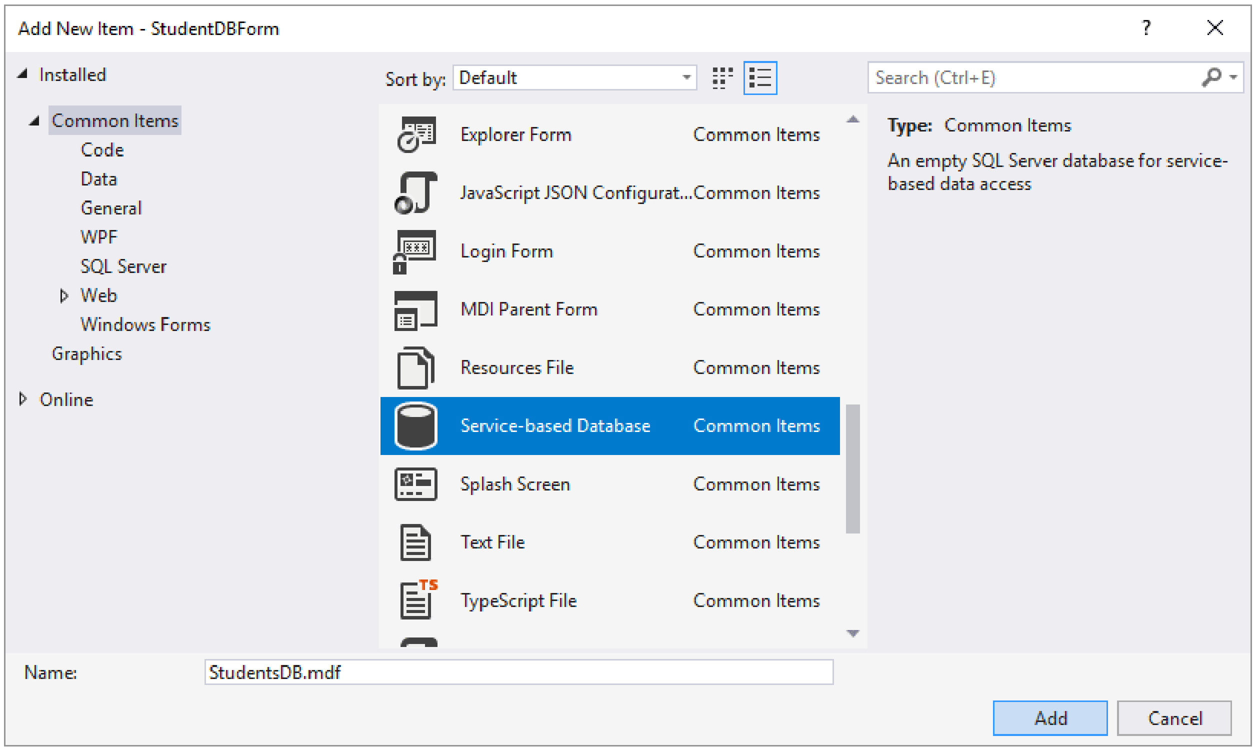
Create a new Windows Forms App project for VB or C# for this. (If you're using Visual Studio 2019, make sure you select either C# or Visual Basic from the Language dropdown at the top. This will narrow down your choices and ensure that you don't select a different form by mistake. Easy to do!)

For the project name, call it anything you like.

Now create a Service-based database by clicking **Project > Add New Item** from the menu at the top of Visual Studio:



From the Add New Item dialog box, select **Service-based database**. Change the name to **StudentsDB.mdf**:


Add New Item - StudentDBForm

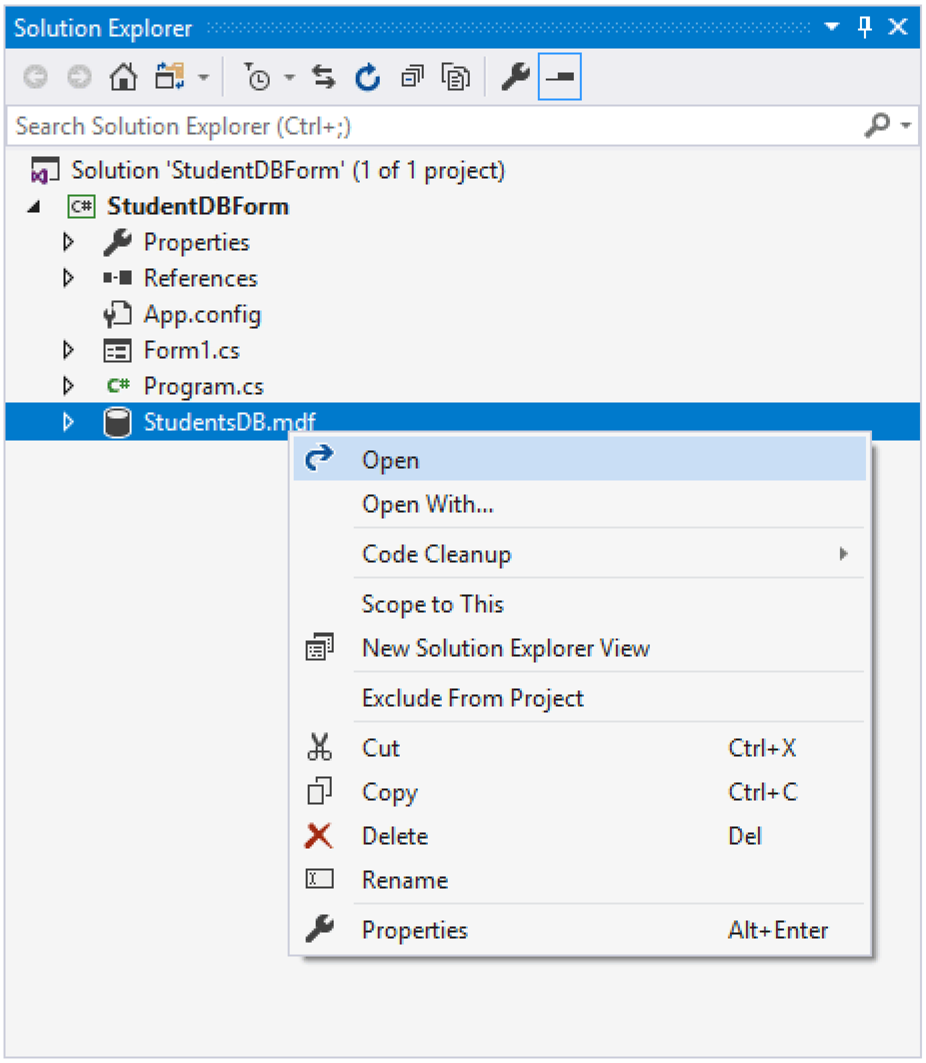
Explorer Form	Common Items
JavaScript JSON Configurat...	Common Items
Login Form	Common Items
MDI Parent Form	Common Items
Resources File	Common Items
Service-based Database	Common Items
Splash Screen	Common Items
Text File	Common Items
TypeScript File	Common Items

Type: Common Items

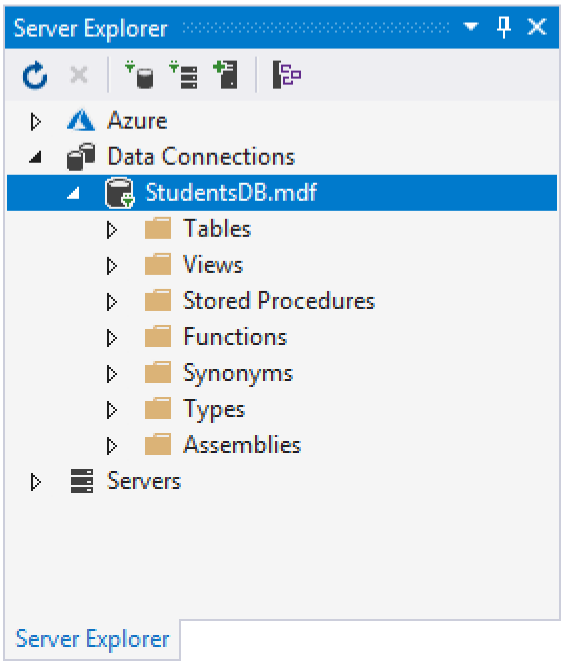
An empty SQL Server database for service-based data access

Name: StudentsDB.mdf

Once you have your database, right-click it in the Solution Explorer and select Open:



On the left of Visual Studio, have a look at the Server Explorer and the **Data Connections** item. (If you can't see the Server Explorer, click the **View** menu at the top of Visual Studio. From the View menu, select **Server Explorer**.) It should look like this:

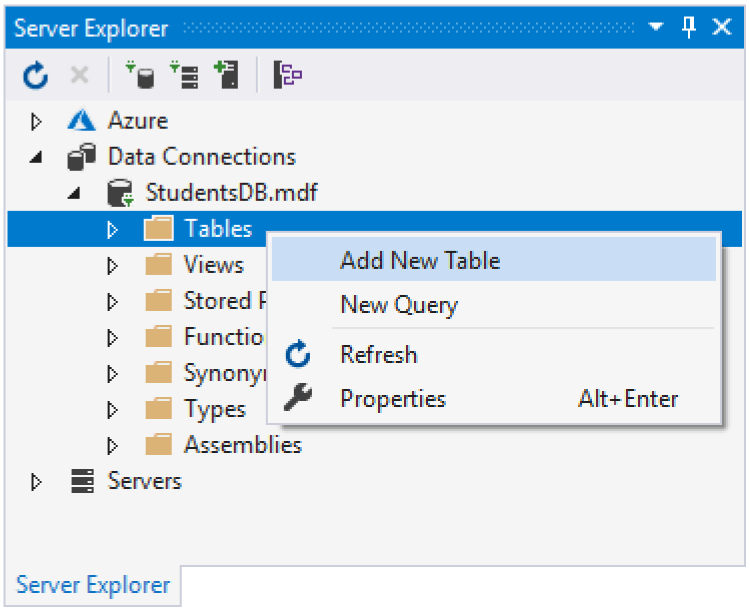


We're going to create two tables. One will hold a list of Students and the other will hold information about the courses the student is taking. We'll create the first table in the next lesson below.

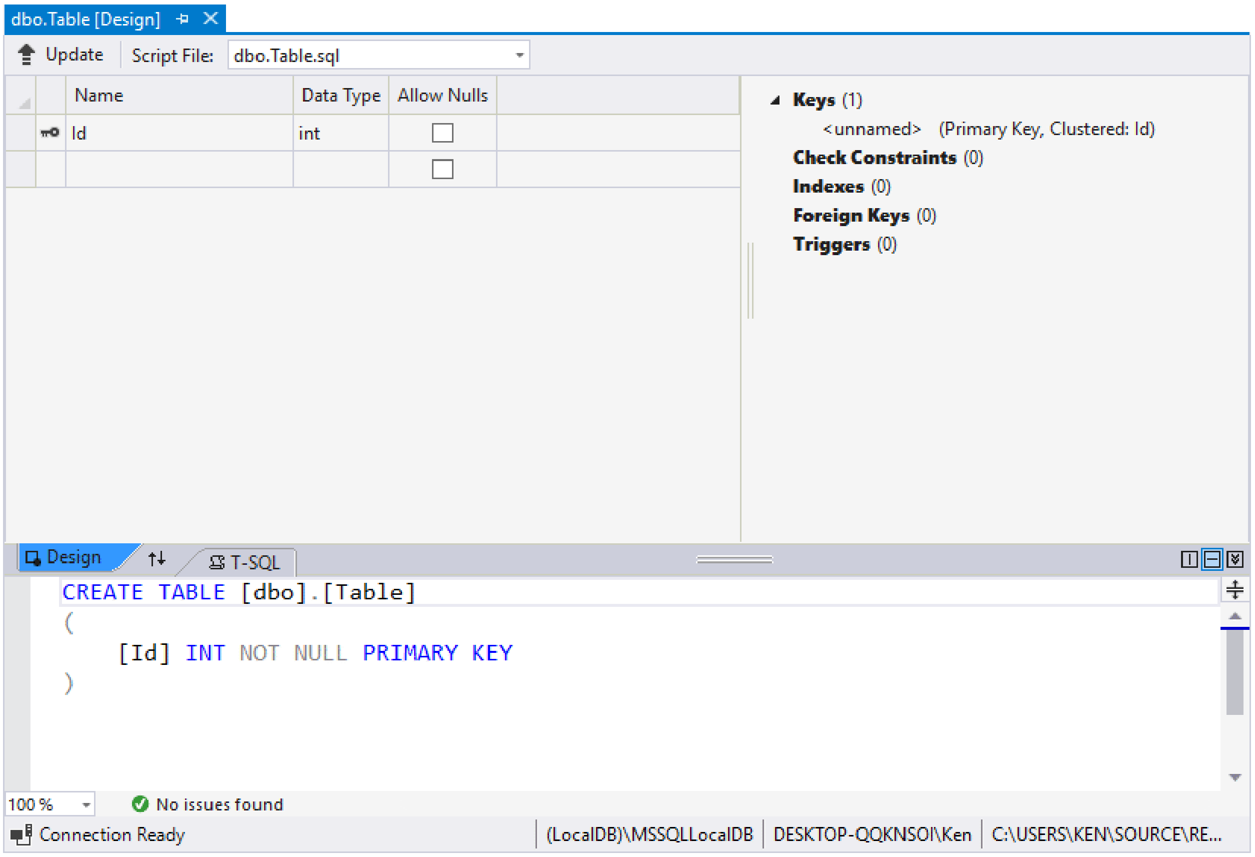
Creating the First Sql Server Database Table

We created the database in our previous lesson. In this lesson, we'll create the first table, which is going to hold alist of students.

In the Server Explorer on the left, right-click on **Tables** and select **Add New Table** from the menu:



When you click on **Add New Table**, the Table Designer will open up in the middle of Visual Studio. It should look like this:



Name	Data Type	Allow Nulls
Id	int	☐
		☐

Keys (1)
<unnamed> (Primary Key, Clustered: Id)
Check Constraints (0)
Indexes (0)
Foreign Keys (0)
Triggers (0)

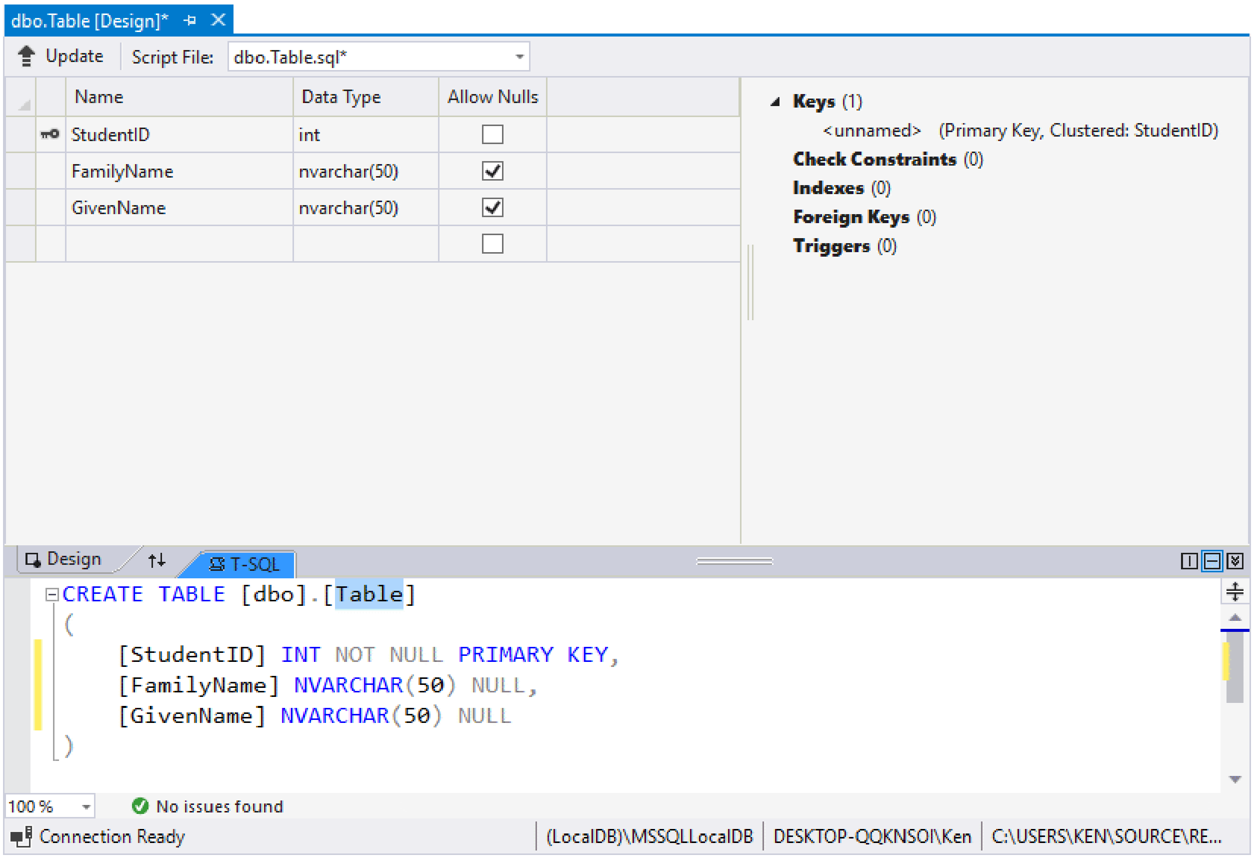
```
CREATE TABLE [dbo].[Table]
(
    [Id] INT NOT NULL PRIMARY KEY
)
```

Our Students table will contain an ID column, a Family Name column, and a Given Name column. The second table we'll create will link the two tables via the ID column. You'll see how this works in a moment. For now, let's add the columns to this table.

Click in the Id column, under Name. Notice the key symbol just to the left of it. This means that the Id column is the Primary Key, which is what we want. A Primary Key, remember, is one that uniquely identifies a row in the table. So you can't have two rows in the table with the same Id.

But change the Id column to **StudentID**. Keep the type on **int** and **Allow Nulls** should be left unchecked.

Click under the StudentID name and enter FamilyName as a new Column. Set its Data Type to NVARCHAR(50). Allow Nulls. Create a third Column. Set the Name to GivenName and the Data Type to NVARCHAR(50) again. Check Allow Nulls. Your Table Designer should look like this:


dbo.Table [Design]*

Update | Script File: dbo.Table.sql*

	Name	Data Type	Allow Nulls
🔑	StudentID	int	☐
	FamilyName	nvarchar(50)	✓
	GivenName	nvarchar(50)	✓
			☐

Keys (1)
<unnamed> (Primary Key, Clustered: StudentID)
Check Constraints (0)
Indexes (0)
Foreign Keys (0)
Triggers (0)

Design | T-SQL

```
CREATE TABLE [dbo].[Table]
(
    [StudentID] INT NOT NULL PRIMARY KEY,
    [FamilyName] NVARCHAR(50) NULL,
    [GivenName] NVARCHAR(50) NULL
)
```

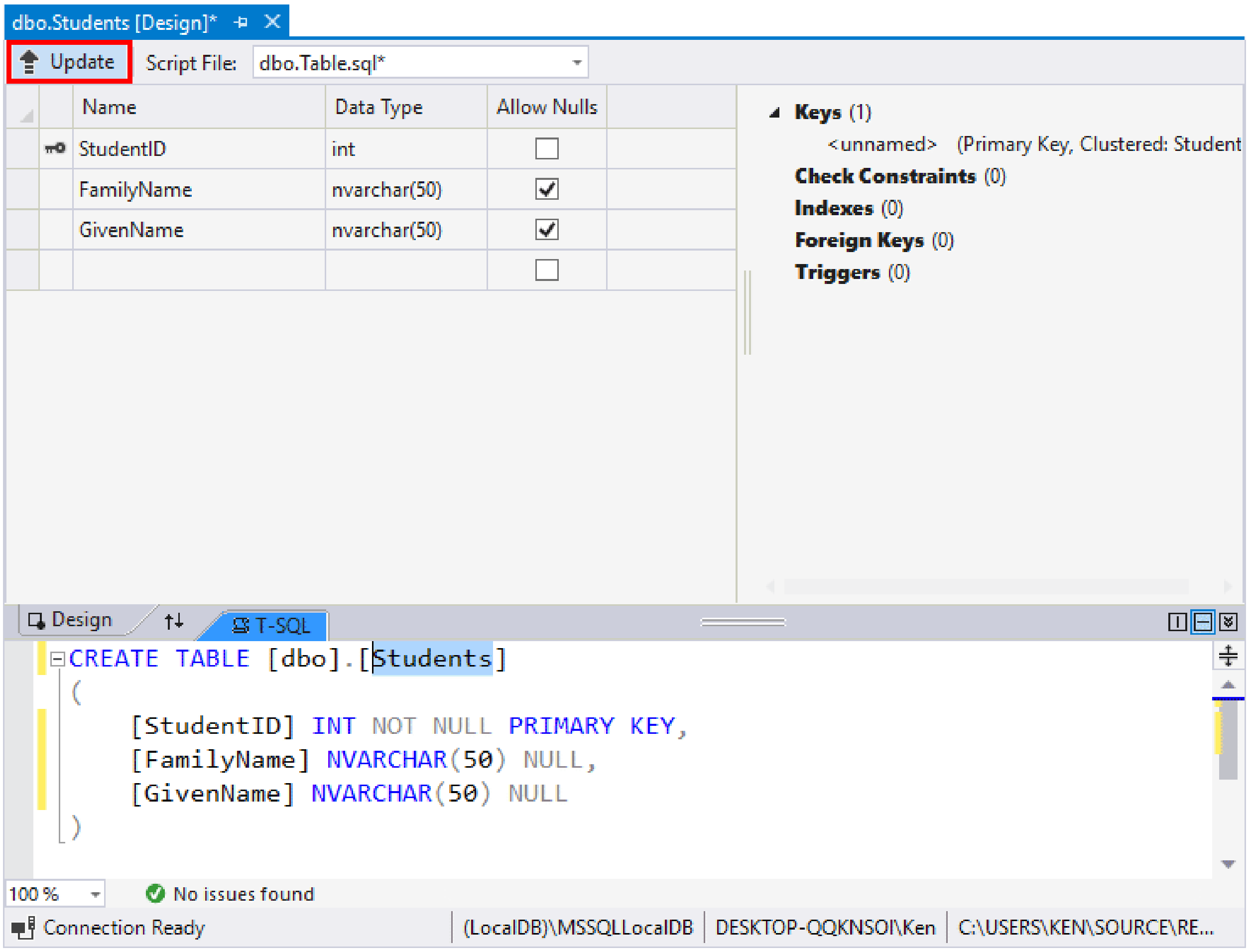
100 % | No issues found

Connection Ready | (LocalDB)\MSSQLLocalDB | DESKTOP-QQKNSOI\Ken | C:\USERS\KEN\SOURCE\RE...

Notice the bottom half of the Table Designer, where it says CREATE TABLE. This is T-SQL. The table name has a default name of [dbo].[TABLE] (the one highlighted). Change **Table** to **Students**. This will be the name of our first table:

```
CREATE TABLE [dbo].[Students]
(
    [StudentID] INT NOT NULL PRIMARY KEY,
    [FamilyName] NVARCHAR(50) NULL,
    [GivenName] NVARCHAR(50) NULL
)
```

Once you've changed the name of the table, click the Update button in the top left:

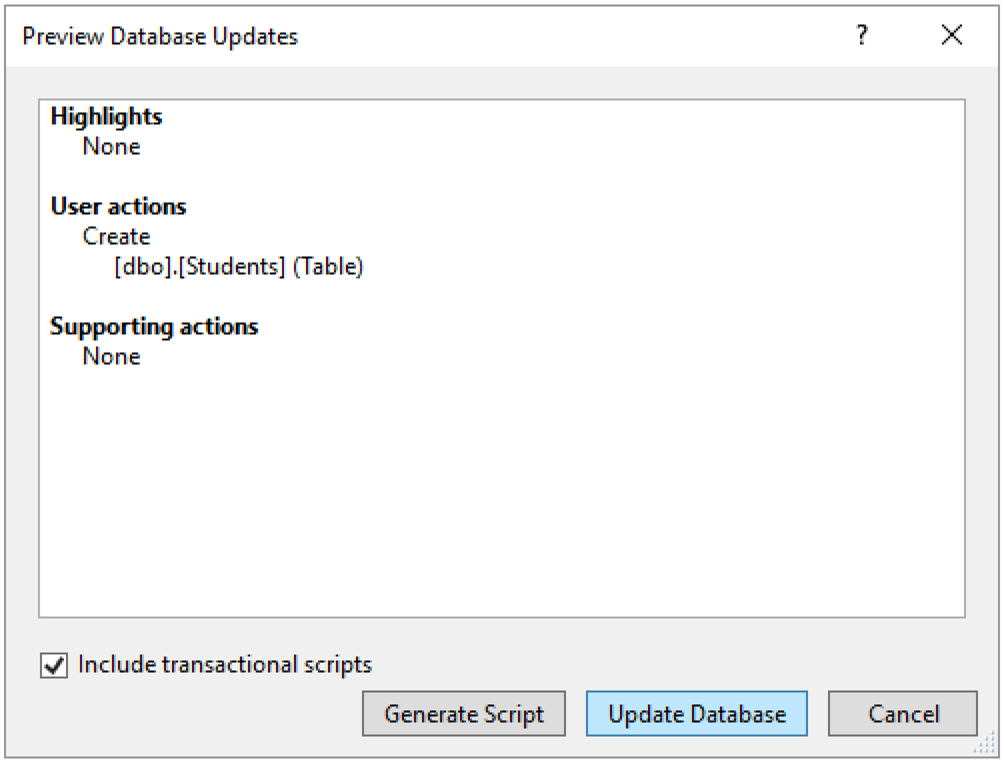


Name	Data Type	Allow Nulls
StudentID	int	☐
FamilyName	nvarchar(50)	✓
GivenName	nvarchar(50)	✓
		☐

Keys (1)
<unnamed> (Primary Key, Clustered: Student
Check Constraints (0)
Indexes (0)
Foreign Keys (0)
Triggers (0)

```
CREATE TABLE [dbo].[Students]
(
    [StudentID] INT NOT NULL PRIMARY KEY,
    [FamilyName] NVARCHAR(50) NULL,
    [GivenName] NVARCHAR(50) NULL
)
```

When you click Update, you should see a dialog box appear. This one:



Preview Database Updates

Highlights
None

User actions
Create
[dbo].[Students] (Table)

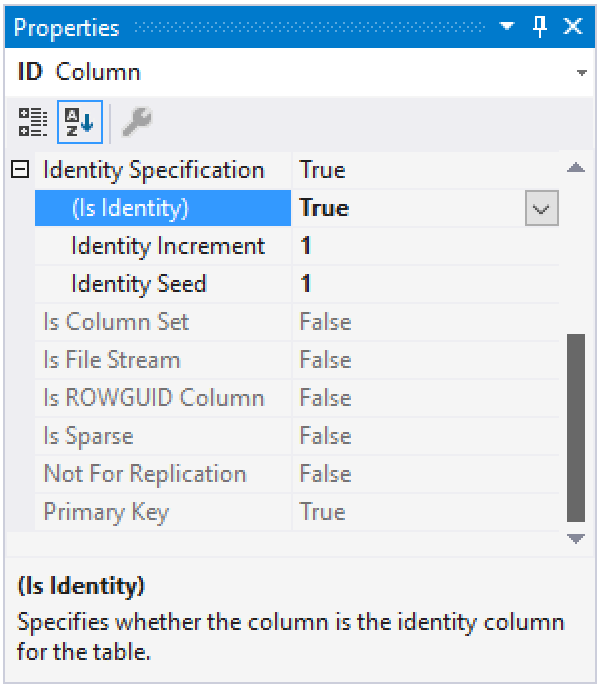
Supporting actions
None

Click the **Update Database** button.

Now that we have one table set up, let's do the other one. We're going to set up a relation between the two tables, with the help of a Foreign Key. We'll do that in the next lesson below.

The Second Database Table - Foreign Keys

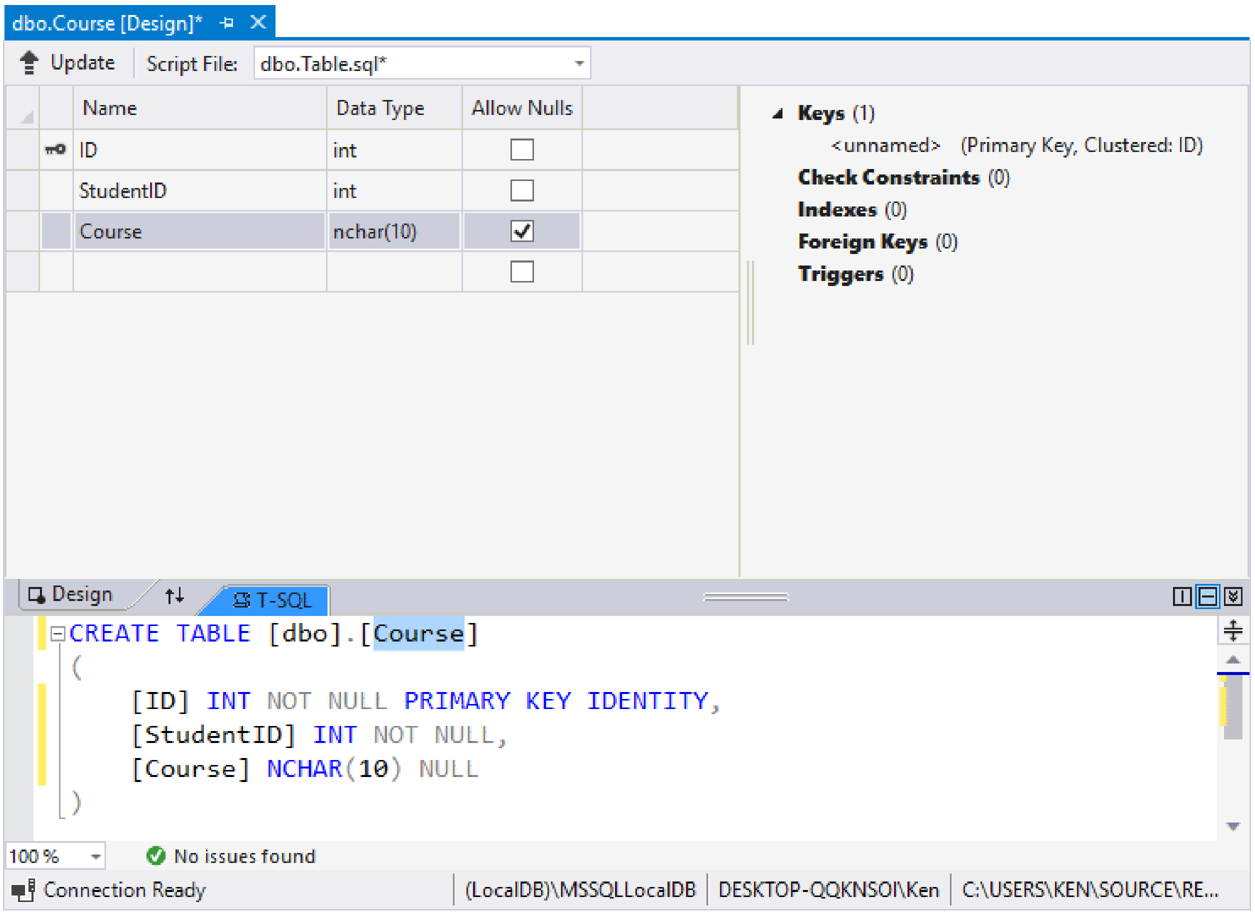
Create a second table, just like you did for the first one. You can leave the ID column in place. But notice we've change it to all capital letters. The default is capital "I" lowercase "d". Keep it on **int** as the Data Type and Allow Nulls unchecked. We want this ID column to **autoincrement**, so change **IsIdentity** (under Identity Specification) to **True** in the Properties area:



Identity Specification	True
(Is Identity)	True
Identity Increment	1
Identity Seed	1
Is Column Set	False
Is File Stream	False
Is ROWGUID Column	False
Is Sparse	False
Not For Replication	False
Primary Key	True

(Is Identity)
Specifies whether the column is the identity column for the table.

Add another Column. Call this one **StudentID**. Set the Data Type to **int** and leave Allow Nulls unchecked. Add a third Column. Call it **Courses**. Set the Data type to **NVARCHAR(10)** and check the Allow Nulls box. In the T-SQL area at the bottom, change [dbo].[**Table**] to [dbo].[**Course**]. Your Table Design window should look like this:



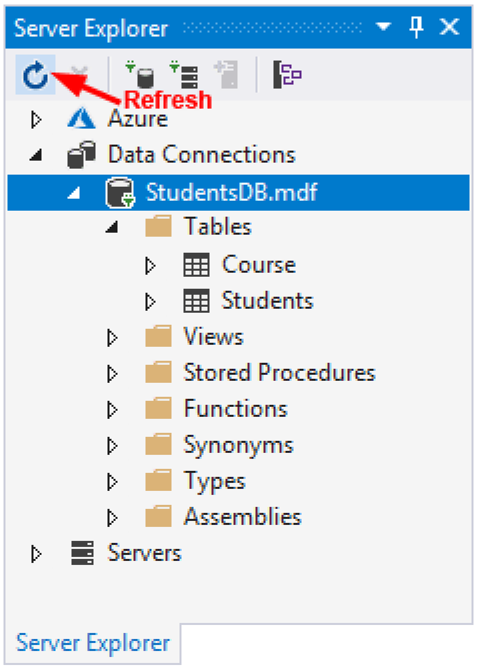
Name	Data Type	Allow Nulls
ID	int	☐
StudentID	int	☐
Course	nchar(10)	✓
		☐

Keys (1)
<unnamed> (Primary Key, Clustered: ID)
Check Constraints (0)
Indexes (0)
Foreign Keys (0)
Triggers (0)

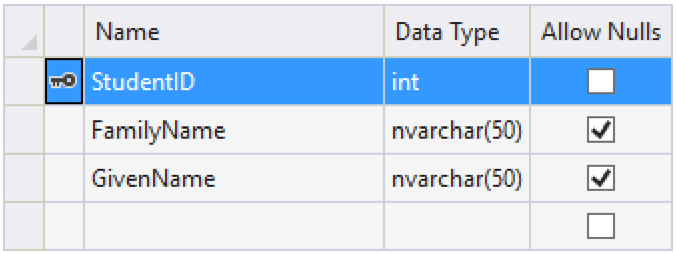
```
CREATE TABLE [dbo].[Course]
(
    [ID] INT NOT NULL PRIMARY KEY IDENTITY,
    [StudentID] INT NOT NULL,
    [Course] NCHAR(10) NULL
)
```

Now click the Update button in the top left.

In the Server Explorer on the left, highlight your table name and click the Refresh button to see your new table:

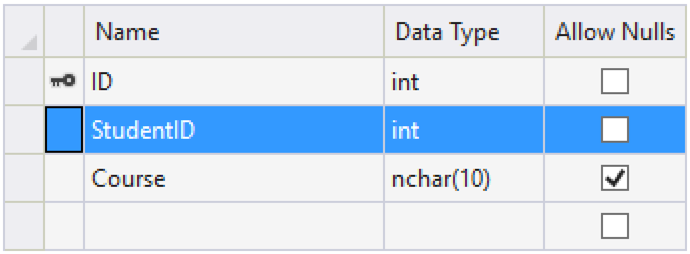


You may have noticed that we have a StudentID column in both tables. This is deliberate. We want to link both tables. And the way you do this is with something called a Foreign Key. The Foreign Key in one table is usually the Primary Key in another table. For example, here's the Students table again:



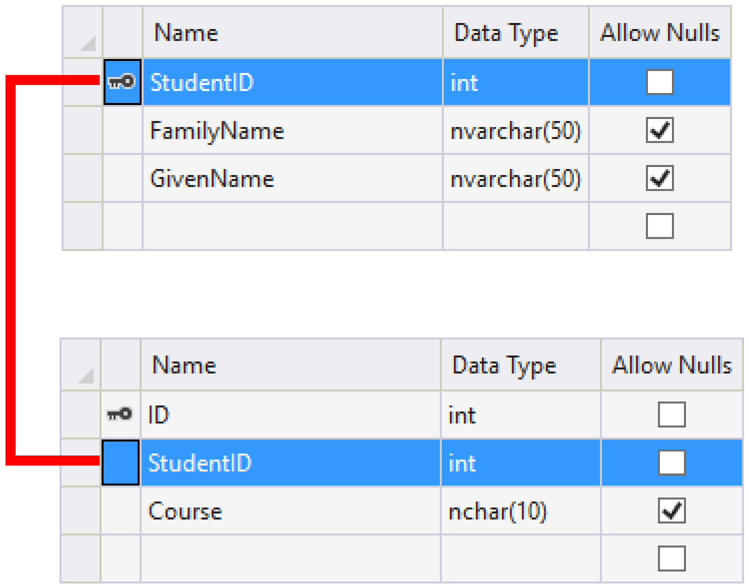
Name	Data Type	Allow Nulls
StudentID	int	☐
FamilyName	nvarchar(50)	✓
GivenName	nvarchar(50)	✓
		☐

And here's the Course table:



Name	Data Type	Allow Nulls
ID	int	☐
StudentID	int	☐
Course	nchar(10)	✓
		☐

We want to link a student in the **Students** table to one or more courses in the Course table. If we ensure that the StudentID values are the same in both tables then we have our link. All we need to do to link the two is to make the StudentID in the Course table a Foreign Key. The link would look like this:



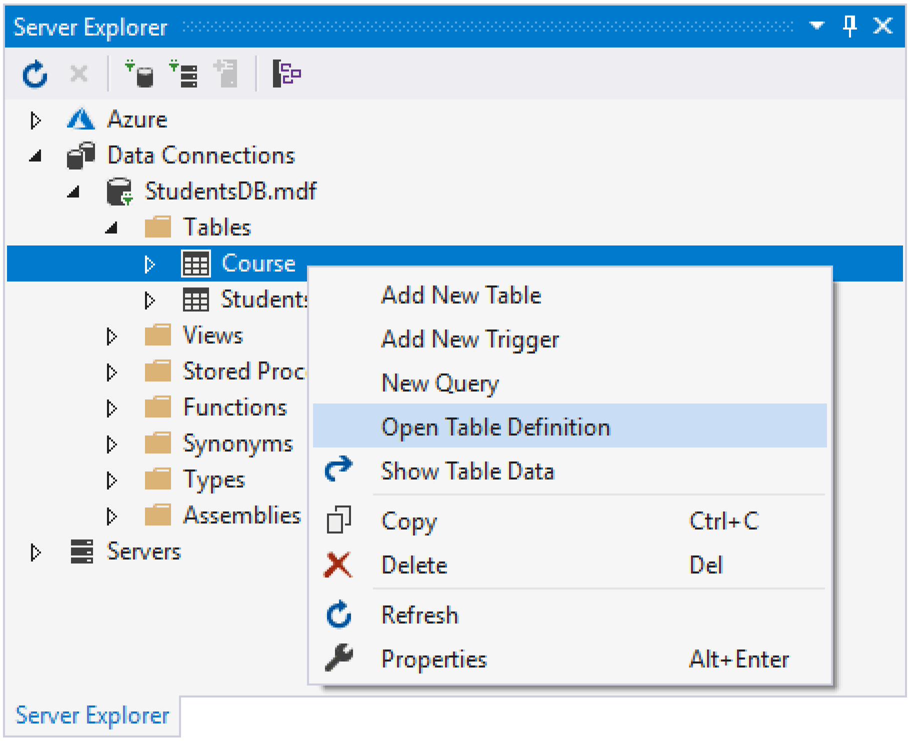
Name	Data Type	Allow Nulls
StudentID	int	☐
FamilyName	nvarchar(50)	✓
GivenName	nvarchar(50)	✓
		☐

Name	Data Type	Allow Nulls
ID	int	☐
StudentID	int	☐
Course	nchar(10)	✓
		☐

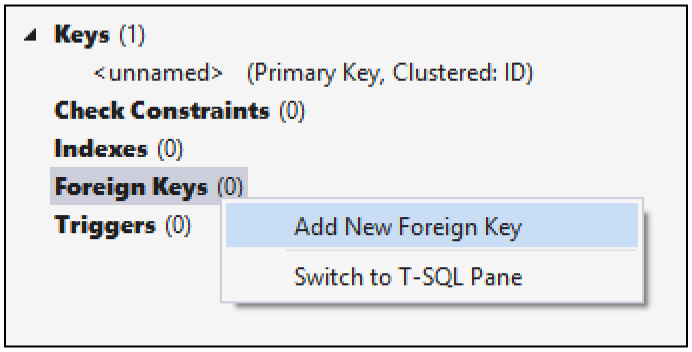
We can enter 10, say, as the value for the StudentID Columns in both tables. We can then tell SQL Server to, "Fetch all the records from both tables where the StudentID values are both 10".

In order to do that, we need to set the StudentID column in the Course table to be a Foreign Key. That way, the Primary Key in the first table becomes a Foreign Key in the Second table. (This, by the way, is known as a One to Many Relationship. Because one record in the first table can have many matching records in the second table. They will match because they'll both have the same StudentID values.)

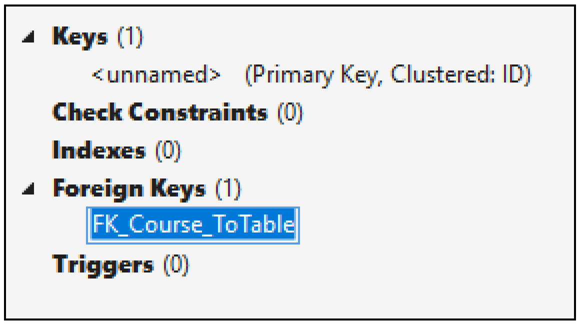
To set a Foreign Key, make sure your Course table is displayed in the Table Design window. If it's not, right-click the table in the Server Explorer. From the menu, select **Open Table Definition**:



In the Table Designer, just to the right of the table definition, you'll see an area that has items for Keys, Check Constraints, Indexes, Foreign Keys, and Triggers. Right-click the Foreign Keys item and select **Add New Foreign Key**:

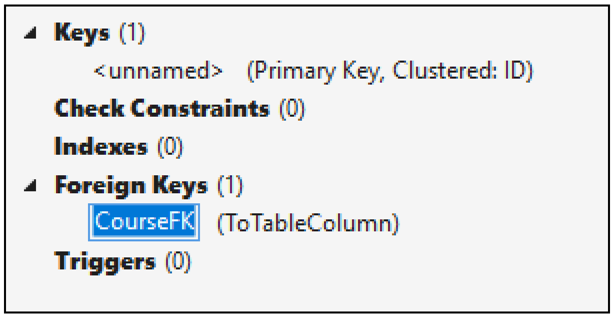


You'll see a small textbox:



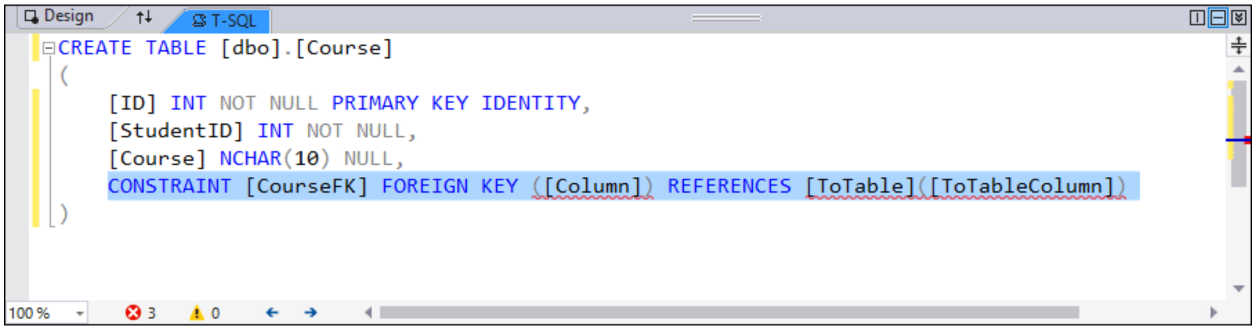
```
Keys (1)
    <unnamed>   (Primary Key, Clustered: ID)
Check Constraints (0)
Indexes (0)
Foreign Keys (1)
    FK_Course_ToTable
Triggers (0)
```

FK_Course_ToTable is a default name. You can change this to anything you like. Type **CourseFK** instead. Press the enter key on your keyboard and the Foreign Key area will look like this:



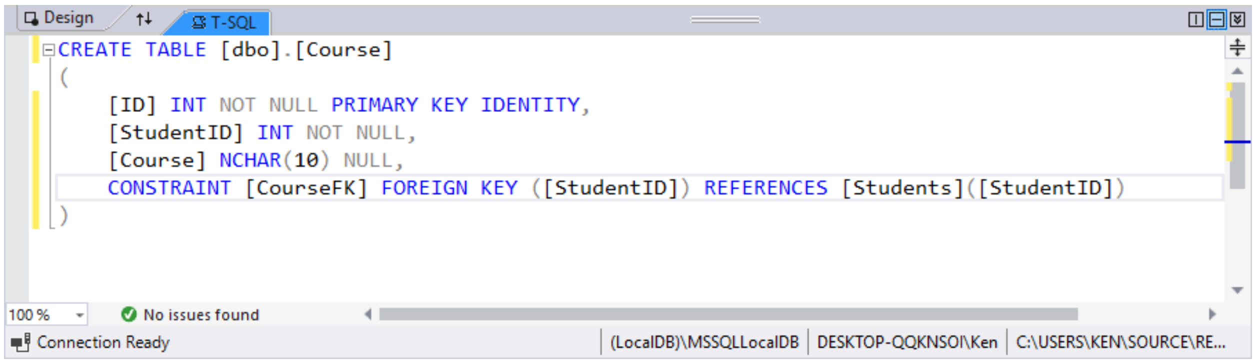
```
Keys (1)
    <unnamed>   (Primary Key, Clustered: ID)
Check Constraints (0)
Indexes (0)
Foreign Keys (1)
    CourseFK   (ToTableColumn)
Triggers (0)
```

Now look at the T-SQL area at the bottom. You should see this rather complicated addition, with some red underlines:



```
CREATE TABLE [dbo].[Course]
(
    [ID] INT NOT NULL PRIMARY KEY IDENTITY,
    [StudentID] INT NOT NULL,
    [Course] NCHAR(10) NULL,
    CONSTRAINT [CourseFK] FOREIGN KEY ([Column]) REFERENCES [ToTable]([ToTableColumn])
)
```

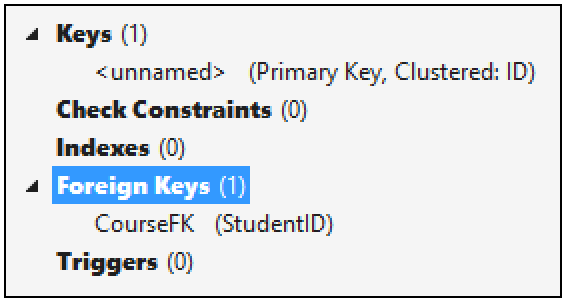
The Constraint is the Foreign Key, and CourseFK is the name we just gave it. After the T-SQL keywords FOREIGN KEY, you need the name of the column where you have your Foreign Key. Change the word **Column** to **StudentID**. This REFERENCES a column in the Students table. So change [ToTable].([ToColumn]) to [Students].([StudentId]). You T-SQL area should look like this:



```
CREATE TABLE [dbo].[Course]
(
    [ID] INT NOT NULL PRIMARY KEY IDENTITY,
    [StudentID] INT NOT NULL,
    [Course] NCHAR(10) NULL,
    CONSTRAINT [CourseFK] FOREIGN KEY ([StudentID]) REFERENCES [Students]([StudentID])
)
```

The red underlines will go away, if you get it right. If you do, save your changes by clicking the Update button in the top left.

If you look top right, you should see a 1 appear next to Foreign Keys:



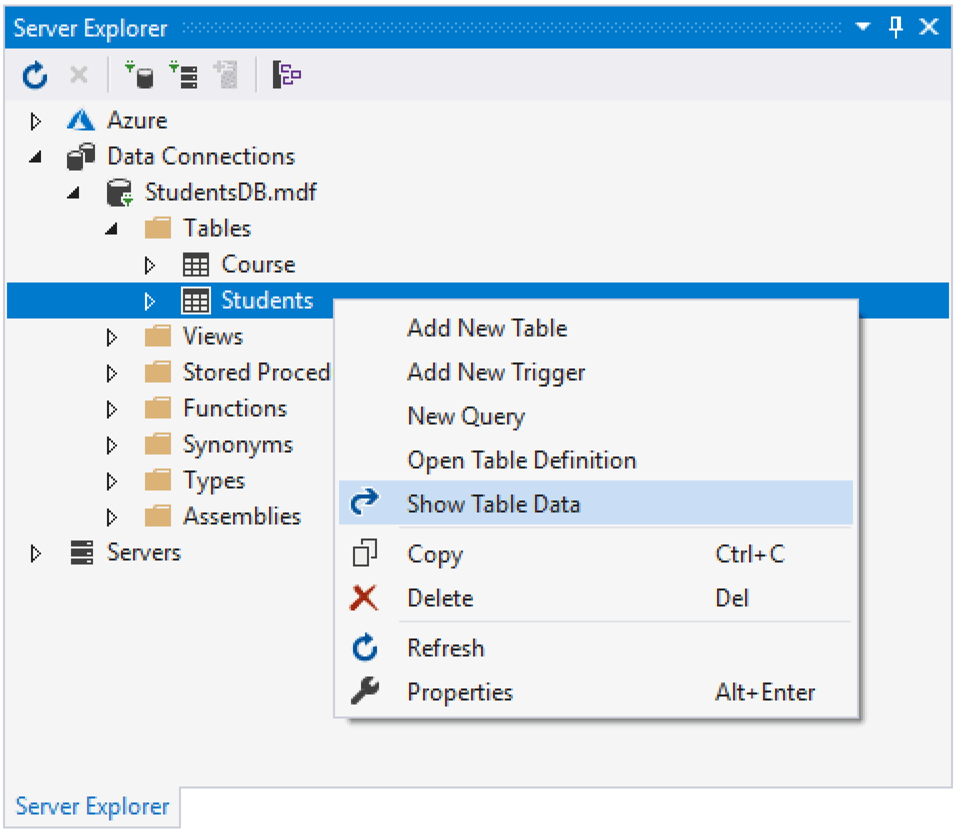
```
◢ Keys (1)
      <unnamed>   (Primary Key, Clustered: ID)
  Check Constraints (0)
  Indexes (0)
◢ Foreign Keys (1)
      CourseFK   (StudentID)
  Triggers (0)
```

Now that we've set up our two tables, we can enter some data into them. We'll do that in the next lesson below.

Adding Data to the Two Tables

In previous lessons, we set up our two tables. Now, we can enter some data into them.

In the Server Explorer area, right-click the Students table. From the menu, select **Show Table Data**:



Start entering data. For the StudentID column, you could have some complex ID here, but we'll keep it simple. Enter a value of 10 as the first ID, then a name:

10, Carney Kenny

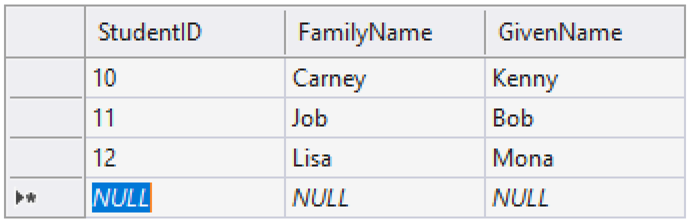
Enter another student, anything you like. But have the ID on 11. You don't have to keep the numbers sequential, but it will help us to remember them for this tutorial:

11, Job, Bob

Let's have a third Student:

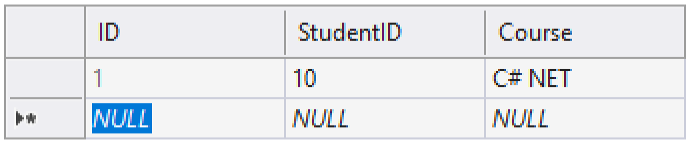
12, Lisa, Mona

Your Table Data area should look something like this:



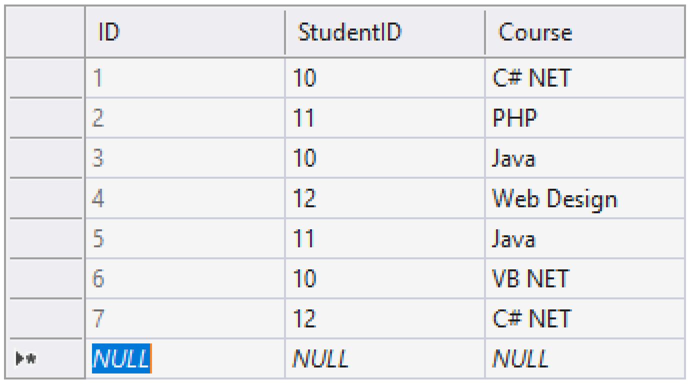
	StudentID	FamilyName	GivenName
	10	Carney	Kenny
	11	Job	Bob
	12	Lisa	Mona
▸*	NULL	NULL	NULL

Now open up the Course table. You don't have to enter anything in the ID column because it's set to autoincrement, meaning it will take care of itself. Click into the StudentID column and enter 10. A value of 10 was the value we gave to our first student. In the Course column, enter a course like C# NET:



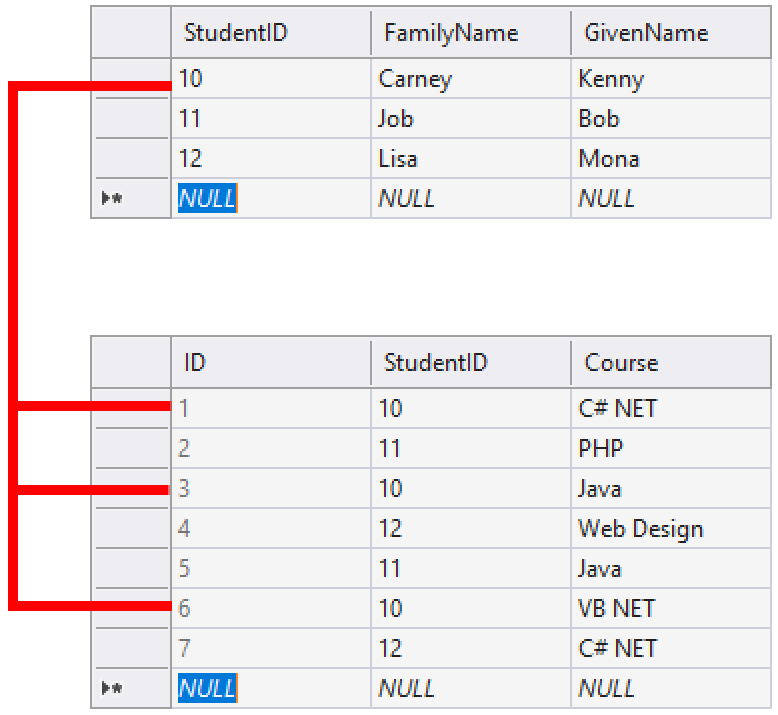
	ID	StudentID	Course
	1	10	C# NET
▸*	NULL	NULL	NULL

Now enter more values. (You can reuse the 10 value as many times as you like.) Enter courses for students 11 and 12. You might have something like this, when you're done:



	ID	StudentID	Course
	1	10	C# NET
	2	11	PHP
	3	10	Java
	4	12	Web Design
	5	11	Java
	6	10	VB NET
	7	12	C# NET
▸*	NULL	NULL	NULL

And here's a visual representation of what we're trying to do here:



	StudentID	FamilyName	GivenName
	10	Carney	Kenny
	11	Job	Bob
	12	Lisa	Mona
▶*	NULL	NULL	NULL

	ID	StudentID	Course
	1	10	C# NET
	2	11	PHP
	3	10	Java
	4	12	Web Design
	5	11	Java
	6	10	VB NET
	7	12	C# NET
▶*	NULL	NULL	NULL

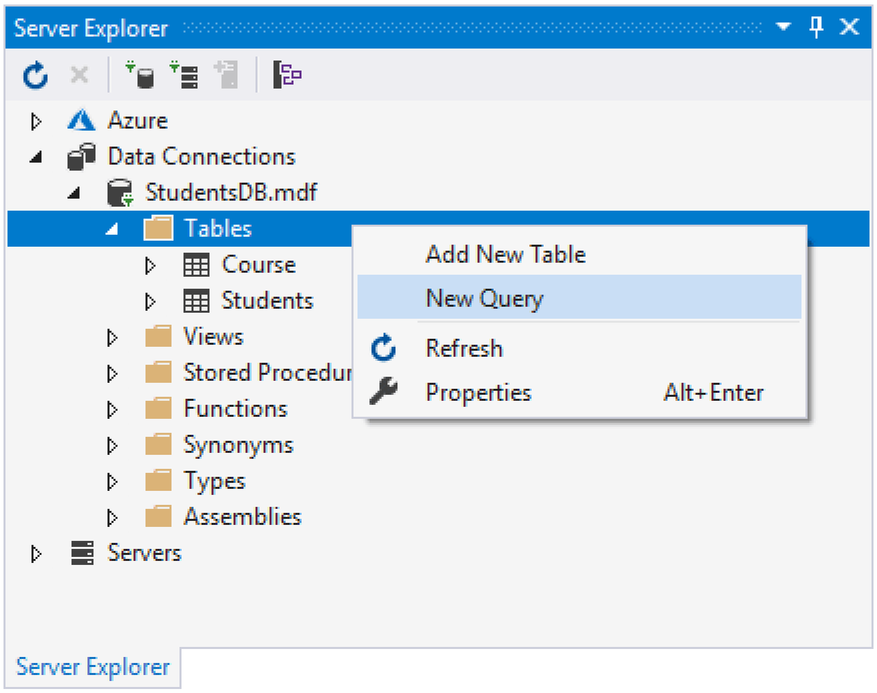
This is the One to Many relationship mentioned earlier. The student called Kenny Carney, with an ID of 10, is linked to three courses: C# NET, Java, and VB NET.
The point of doing things this way is so that we can write some SQL to get at the records from both tables. Let's do that now as we turn to SQL Queries.

SQL Queries, Joins, Stored Procedures

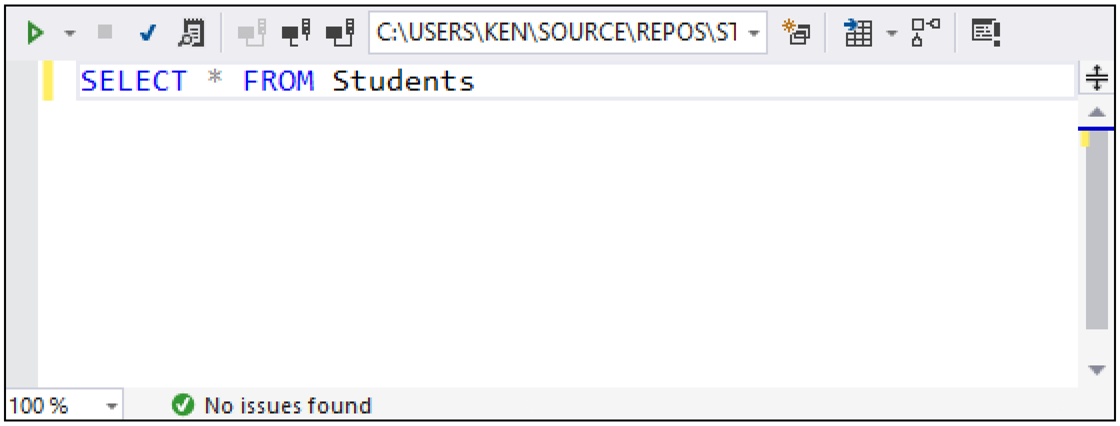
You've used SQL queries in previous lessons. You selected all the record from a single table with something like this:

SELECT * FROM Students

You can try that again. In the Server Explorer window, right click on the Tables folder and Select New Query from the menu:



In the Query window enter the above SQL. You query window should look like this:

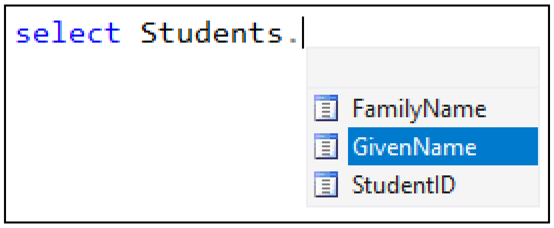


```
SELECT * FROM Students
```

(SQL Commands are not case-sensitive, by the way. You could enter the SELECT and the FROM in lowercase, if you wanted to.)

Click on the Green Arrow in the top left to run the query. You should see all the records from the Students table displayed. Now change the Students to Course. Run the query again and you'll see all the records from the Course table displayed.

You don't have to select all the records. You can specify the columns you want by typing the table name first, then a dot. You should see a list of available columns in your table:



```
select Students.
```

Select the column you want in your results. To add more columns, separate them with commas:

```
SELECT Students.GivenName, Students.FamilyName
FROM Students
```

However, what we want to do is to display results from both tables. To do that, we need something called a JOIN.

Sql Server Database JOIN

If you wanted to display only certain selected columns from both tables, you could do it like this:

```
SELECT Students.GivenName, Students.FamilyName,
Course.Course
FROM Students, Course
WHERE Students.StudentID = Course.StudentID
```

Here, we're selecting only three columns FROM both tables. Notice that the table names are separated by commas. (The dbo part is the schema that SQL Server gives us by default.) The WHERE clause at the end tells SQL Server to only display the records where the StudentID columns match in both tables.

In fact, try that out. Open a New Query and type the above into the window. Run your query to see the results.

There is another way to do this, though: use a JOIN.

There are a few different types of JOIN (inner, outer, cross, left, right, full) but they all work with a Primary Key in one Table and a Foreign Key in another. The difference is in how many results you want back, and whether to include NULL records or not.

The one we'll use is an INNER JOIN. The format is this:

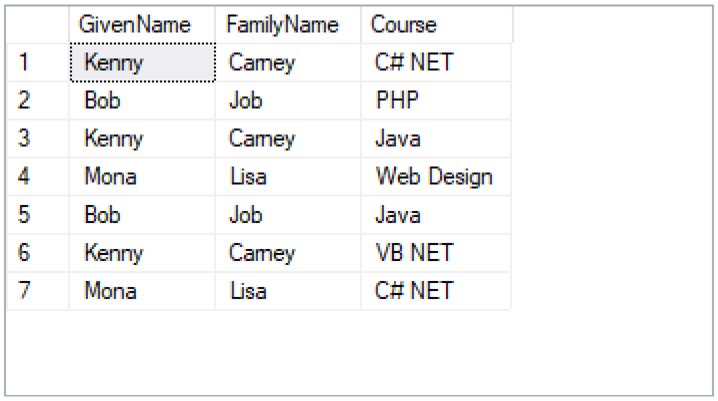
```
SELECT [Columns]
FROM [Parent Table]
INNER JOIN [CHILD TABLE]
ON [Primary Key = Foreign Key]
```

The SQL Keywords are SELECT, FROM, INNER JOIN, ON. The parent table is the one with the StudentID Primary Key. The child table is the one with the StudentID as the Foreign Key. The INNER JOIN is ON the two.

To try it out, create a new query. Now enter the following: (You don't have to have the SQL on separate lines, but it does make it more readable.)

```
SELECT Students.GivenName, Students.FamilyName,
Course.Course
FROM Students
INNER JOIN Course
ON Students.StudentID = Course.StudentID
```

Run your query and you'll see the results appear:



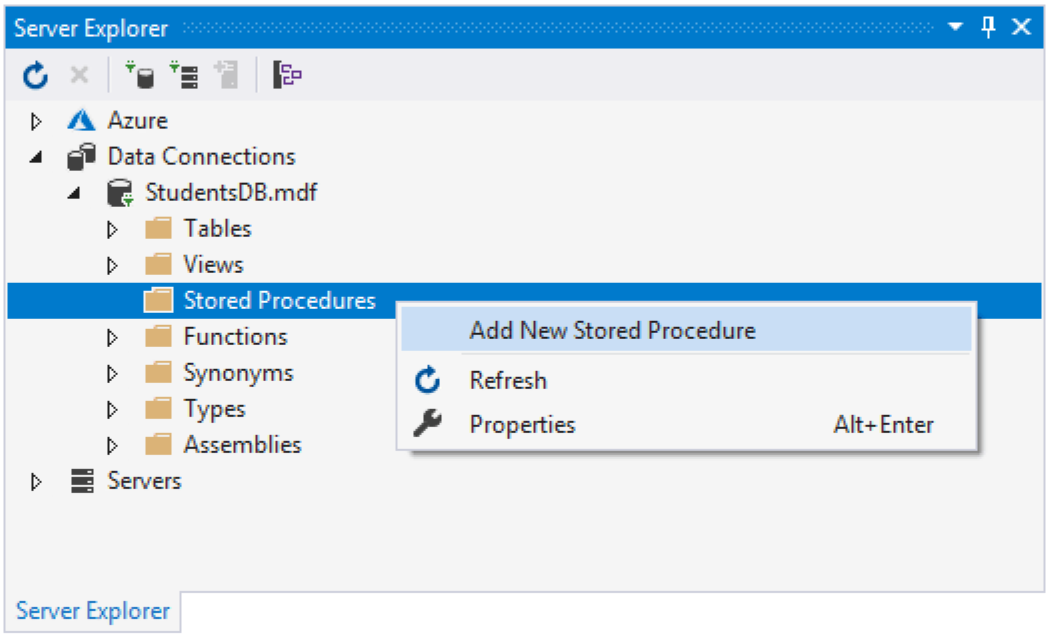
	GivenName	FamilyName	Course
1	Kenny	Carney	C# NET
2	Bob	Job	PHP
3	Kenny	Carney	Java
4	Mona	Lisa	Web Design
5	Bob	Job	Java
6	Kenny	Carney	VB NET
7	Mona	Lisa	C# NET

You can actually miss out the INNER part at the start. So just have JOIN instead of INNER JOIN. The result is the same.

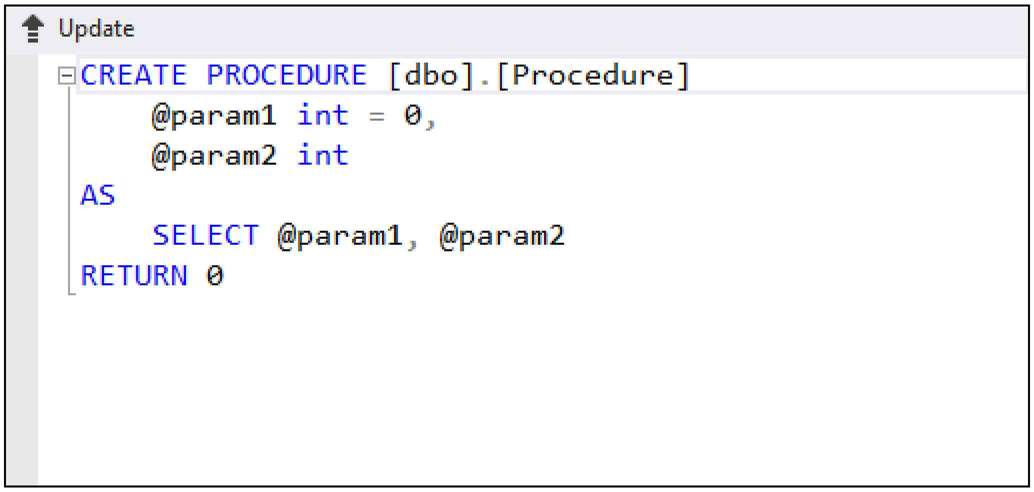
The point about learning all this SQL is that you can use it in your programming code to pull records from your database. To make your life easier as a programmer, and to help with security issues, you can create something called a Stored Procedure in your database. You can then grab your Stored Procedures with code. Let's create one.

Stored Procedures

To create a Stored Procedure, right-click the Stored Procedure folder in the Server Explorer. From the menu select **Add New Stored Procedure**:



A new window will appear with some complex-looking SQL. This:

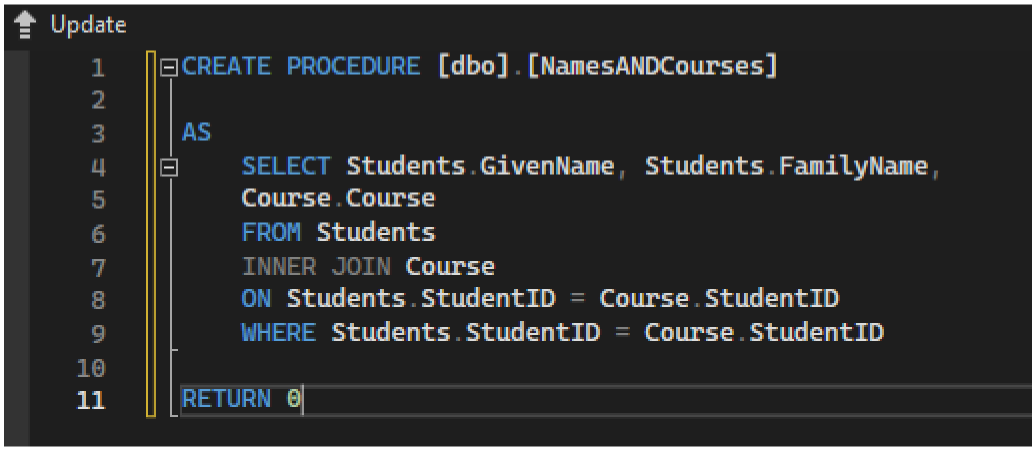


```
CREATE PROCEDURE [dbo].[Procedure]
    @param1 int = 0,
    @param2 int
AS
    SELECT @param1, @param2
RETURN 0
```

The CREATE line is where you can change the name of your procedure. Click into the square brackets and delete the Word Procedure. Type a new name. Type NamesAndCourses:

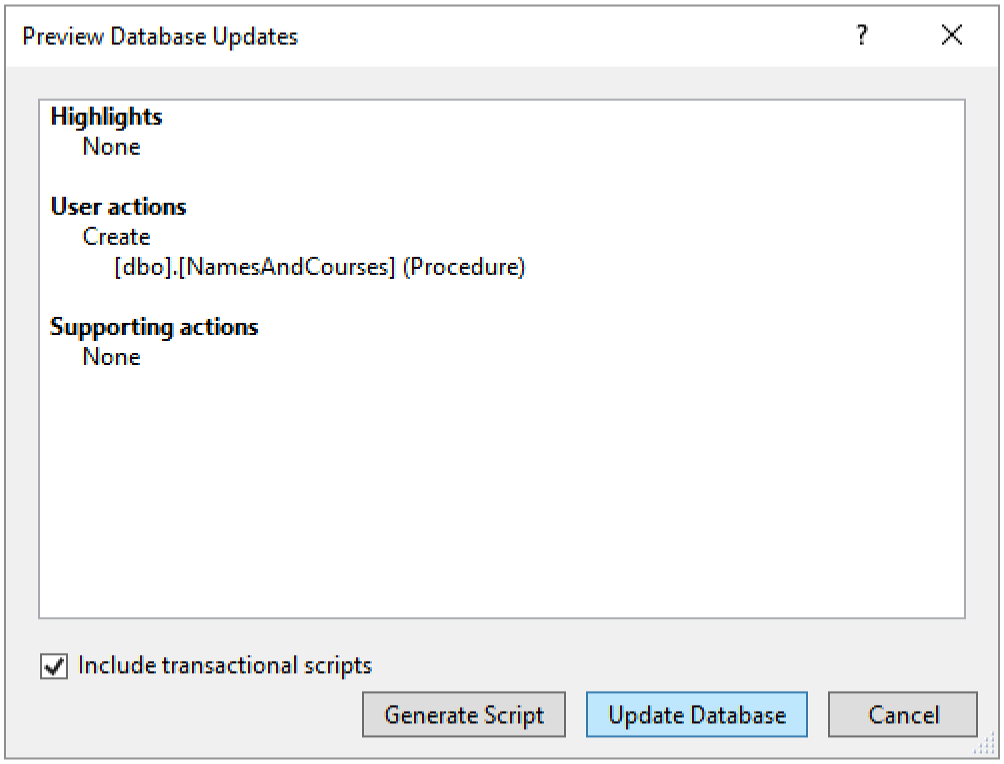
CREATE PROCEDURE [dbo].[NamesAndCourses]

We'll get onto parameters shortly. For now, delete the two lines that start with @param. Leave the As keyword. In place of the SELECT line, copy and paste your Join SQL. You should now have this:



```
CREATE PROCEDURE [dbo].[NamesANDCourses]

AS
    SELECT Students.GivenName, Students.FamilyName,
    Course.Course
    FROM Students
    INNER JOIN Course
    ON Students.StudentID = Course.StudentID
    WHERE Students.StudentID = Course.StudentID

RETURN 0
```

Click the Update button and you'll see the Preview box appear:



Highlights
None

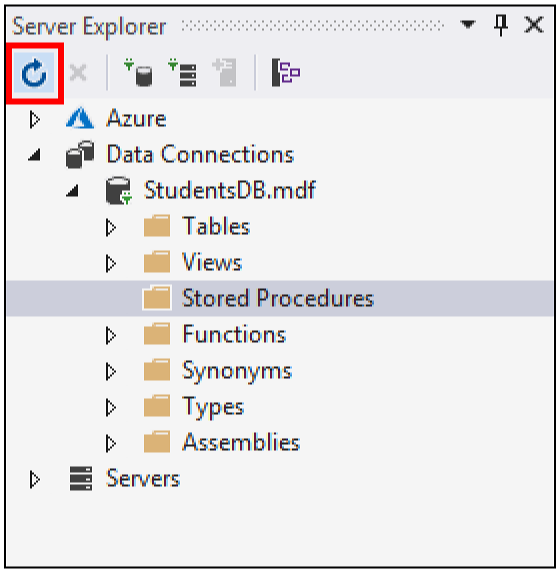
User actions
Create
[dbo].[NamesAndCourses] (Procedure)

Supporting actions
None

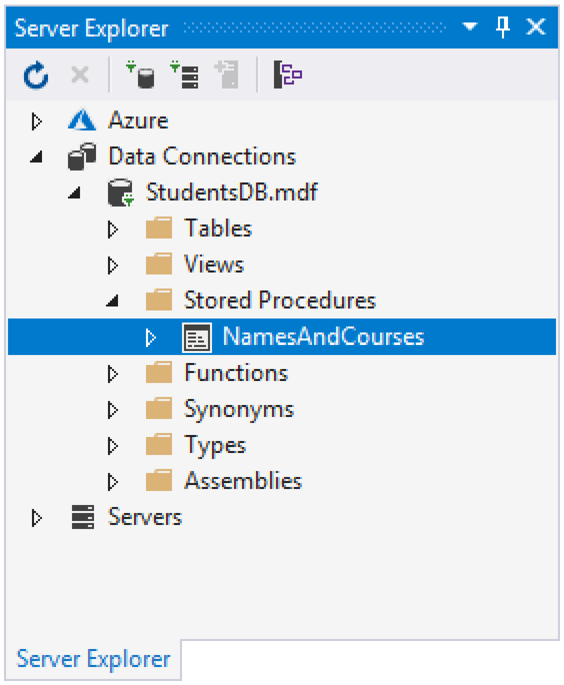
☑ Include transactional scripts

Click Update Database.

In the Server Explorer on the left click the Refresh button:

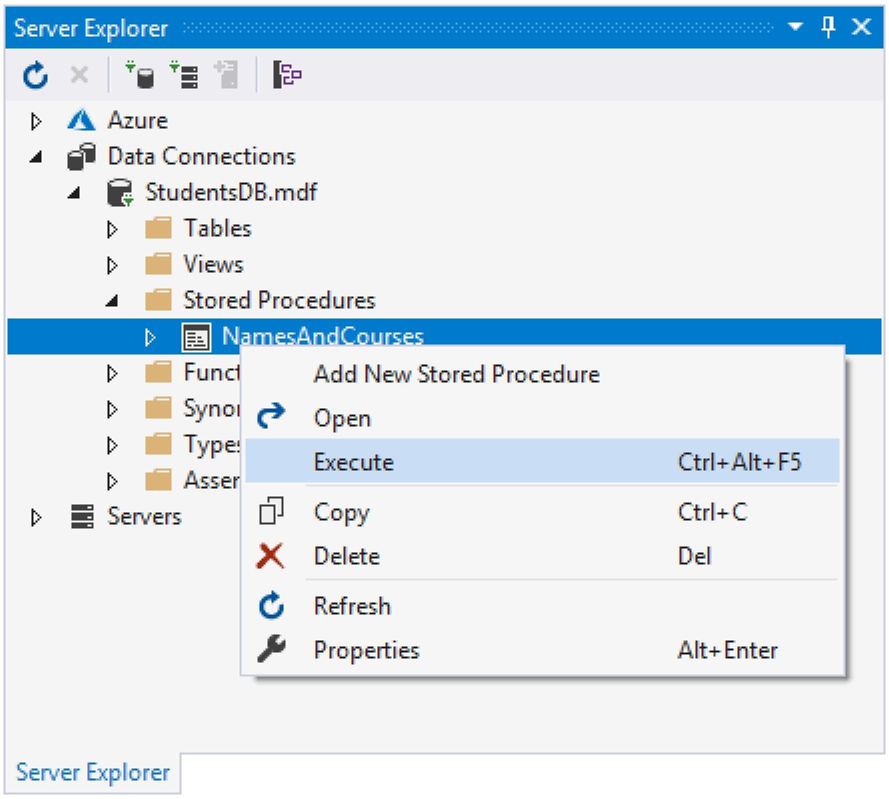


You should now see your new Stored Procedure in the folder:

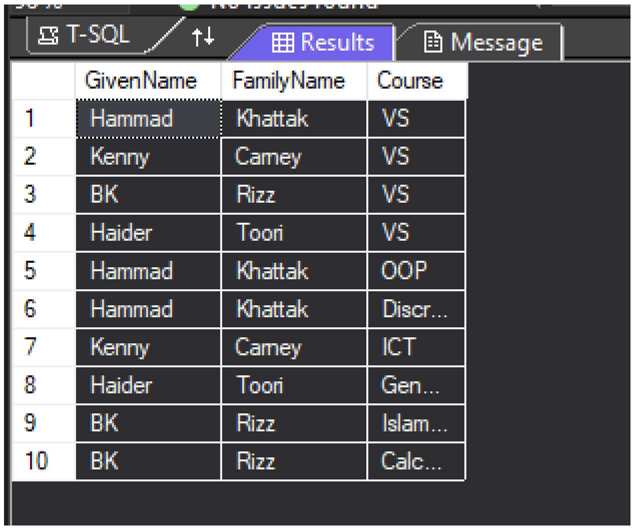


When we get to the coding section, we'll grab this Stored Procedure using its name, **NamesAndCourses**.

You can run this Stored Procedure. Right click it and select Execute from the menu:



You should the results of the query displayed in the bottom window and some SQL at the top. The table of results is what we're after. This is what we'll get when execute our query with code later.



	GivenName	FamilyName	Course
1	Hammad	Khattak	VS
2	Kenny	Carney	VS
3	BK	Rizz	VS
4	Haider	Toori	VS
5	Hammad	Khattak	OOP
6	Hammad	Khattak	Discr...
7	Kenny	Carney	ICT
8	Haider	Toori	Gen...
9	BK	Rizz	Islam...
10	BK	Rizz	Calc...

Now that we have a Stored Procedure that gets all the results, let's create one to limit the results we get back. This time, we'll use those one of the @param values you deleted earlier. We'll do that in the next lesson below.

Using Sql Parameters in Stored Procedures

Create a new Stored Procedure, just like you did the last time. Change the name to **GetStudentByID**:

```
CREATE PROCEDURE [dbo].[GetStudentByID]
```

What we want to do in our C# code is to have a user type a Student ID in a text box and then search the database for a student that matches the ID. We can use the same SELECT statement we had before:

```
SELECT Students.GivenName, Students.FamilyName,
Course.Course
FROM Students
JOIN Course
ON Students.StudentID = Course.StudentID
```

You've met WHERE clauses in a previous lesson. We can add one here, as well. If you were hard-coding the WHERE clause, it would look something like this:

```
WHERE Students.StudentID = 10
```

You can replace that hard-coded 10 with a parameter. By default, SQL Server gives you two parameters:

```
@param1 int = 0,
@param2 int
```

Parameters are placeholders for values. In our example, you can replace to hard-coded 10 with your parameter:

```
WHERE Students.StudentID = @param1
```

You'll see in a moment how to execute a statement in SQL Server. But you pass a value over and it ends up in the placeholder, @param1. This will happen in our C# code, as well: whatever student id number a user types into a text box will end up in @param1.

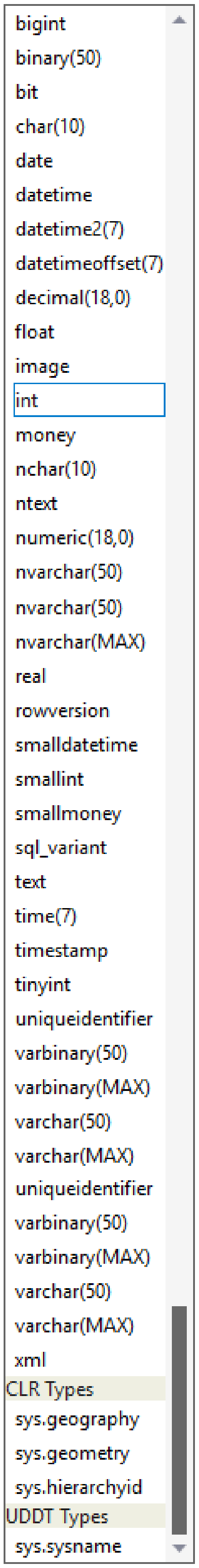
Except, param1 is just a default name for the parameter. You can have almost anything you like here. Change param1 to something more appropriate. Call it GetStudentID. And you can delete the second parameter as we'll have just the one. If you wanted to, you could use a second placeholder. For example, suppose you wanted to search on the StudentID and the FamilyName. You could add an AND part to the WHERE clause:

```
WHERE Students.StudentID = @param1 AND FamilyName
= 'Carney'
```

But delete the second parameter and the top of your Stored Procedure will look like this: (Don't worry about the underlines):

```
CREATE PROCEDURE [dbo].[GetStudentByID]

    @GetStudentID int = 0

AS
```

The int after the parameter is a type, and it's just like setting up a variable in C# - you need to specify the type of value going into your parameter. But the types here are the same ones from the dropdown list you used when setting up your table, the ones with NVARCHAR. These ones:


bigint
binary(50)
bit
char(10)
date
datetime
datetime2(7)
datetimeoffset(7)
decimal(18,0)
float
image
int
money
nchar(10)
ntext
numeric(18,0)
nvarchar(50)
nvarchar(50)
nvarchar(MAX)
real
rowversion
smalldatetime
smallint
smallmoney
sql_variant
text
time(7)
timestamp
tinyint
uniqueidentifier
varbinary(50)
varbinary(MAX)
varchar(50)
varchar(MAX)
uniqueidentifier
varbinary(50)
varbinary(MAX)
varchar(50)
varchar(MAX)
xml
CLR Types
sys.geography
sys.geometry
sys.hierarchyid
UDDT Types
sys.sysname

We set up the StudentID column to be an int, so the parameter needs to be an int as well. Had we set it up as NVARVCHAR our parameter type would be this:

```
@GetStudentID NVARCHAR(50)
```

The = 0 on the end means you're setting a default value for your parameter.

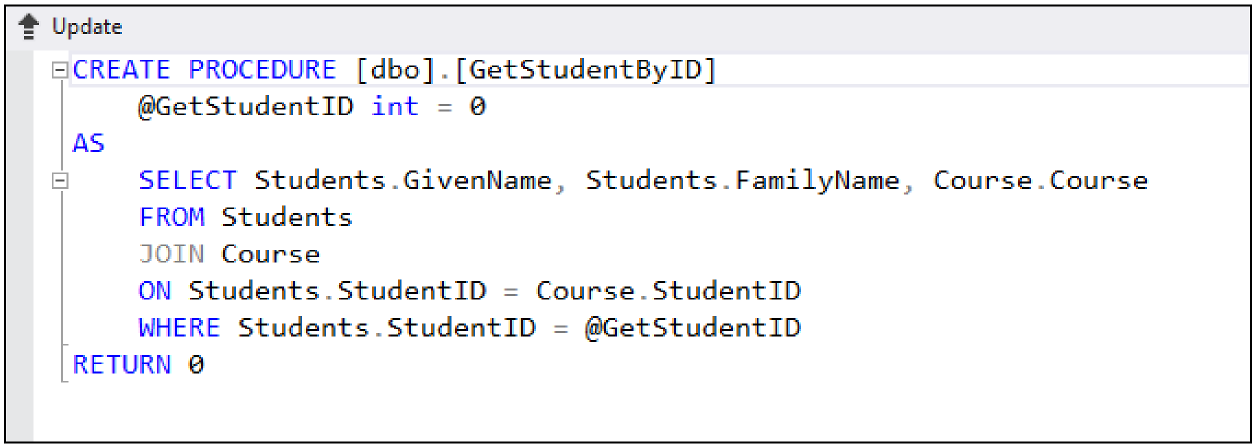
You can paste in your SELECT statement, the one you had before. This one:

```
SELECT Students.GivenName, Students.FamilyName,
Course.Course
FROM Students
JOIN Course
ON Students.StudentID = Course.StudentID
```

Paste it in after the AS keyword and before the RETURN 0 at the end. Add the WHERE clause, as well:

```
SELECT Students.GivenName, Students.FamilyName,
Course.Course
FROM Students
JOIN Course
ON Students.StudentID = Course.StudentID
WHERE Students.StudentID = @GetStudentID
```

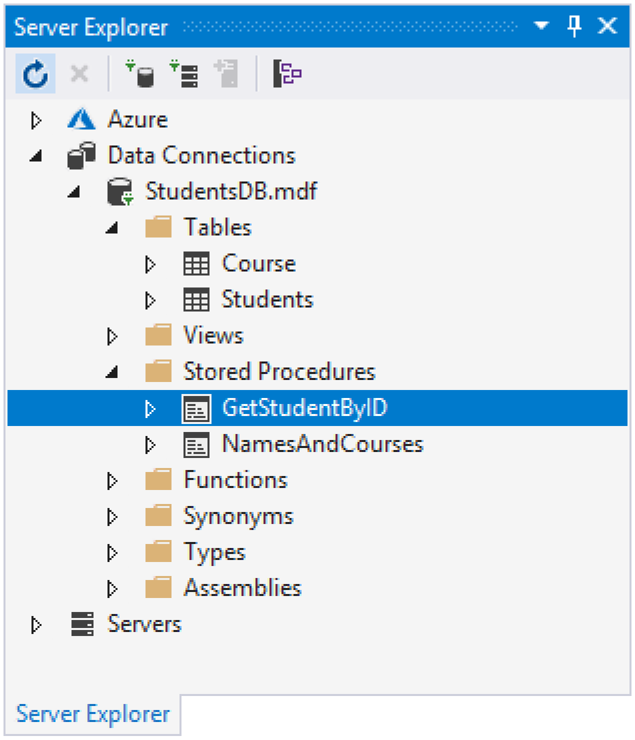
Your Stored Procedure window should look like this:



Update

```
CREATE PROCEDURE [dbo].[GetStudentByID]
    @GetStudentID int = 0
AS
    SELECT Students.GivenName, Students.FamilyName, Course.Course
    FROM Students
    JOIN Course
    ON Students.StudentID = Course.StudentID
    WHERE Students.StudentID = @GetStudentID
RETURN 0
```

Click the Update button in the top left when you're done. In the Server Explorer area on the left, click the Refresh button again. You should see your second Stored Procedure in the folder:



Let's test it out.

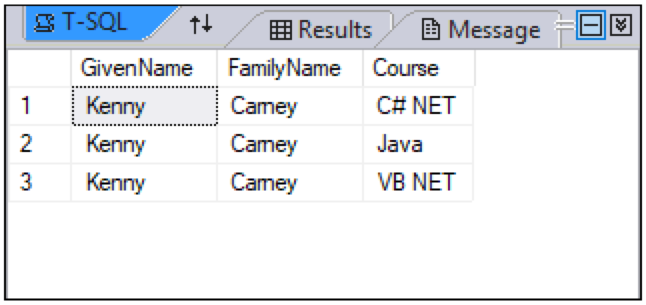
Right-click on the Tables folder. Select New Query from the menu. To execute a Stored Procedure, you ned this format:

EXEC [Stored_Procedure_Name] [Any_Parameters here]

Type this in the Query window:

EXEC dbo.GetStudentByID 10

Click the Green button to run the query. You should see three results displayed:



	GivenName	FamilyName	Course
1	Kenny	Carney	C# NET
2	Kenny	Carney	Java
3	Kenny	Carney	VB NET

Change the 10 into 11 and run the query again. You should see the results for Bob Job appear. Try 12 to see Mona Lisa's result.

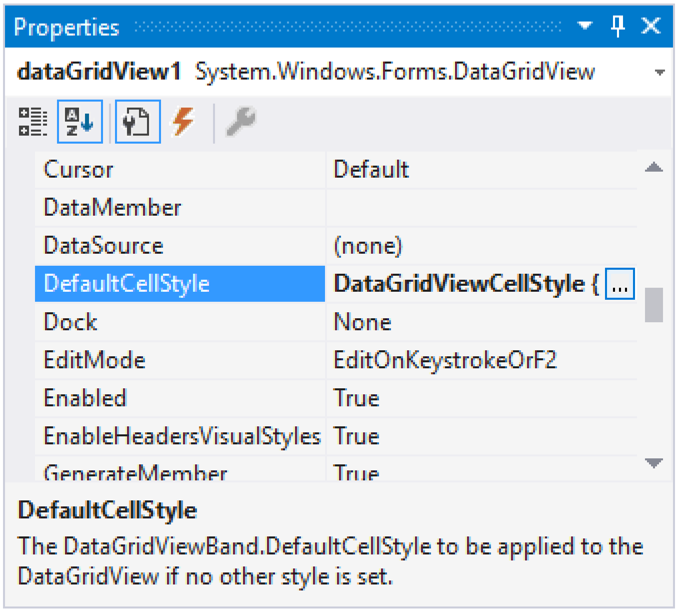
OK, now that we have a database, two tables, and some Stored Procedures, we can do the C# coding.

Designing the User Form with a Data Grid

What we'll do with our Windows Form is to add a couple of buttons, a text box, and a data grid. We'll use the first button to pull all the records from the tables and place them in the data gird. For the second button, we'll allow a user to type a student number into the text box. We'll get that number and display the records from a single student. We'll use our Stored Procedures in the code.

Add a button to your form. Change the name to something appropriate, like **BtnGetAll**. Change the Text property, as well. In the **Data** section of the Visual Studio Toolbox, drag and drop a **DataGridView** control onto your form.

DataGridViews have lots of properties you can change. One you might want to change is the font size. Scroll down and locate the **DefaultCellStyle** property:



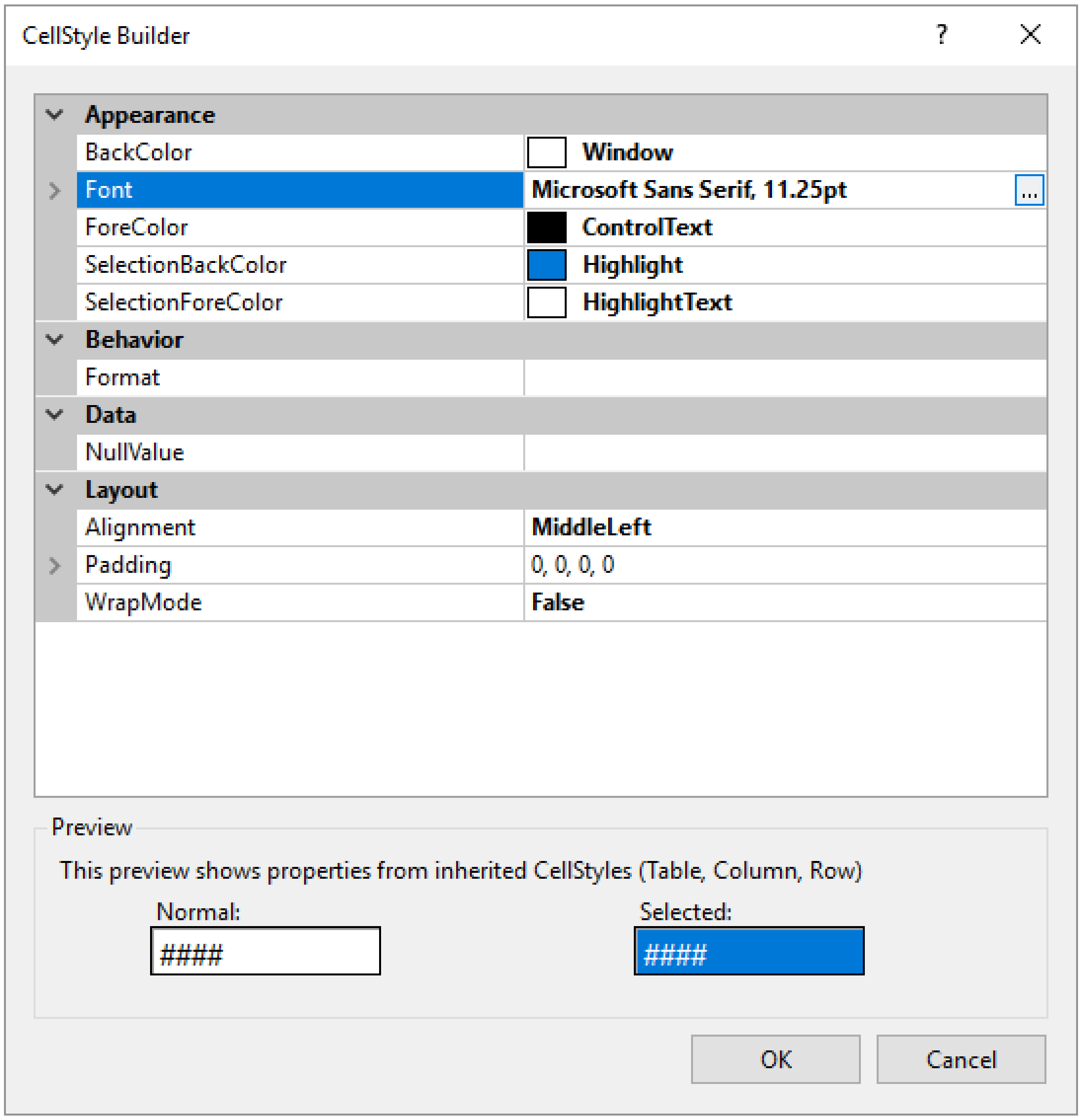
dataGridView1 System.Windows.Forms.DataGridView

Cursor	Default
DataMember	
DataSource	(none)
DefaultCellStyle	DataGridViewCellStyle {
Dock	None
EditMode	EditOnKeystrokeOrF2
Enabled	True
EnableHeadersVisualStyles	True
GenerateMember	True

DefaultCellStyle

The DataGridViewBand.DefaultCellStyle to be applied to the DataGridView if no other style is set.

Click the button with the three dots in it. You'll see this dialog box appear:



Appearance	
BackColor	⬜ **Window**
Font	**Microsoft Sans Serif, 11.25pt**
ForeColor	⬛ **ControlText**
SelectionBackColor	🟦 **Highlight**
SelectionForeColor	⬜ **HighlightText**
Behavior	
Format	
Data	
NullValue	
Layout	
Alignment	**MiddleLeft**
Padding	0, 0, 0, 0
WrapMode	**False**

Preview

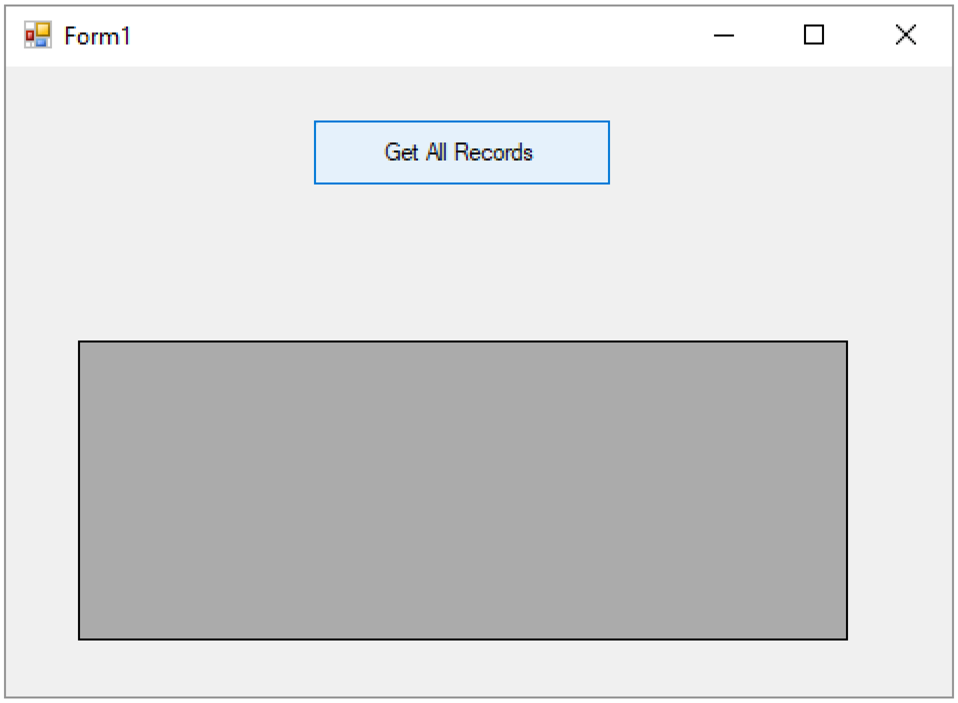
This preview shows properties from inherited CellStyles (Table, Column, Row)

Normal: ####

Selected: ####

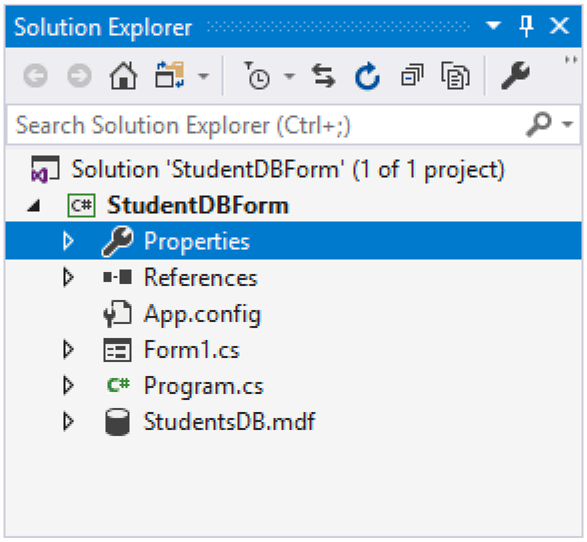
Change the Font to anything you like. The bigger sizes don't look too good, however.

Your form will look something like this:

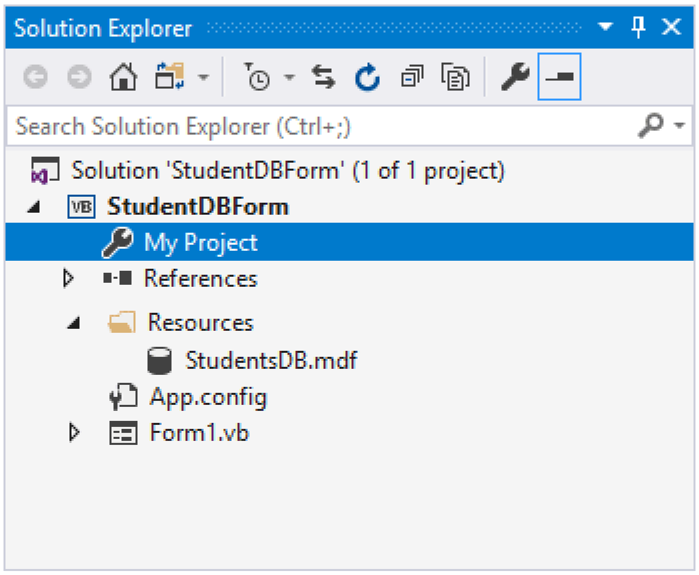


Before getting to the code, we can set a Property for the Data Source of our database. That way, we won't have to add a long and complex string.

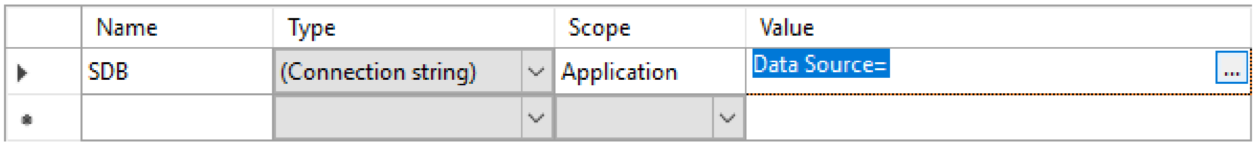
In the Solution Explorer on the right, C# users should double-click the **Properties** item:



VB NET users should double-click the **My Project** item:

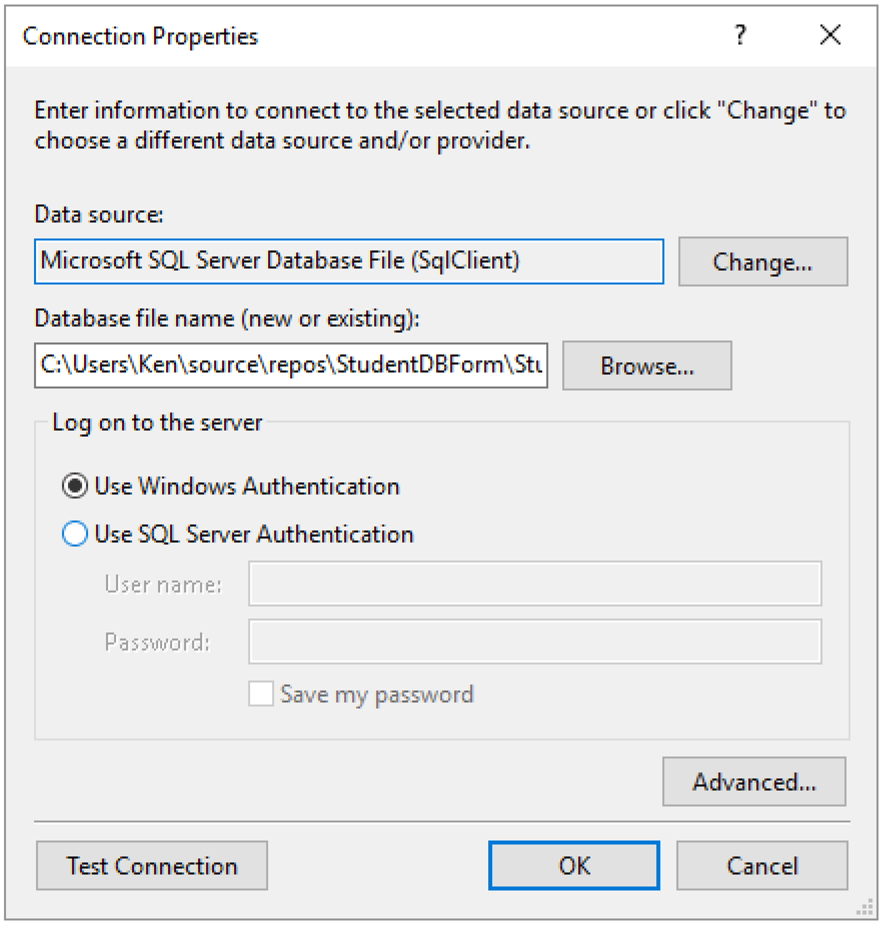


Click on the Settings tab and set the **Name** to anything you like, something like SDB will do, for Student Database. Set the **Type** to Connection String and the **Scope** to **Application**. Click inside the Value box and the click the button on the right:



	Name	Type	Scope	Value
▶	SDB	(Connection string)	Application	Data Source=
*				

When you click the button, you'll see this dialog box:


Connection Properties

Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

Data source: Microsoft SQL Server Database File (SqlClient)

Database file name (new or existing): C:\Users\Ken\source\repos\StudentDBForm\Stu

Log on to the server

(●) Use Windows Authentication

( ) Use SQL Server Authentication

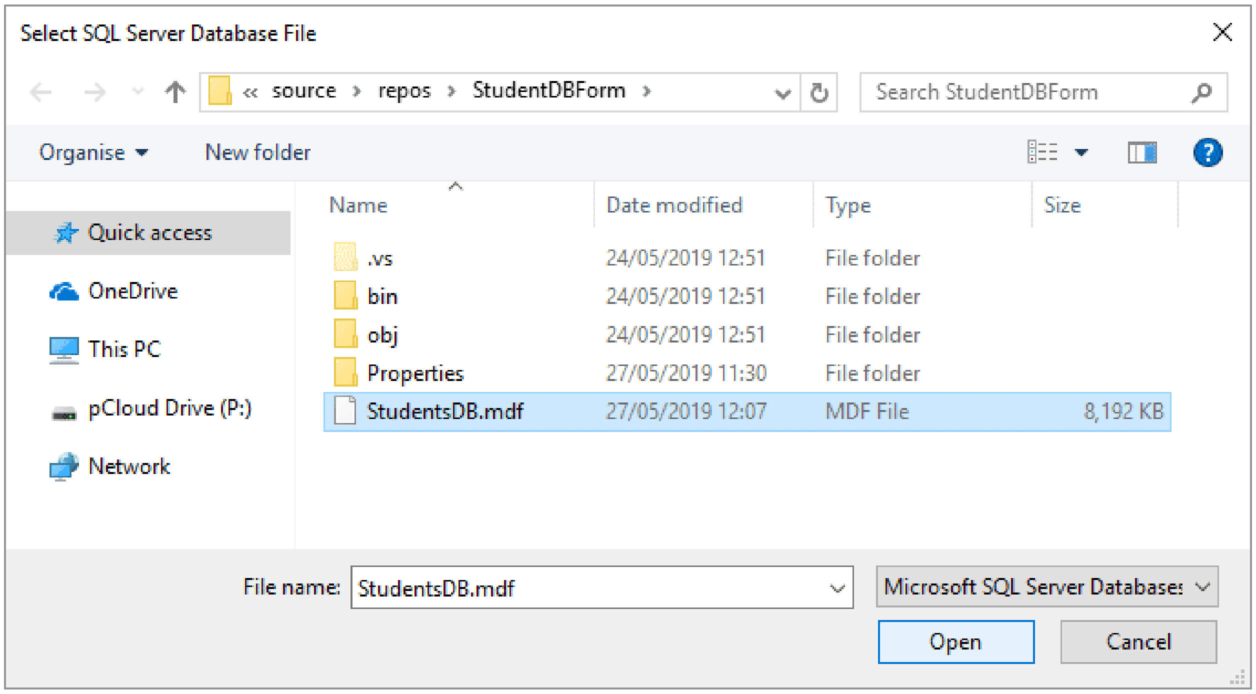
User name:

Password:

☐ Save my password

Change the Data source to Microsoft SQL Server Database File (SqlClient).

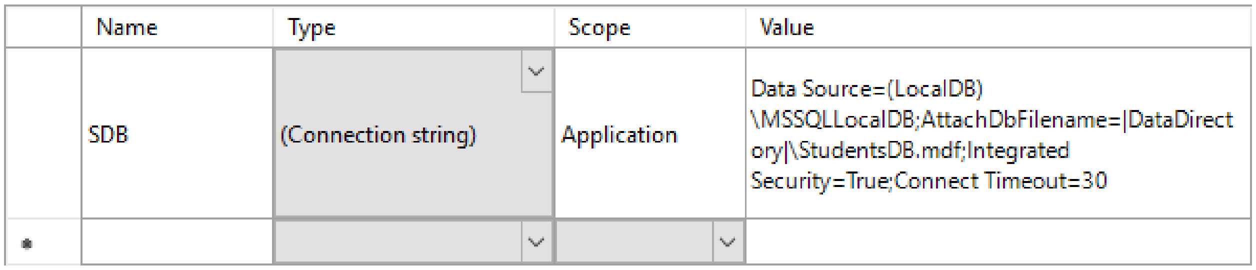
Click the Browse button and locate your database. It should be in your project folder. The image below shows ours, with the StudentsDB file selected:



Name	Date modified	Type	Size
.vs	24/05/2019 12:51	File folder	
bin	24/05/2019 12:51	File folder	
obj	24/05/2019 12:51	File folder	
Properties	27/05/2019 11:30	File folder	
StudentsDB.mdf	27/05/2019 12:07	MDF File	8,192 KB

File name: StudentsDB.mdf
Microsoft SQL Server Databases

You can test your connection to make sure everything's working. Then click OK. The Value column in your settings will then be like this:



	Name	Type	Scope	Value
	SDB	(Connection string)	Application	Data Source=(LocalDB)\MSSQLLocalDB;AttachDbFilename=|DataDirectory|\StudentsDB.mdf;Integrated Security=True;Connect Timeout=30
*				

You can close the Properties tab now. In the next lesson, we'll start the coding

Writing the code for the Stored Procedure

Now that you have a form and a connection object set up, double-click your button to open up a code stub.

We're going to need something called a Binding Source. This is used to bind the Data Grid to the Data Table we'll create shortly. Add the following line to you code, just above the button code stub but inside of the class:

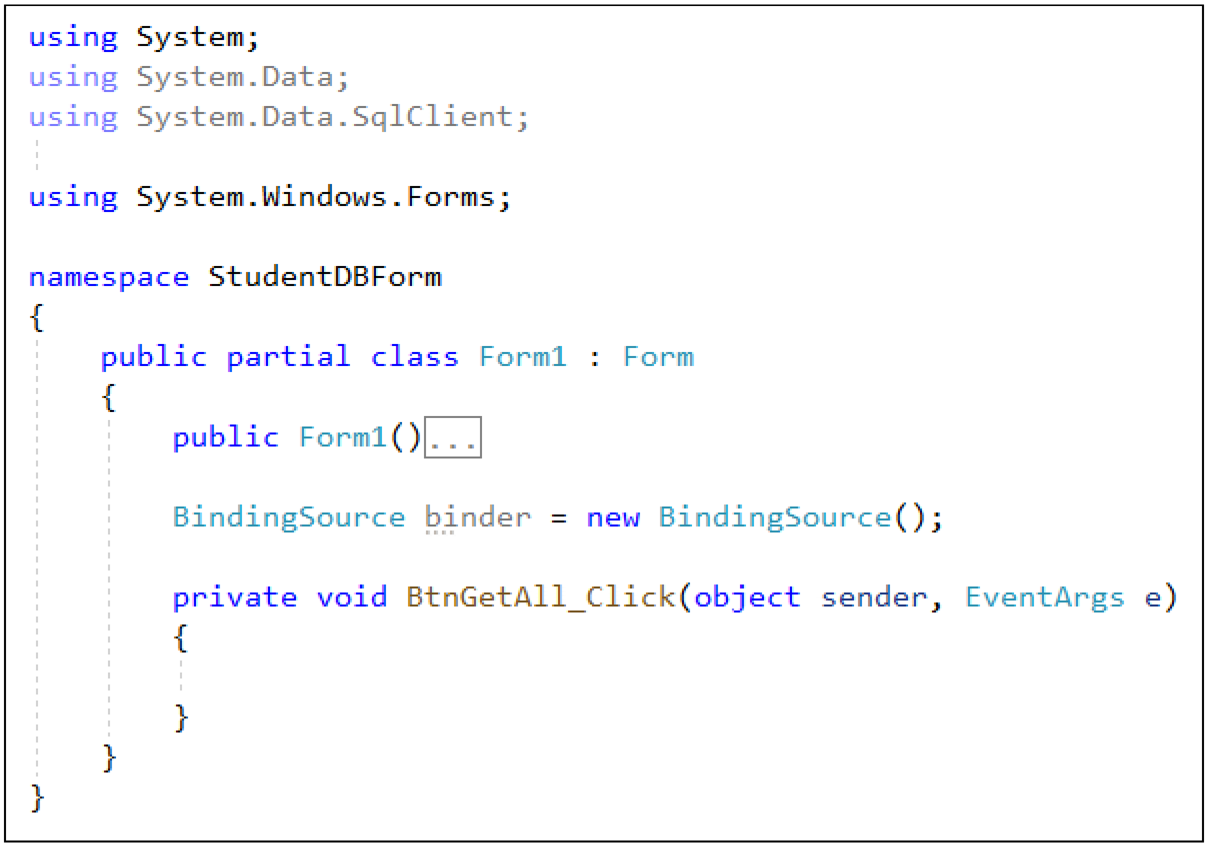
C#

BindingSource binder = new BindingSource();

For C# users, at the top of the code, add the following using statement:

using System.Data.SqlClient;

You code should look like this:



```
using System;
using System.Data;
using System.Data.SqlClient;

using System.Windows.Forms;

namespace StudentDBForm
{
    public partial class Form1 : Form
    {
        public Form1()...

        BindingSource binder = new BindingSource();

        private void BtnGetAll_Click(object sender, EventArgs e)
        {

        }
    }
}
```

We can set up a connection string and get that property we set up. Add these two lines inside your button code stub:

C#

```
string conString;
conString = Properties.Settings.Default.SDB;
```

The SDB part was the one we set up. If you typed something else as your connection name, select that instead.

We now need a SqlConnection object so we can open up a connection:

C#

```
SqlConnection con = new SqlConnection(conString);
```

In between the round brackets of you SqlConnection object, you need your connection string. Open a connection with this line:

C#

```
con.Open();
```

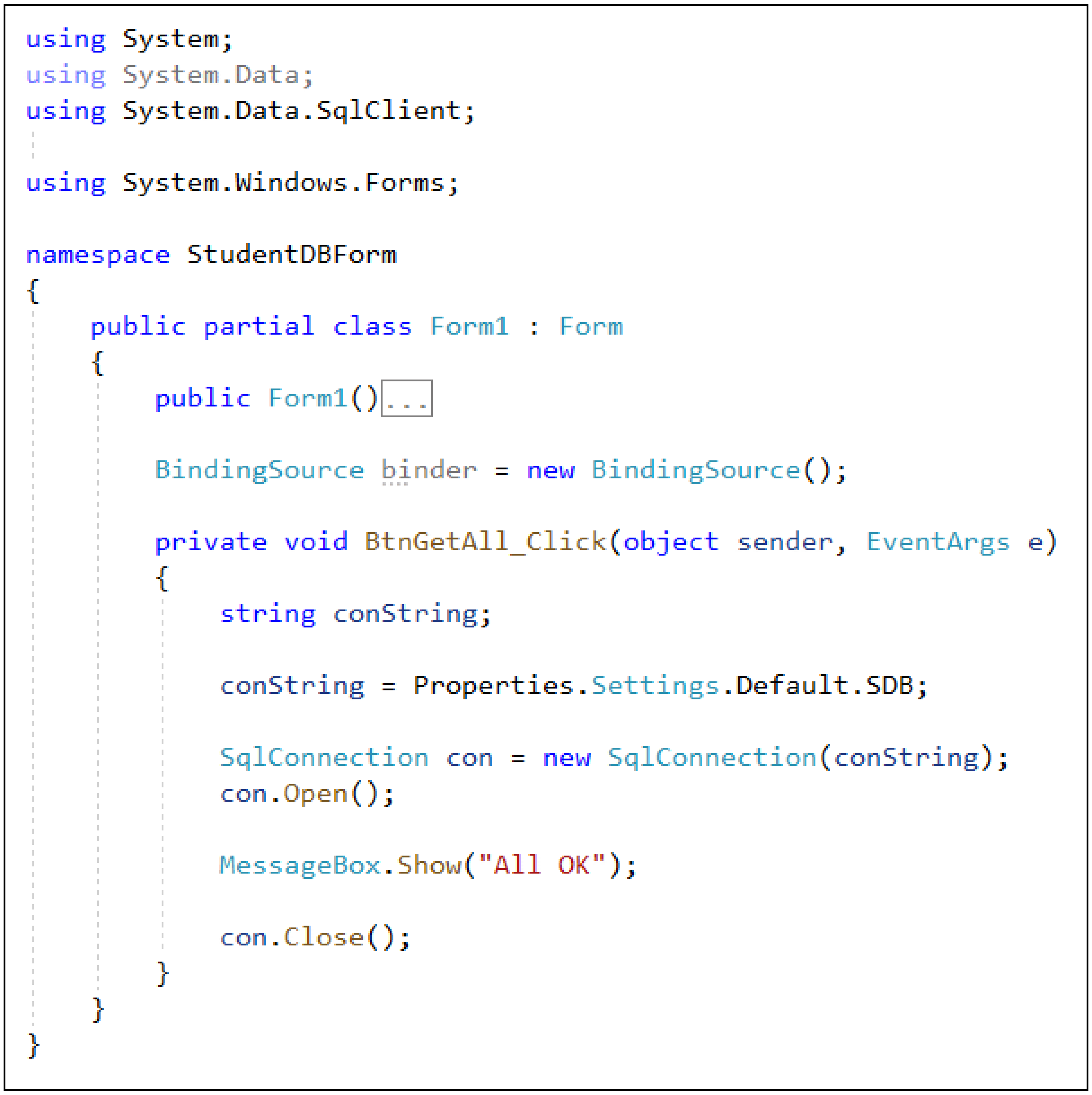
To check if everything is working correctly, you can add a message box (without the semicolon on the end, for VB users):

```
MessageBox.Show("All OK");
```

Then close the connection with: (Again without the semicolon on the end, for VB users.)

```
con.Close();
```

C# .NET users, your code should look like this:



```
using System;
using System.Data;
using System.Data.SqlClient;

using System.Windows.Forms;

namespace StudentDBForm
{
    public partial class Form1 : Form
    {
        public Form1()...

        BindingSource binder = new BindingSource();

        private void BtnGetAll_Click(object sender, EventArgs e)
        {
            string conString;

            conString = Properties.Settings.Default.SDB;

            SqlConnection con = new SqlConnection(conString);
            con.Open();

            MessageBox.Show("All OK");

            con.Close();
        }
    }
}
```

Next, we can create a SqlCommand object and grab one of our Stored Procedures.

After con.Open, add this line: (you can get rid of your message box, if you want, or comment it out.)

C#

```
SqlCommand cmd = new
SqlCommand("NamesAndCourses", con);
```

In between the round brackets of the new SqlCommand we have two things. The first is the name of a Stored Procedure in double quotes. After a comma, we then have the connection object.

You can add parameter information to the command object, and we'll do that with the second button. But our first Stored Procedure didn't have any, so the single line is enough.

We now need a Data Adapter. This is an object that sits between the database and the data table we'll add. So here's your next line of code:

C#

```
SqlDataAdapter adapter = new SqlDataAdapter(cmd);
```

Notice that in between the round brackets of the SqlDataAdapter is the command object we set up, the one with the Stored Procedure and the Connection object.

We need a Data Table. This table will be filled with the result we get back from the Stored Procedure. (Because we have a Data Grid we can't use a DataSet, like last time.) To set up a Data Table, add this line:

C#

```
DataTable dataTable = new DataTable();
```

You use the adapter to fill the table with the results. Here's the line: (VB Net usrs, delete the semicolon at the end):

C# and VB Net

```
adapter.Fill(dataTable);
```

The Data Grid has a DataSource property. You can set this to the Binding Source we set up at the top of the code:

C#

```
dataGridView1.DataSource = binder;
```

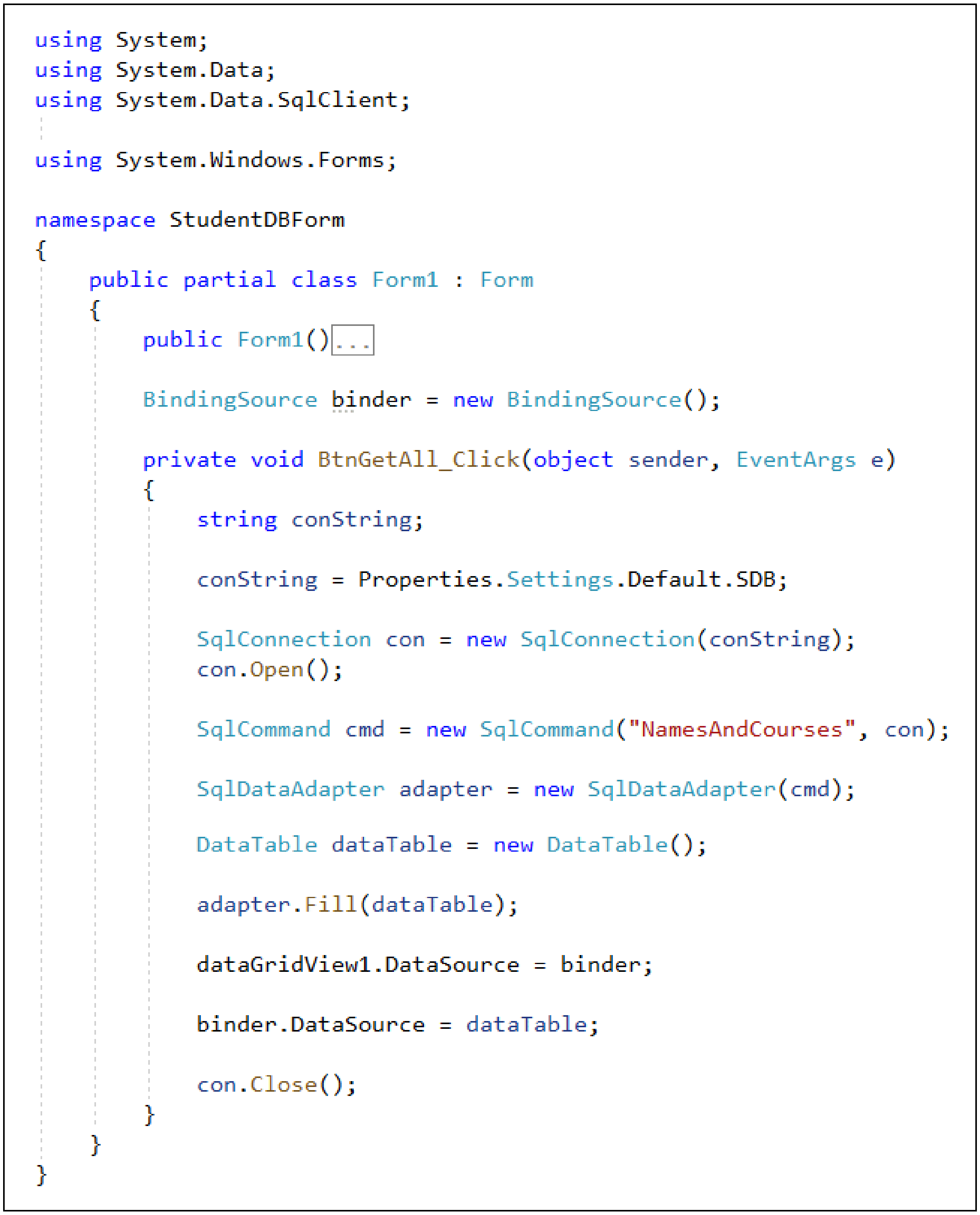
The Binding Source object also has a DataSource property. We can point our Data Table at this. The Data Table, remember, now has all the results from the database. Add this: (Again, no semicolon on the end, if you're coding in Visual Basic.)

```
binder.DataSource = dataTable;
```

The final line is to close the connection:

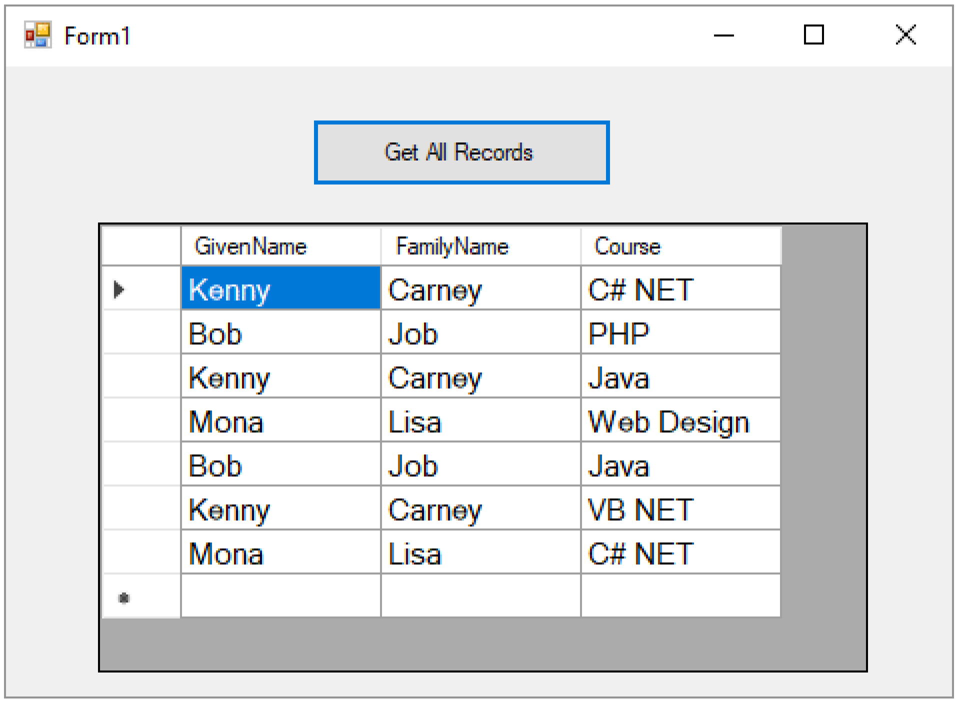
```
con.Close();
```

Your code should look like this in C#:



```
using System;
using System.Data;
using System.Data.SqlClient;

using System.Windows.Forms;

namespace StudentDBForm
{
    public partial class Form1 : Form
    {
        public Form1()...

        BindingSource binder = new BindingSource();

        private void BtnGetAll_Click(object sender, EventArgs e)
        {
            string conString;

            conString = Properties.Settings.Default.SDB;

            SqlConnection con = new SqlConnection(conString);
            con.Open();

            SqlCommand cmd = new SqlCommand("NamesAndCourses", con);

            SqlDataAdapter adapter = new SqlDataAdapter(cmd);

            DataTable dataTable = new DataTable();

            adapter.Fill(dataTable);

            dataGridView1.DataSource = binder;

            binder.DataSource = dataTable;

            con.Close();
        }
    }
}
```

You can run your code, now. Click your button and you should see all the results appear in your Data Grid:

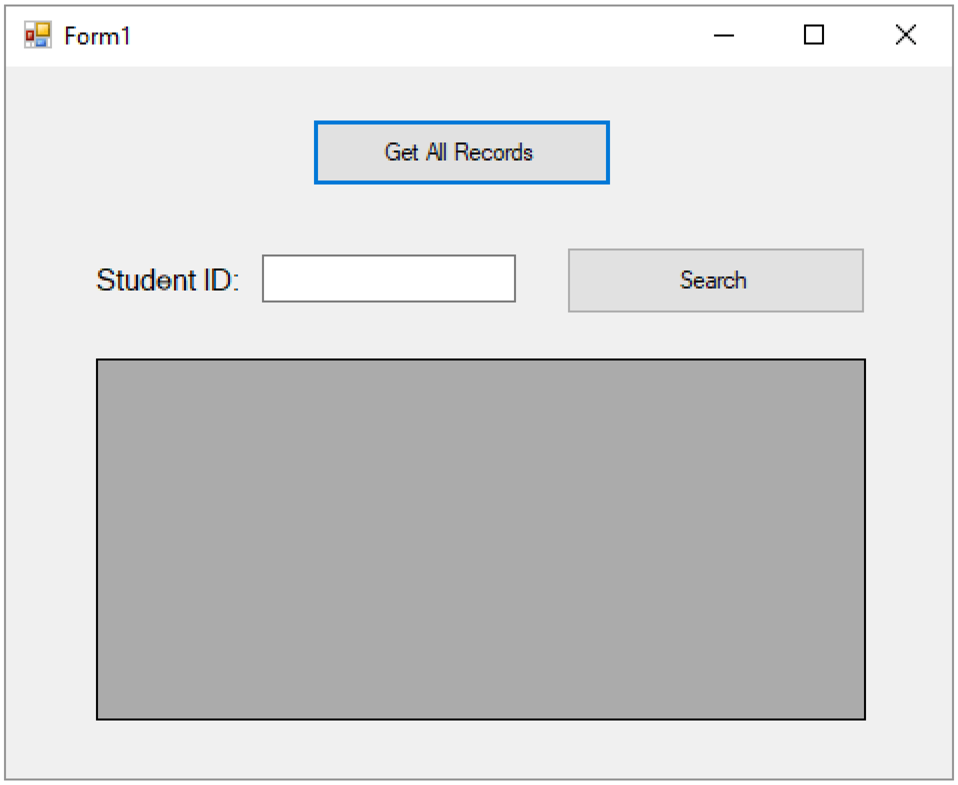


	GivenName	FamilyName	Course
▶	Kenny	Carney	C# NET
	Bob	Job	PHP
	Kenny	Carney	Java
	Mona	Lisa	Web Design
	Bob	Job	Java
	Kenny	Carney	VB NET
	Mona	Lisa	C# NET
*			

We have successfully grabbed data from two database tables and displayed the results in a DataGrid. Now let's use our other Stored Procedure so that we can search for a record based on the StudentID.

Stored Procedures and Parameters with C#

Add a text box and a button to the form you have already created. You can add a label, as well, so that your form looks something like this:



What we're going to do here is to enter a Student ID in the text box. When the button is clicked, the Student's details will appear in the DataGrid. We'll use our Stored Procedure for this, the one we set up with Parameters.

Double-click your button to open up the code stub.

For convenience sake, we'll just copy and paste the code from the first button, and then make a few additions. Obviously, it would be better to set up a few functions, or maybe a class, to cut down on duplicate code. But it's nice to compare the two versions, and see how they differ.

Highlight all you code from your first button. Copy and paste it into the code stub for the second button.

The first change we can make is to the name of the Stored Procedure. Change it from **NamesAndCourses** to the name of our second Stored Procedure, which was **GetStudentByID**:

C#

```
SqlCommand cmd = new
SqlCommand("GetStudentByID", con);
```

The SqlCommand object has a Propertry called **CommandType**. You can set this to Stored Procedure. Add this line just after your **cmd** line:

C# and VB Net (without the semicolon)

```
cmd.CommandType = CommandType.StoredProcedure;
```

Another Property of the SqlCommand object is **Parameters**. This will allow us to **Add** that GetStudentID parameter we set up. Add this line just below the new one:

C#

```
cmd.Parameters.Add(new
SqlParameter("@GetStudentID", 10));
```

The Parameters Property has a method called **Add**. In between the round brackets of Add, you need a new **SqlParameter** object. In between the round brackets of SqlParameter you can add your parameters and any values for them:

```
"@GetStudentID", 10
```

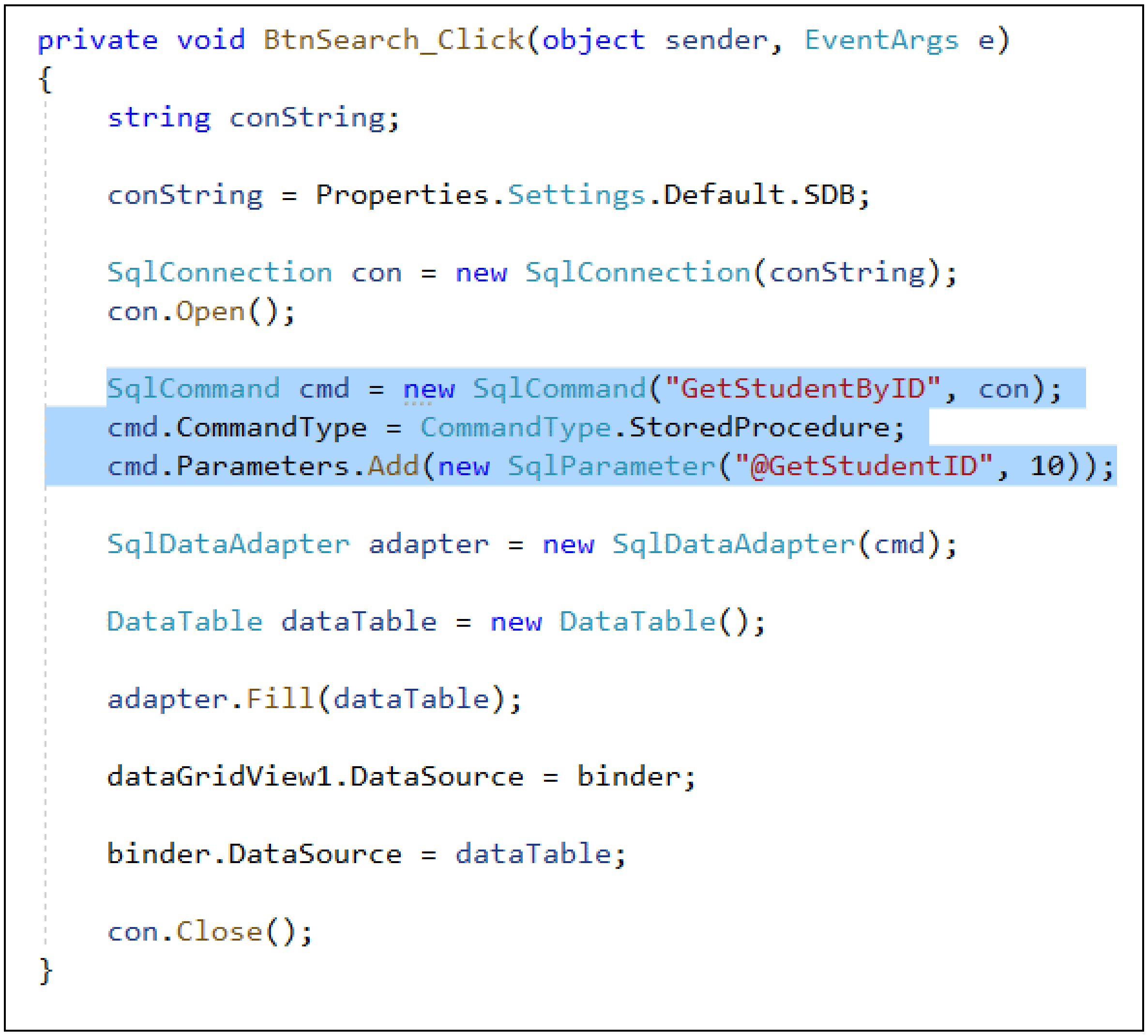
The parameter goes between double quotes, and with the @ sign at the start. After a comma, you type a value for the parameter. Remember how we did it in SQLServer:

```
EXEC dbo.GetStudentByID 10
```

A value of 10 is passed to the parameter in the **GetStudentByID** Stored Procedure. By saying, SqlParameter("@GetStudentID", 10), you're doing the same thing - handing a value to the parameter.

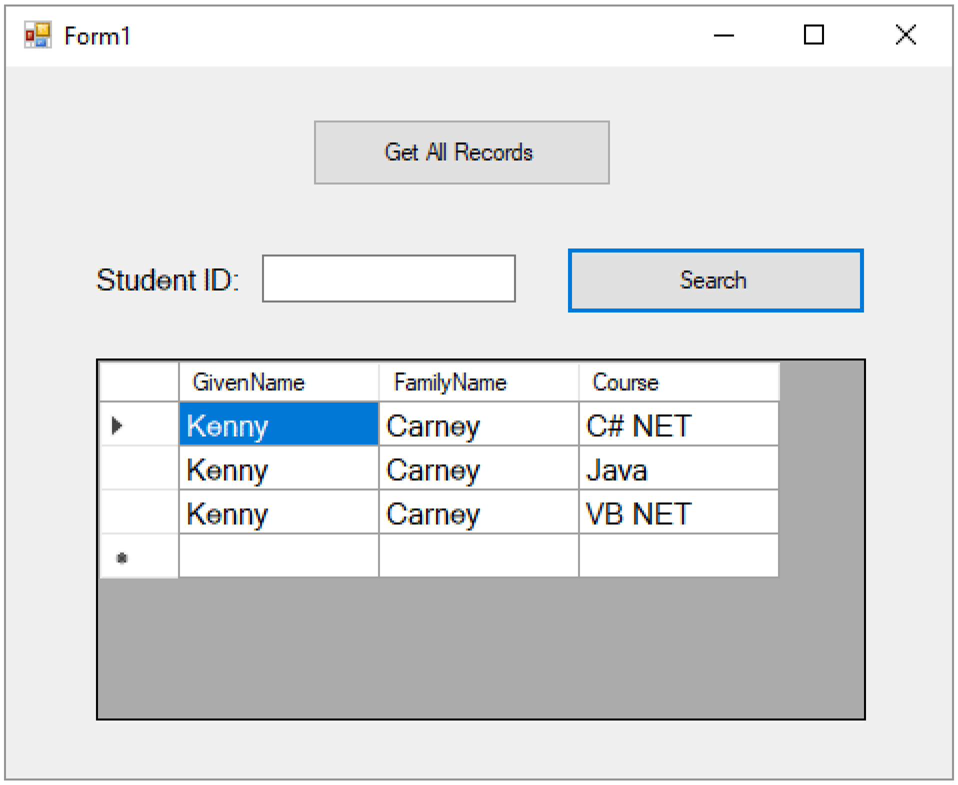
However, we don't want to hard-code a value. We'll replace that soon with a value from the text box on the form. But here's what the code for your new button should look like so far, with the changes and additions highlighted:

C#



```
private void BtnSearch_Click(object sender, EventArgs e)
{
    string conString;

    conString = Properties.Settings.Default.SDB;

    SqlConnection con = new SqlConnection(conString);
    con.Open();

    SqlCommand cmd = new SqlCommand("GetStudentByID", con);
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.Add(new SqlParameter("@GetStudentID", 10));

    SqlDataAdapter adapter = new SqlDataAdapter(cmd);

    DataTable dataTable = new DataTable();

    adapter.Fill(dataTable);

    dataGridView1.DataSource = binder;

    binder.DataSource = dataTable;

    con.Close();
}
```

Try it out with the hard-coded value of 10. You should see this:



	GivenName	FamilyName	Course
▶	Kenny	Carney	C# NET
	Kenny	Carney	Java
	Kenny	Carney	VB NET
*			

The only thing left to do is to get the text box working.

Add the following to the top of your button code:

C#

```
int studentIdValue = int.Parse(textBox1.Text);
```

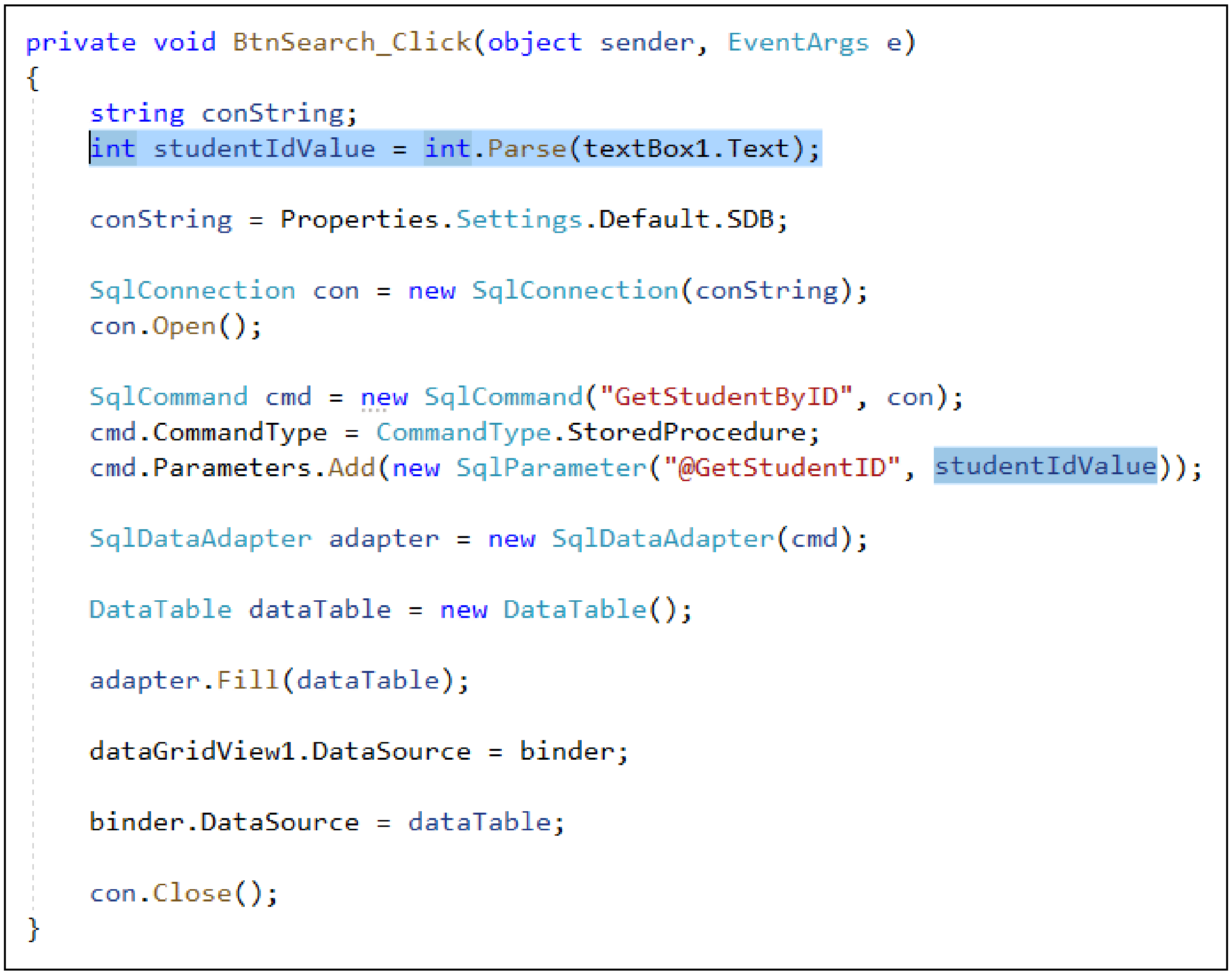
This sets up an integer called **studentIdValue**. But we need to convert the string from the text box to an integer. You can convert a string to an integer in C# with int.Parse(). You'd probably need to do some error checking as well, here, just to make sure it is a valid integer. But we won't bother. In VB Net, you can just use Val().

Now delete the hard-coded 10 from the SqlParameter line and replace it with your int variable:

C#

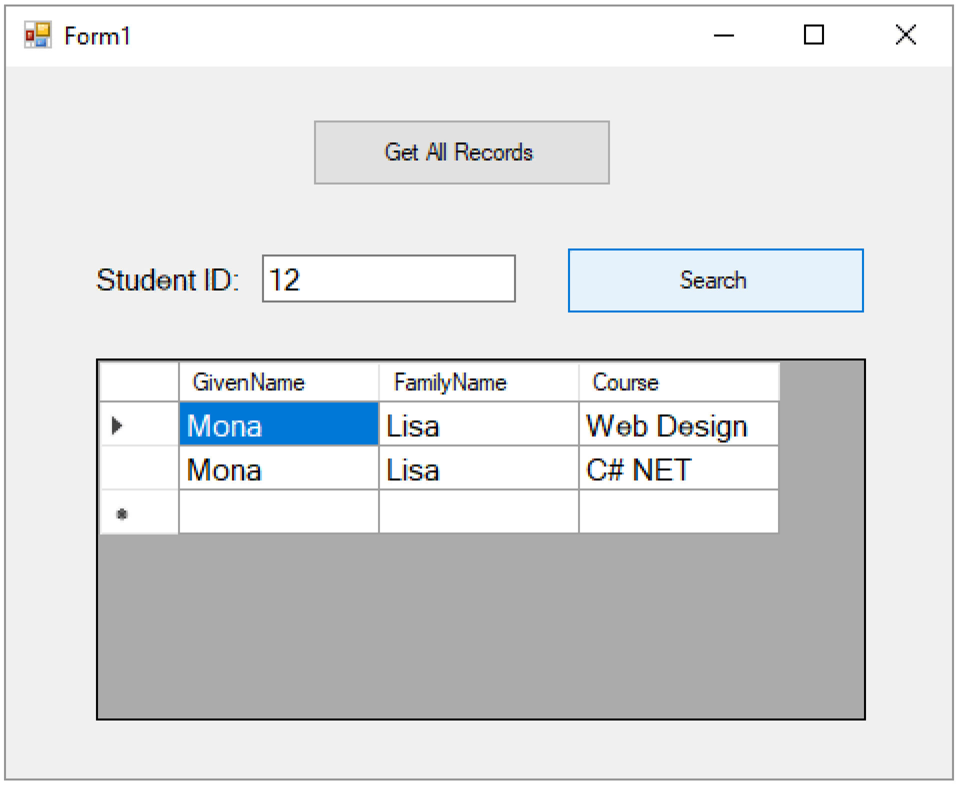
```
cmd.Parameters.Add(new
SqlParameter("@GetStudentID", studentIdValue));
```

Here's what you code should look like in C#, with the new additions highlighted:



```
private void BtnSearch_Click(object sender, EventArgs e)
{
    string conString;
    int studentIdValue = int.Parse(textBox1.Text);

    conString = Properties.Settings.Default.SDB;

    SqlConnection con = new SqlConnection(conString);
    con.Open();

    SqlCommand cmd = new SqlCommand("GetStudentByID", con);
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.Add(new SqlParameter("@GetStudentID", studentIdValue));

    SqlDataAdapter adapter = new SqlDataAdapter(cmd);

    DataTable dataTable = new DataTable();

    adapter.Fill(dataTable);

    dataGridView1.DataSource = binder;

    binder.DataSource = dataTable;

    con.Close();
}
```

Run your form again. Enter a value in the text box and click your Search button. Here's what you should see if a value of 12 is entered:



	GivenName	FamilyName	Course
▶	Mona	Lisa	Web Design
	Mona	Lisa	C# NET
*			

Try out the other numbers we set up as Student IDs, 10 and 11. Try a different number, one we didn't add, and see what happens. Does it crash?

So that's it. That's how to open multiple database tables with C# and VB Net code. Just remember: Stored Procedures can make your life much easier. In the next lesson, you'll see how to create a Stored Procedure to update a table. We'll then write code to execute it.

Create a Stored Procedure to Update a Table

What we'll do now is to add an update button. When the button is clicked, it will update the Students table. (It doesn't make sense to update both our tables because the Courses table is the Many table in a One-to-Many relationship. We'll just be updating a Student's name, which is in the Students table.)

We'll need two more Stored Procedures. One is a simple on that grabs all the records from Students. The second one will do the updating.

In the Solution Explorer on the left, create a new Stored Procedure just like you did before. Change default file to this:

```
CREATE PROCEDURE [dbo].[AllStudents]

AS

    SELECT * FROM Students

RETURN 0
```

This is a simple SELECT statement that gets all the records from the Students table.

Click the Update button. In the Solution Explorer, Refresh to see your new Stored Procedure in the folder. (You might need to refresh a few times, as it's sometimes slow to appear.) You can right-click Stored Procedure and select Execute, if you want to test it out.

The second Stored Procedure is a more difficult one. Here it is for you to create (you don't have to do all the indenting, but it does make it more readable):

```
CREATE PROCEDURE [dbo].[UpdateStudentsTable]

    (@sID int = 0,
    @Fam nvarchar(50),
    @Giv nvarchar(50))

AS

    BEGIN

        UPDATE Students

            SET FamilyName = @Fam,
                    GivenName = @Giv
                    WHERE StudentID = @sID

          END

     RETURN 0
```

The name of the Stored Procedure is UpdateStudentsTable. It has three parameters, @sID, @Fam, and @Giv. The last two are set up as nvchar(50) as that's what their Data Types were set up as when we created this table.

After the AS keyword, we have a BEGIN and END block. The update code goes inside the two. It starts with the keyword UPDATE. After a space, you need the name of a table. You SET values for columns in your table. We have a column called FamilyName and this is being assigned the @Fam parameter. We have another column called GivenName and this gets the @Giv parameter. But we only want to update these two columns WHERE we have a match for the StudentID.

Click the update button in the top left of your Stored Procedure.

To test it out, create a new Sql Query. (Right-click the Tables item in the Server Explorer then select New Query from the menu.) Try this as the query:

```
EXEC UpdateStudentsTable 10, "Carn", "Ken"
```

Click the green button to run the query. To see the results, right-click your AllStudents query and select Execute from the menu. You should see that the student called Kenny Carney has been changed to Ken Carn.

Change it back with this:

```
EXEC UpdateTest 10, "Carney", "Kenny"
```

Incidentally, you can update more than one table at a time. Just add another UPDATE:

```
BEGIN

        UPDATE Students

        SET FamilyName = @Fam,
        GivenName = @Giv

        WHERE StudentID = @sID

    UPDATE Another_Table

        SET ColumnA = @param4,
        ColumnB = @param5
        WHERE ColumnC = @param6

END
```

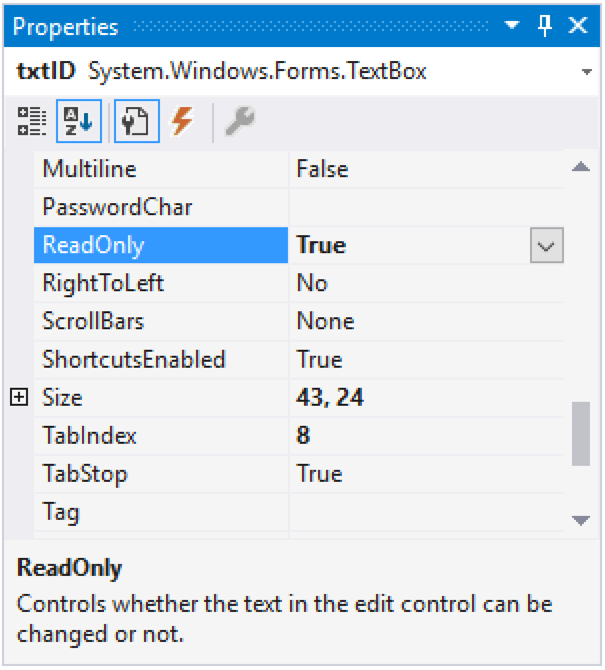
OK, now we have our Stored Procedures we can write some code.

Coding for the Update

We'll add some new buttons and a few text boxes to our form. Add a button and change the name to **BtnGetStudentsTable**. Set the Text property to **Get Students Table**. When this button is clicked, all the record from just the Students table will appear in the Data Grid.

Editing the cells in a Data Grid can be a little fiddly. What we'll do is to have a row from the Data Grid appear in text boxes whenever you click in the grid. That way, they will be easier to change.

Add three text boxes to your form. The first is for the ID. Change the name to txtID. We don't want anyone changing the ID, so you can make the text box read only. With the txtID text box selected, scroll down the Properties and locate the ReadOnly property. Change this to **True**.



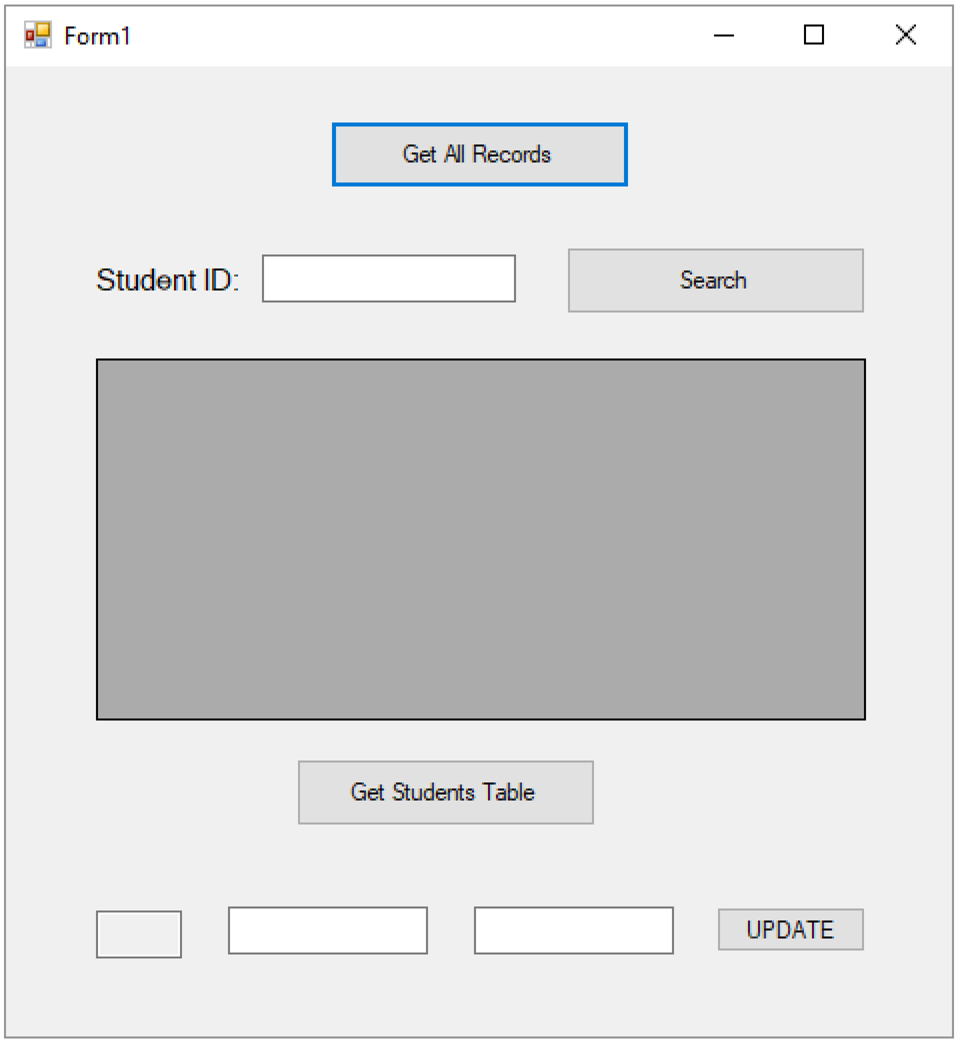
txtID System.Windows.Forms.TextBox

Multiline	False
PasswordChar	
ReadOnly	True
RightToLeft	No
ScrollBars	None
ShortcutsEnabled	True
Size	43, 24
TabIndex	8
TabStop	True
Tag	

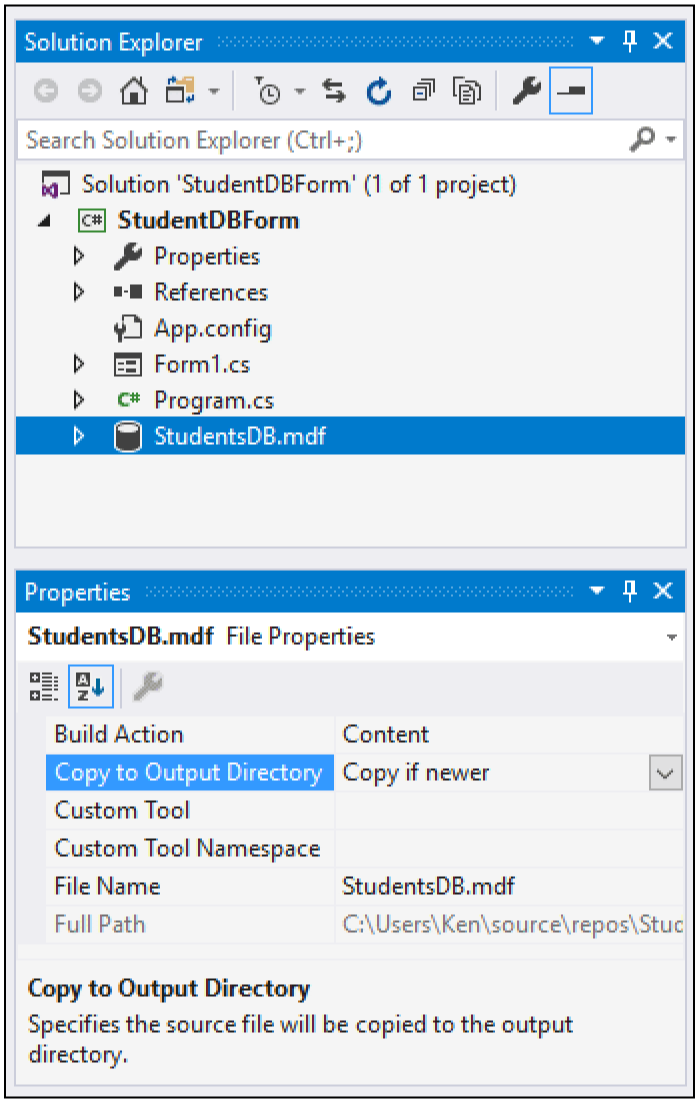
ReadOnly
Controls whether the text in the edit control can be changed or not.

For the other two text boxes, change the names to **txtFamily** and **txtGiven**.

Now add another button. Change the name to BtnUpdate. Change the text on the button to UPDATE. Your form should look something like this:



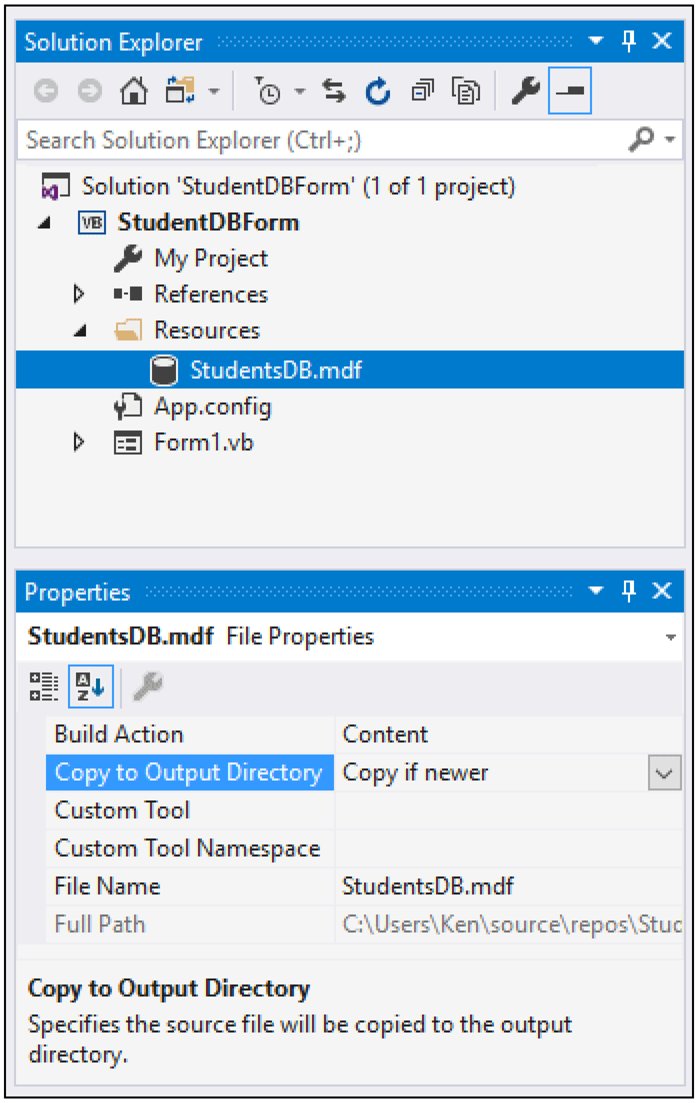
One last thing we can do before the coding. In the Solution Explorer on the right, click on your database to highlight it. In the Properties area, locate the **Copy to Output Directory** item. Change this to **Copy if newer**: (If you don't do this, you won't see any updates that you make. C# on the left, VB on the right.)



StudentsDB.mdf File Properties

Build Action	Content
Copy to Output Directory	Copy if newer
Custom Tool	
Custom Tool Namespace	
File Name	StudentsDB.mdf
Full Path	C:\Users\Ken\source\repos\Stuc

Copy to Output Directory
Specifies the source file will be copied to the output directory.


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'StudentDBForm' (1 of 1 project)
StudentDBForm
My Project
References
Resources
StudentsDB.mdf
App.config
Form1.vb
Properties
StudentsDB.mdf File Properties

Build Action	Content
Copy to Output Directory	Copy if newer
Custom Tool	
Custom Tool Namespace	
File Name	StudentsDB.mdf
Full Path	C:\Users\Ken\source\repos\Stuc

Copy to Output Directory
Specifies the source file will be copied to the output directory.

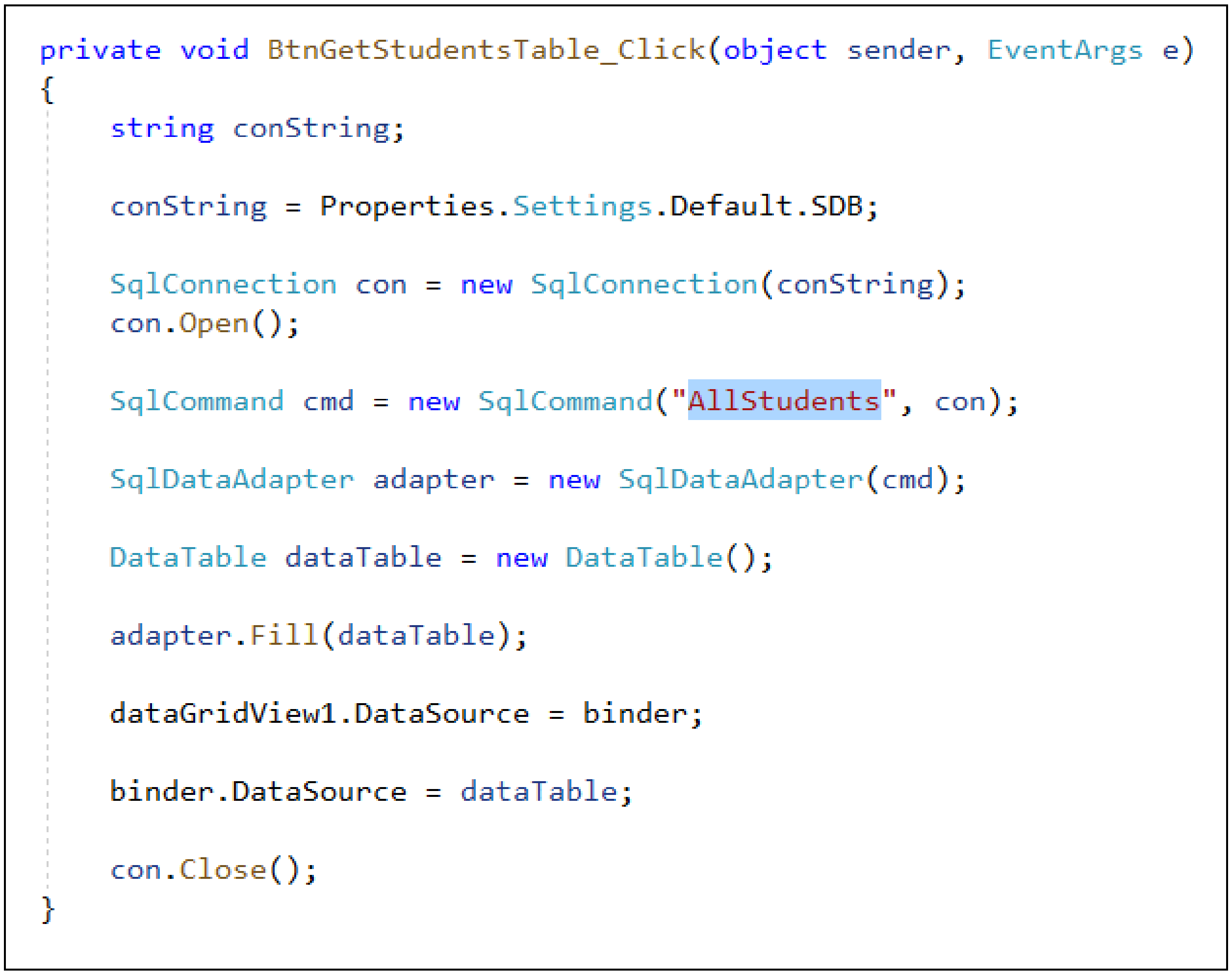
Now we're all set to do the coding.

Double-click your **Get Students Table** button to create the code stub for it.

Again, we can copy and paste the code from the first button, the **BtnAll** one. Again, it would be better if we created a separate method, as there's lots of duplicate code. (It's a good exercise to do this yourself.)

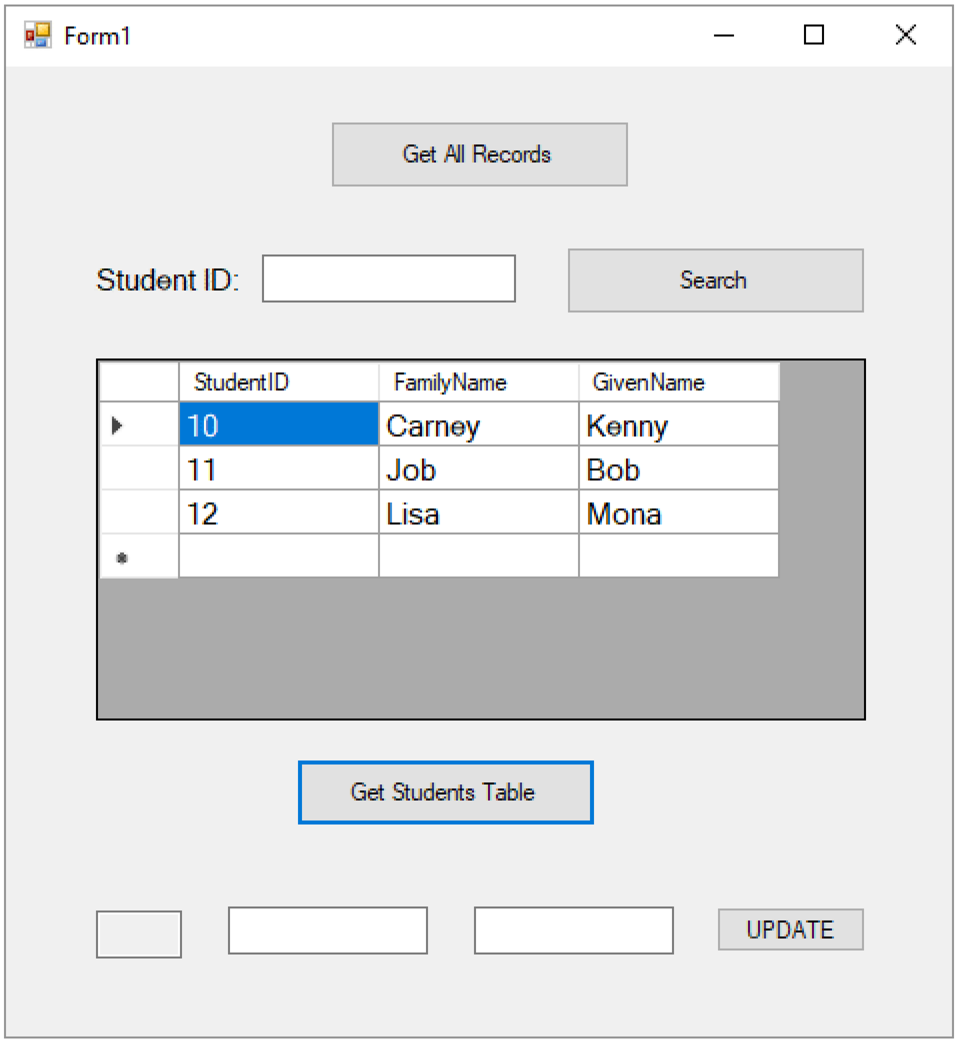
Once you've pasted the code over, change the SqlCommand from NamesAndCourses to AllStudents, which is the name of one of the

Stored Procedures we've just created. But your code should look like this in C#:



```
private void BtnGetStudentsTable_Click(object sender, EventArgs e)
{
    string conString;

    conString = Properties.Settings.Default.SDB;

    SqlConnection con = new SqlConnection(conString);
    con.Open();

    SqlCommand cmd = new SqlCommand("AllStudents", con);

    SqlDataAdapter adapter = new SqlDataAdapter(cmd);

    DataTable dataTable = new DataTable();

    adapter.Fill(dataTable);

    dataGridView1.DataSource = binder;

    binder.DataSource = dataTable;

    con.Close();
}
```

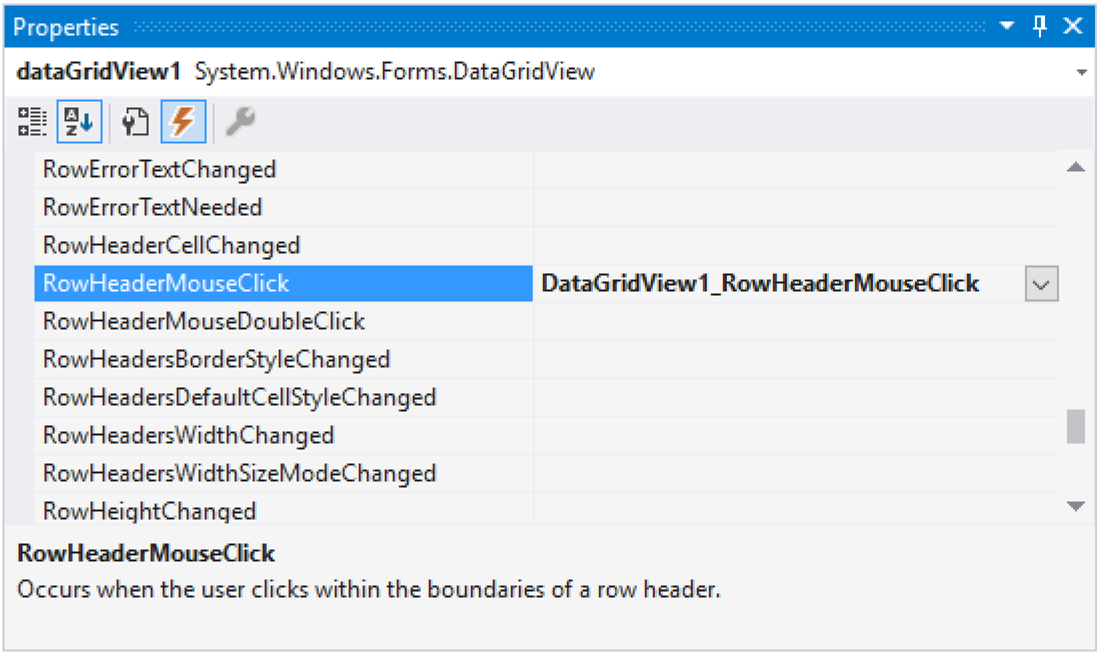
Test it out. Run your form and click the **Get Students Table** button. You should see this:



	StudentID	FamilyName	GivenName
▶	10	Carney	Kenny
	11	Job	Bob
	12	Lisa	Mona
*			

Now what we want to do is to click onto any of the rows and have the data in that row appear in the text boxes. To do this, there is an event of the Data Grid called **RowHeaderMouseClick**.

Go back to your form in design view. Click on your Data Grid to select it. Now have a look at the Properties area. Click the lightning bolt symbol to see a list of events. Double click the one called **RowHeaderMouseClick**:



Properties

dataGridView1 System.Windows.Forms.DataGridView

RowErrorTextChanged	
RowErrorTextNeeded	
RowHeaderCellChanged	
RowHeaderMouseClick	DataGridView1_RowHeaderMouseClick
RowHeaderMouseDoubleClick	
RowHeadersBorderStyleChanged	
RowHeadersDefaultCellStyleChanged	
RowHeadersWidthChanged	
RowHeadersWidthSizeModeChanged	
RowHeightChanged	

RowHeaderMouseClick

Occurs when the user clicks within the boundaries of a row header.

Let's set up some variables, first. Add these to the code stub in C#:

```
string sID;
string fName;
string gName;
```

To get at the row values in a Data Grid, there is a class called DataGridViewRow. This is used to get an array of all the cells in the row. Add the following line:

C#

```
DataGridViewRow selectRow =
dataGridView1.Rows[e.RowIndex];
```

Our object array is called **selectRow**. After the equal sign, we have this:

```
dataGridView1.Rows[e.RowIndex]
```

The **Rows** property gets you which row was selected. But it does that through **e.RowIndex**. The e part comes from round brackets of the Button:

```
          object sender, DataGridViewCellMouseEventArgs e
```

The e event has a property called **RowIndex**, which is the row number you clicked.

We can now access each cell in the row. Add this line:

C#

```
     sID = selectRow.Cells[0].Value.ToString();
```

The difference between the two lines of code is the C# uses square brackets for Cells and VB Net uses round ones.

Our selectRow object has a Cells item. In between square or round brackets, you type the cell number you want. The first cell is 0. You get this Value and convert to a string.

We need two more, so add these lines:

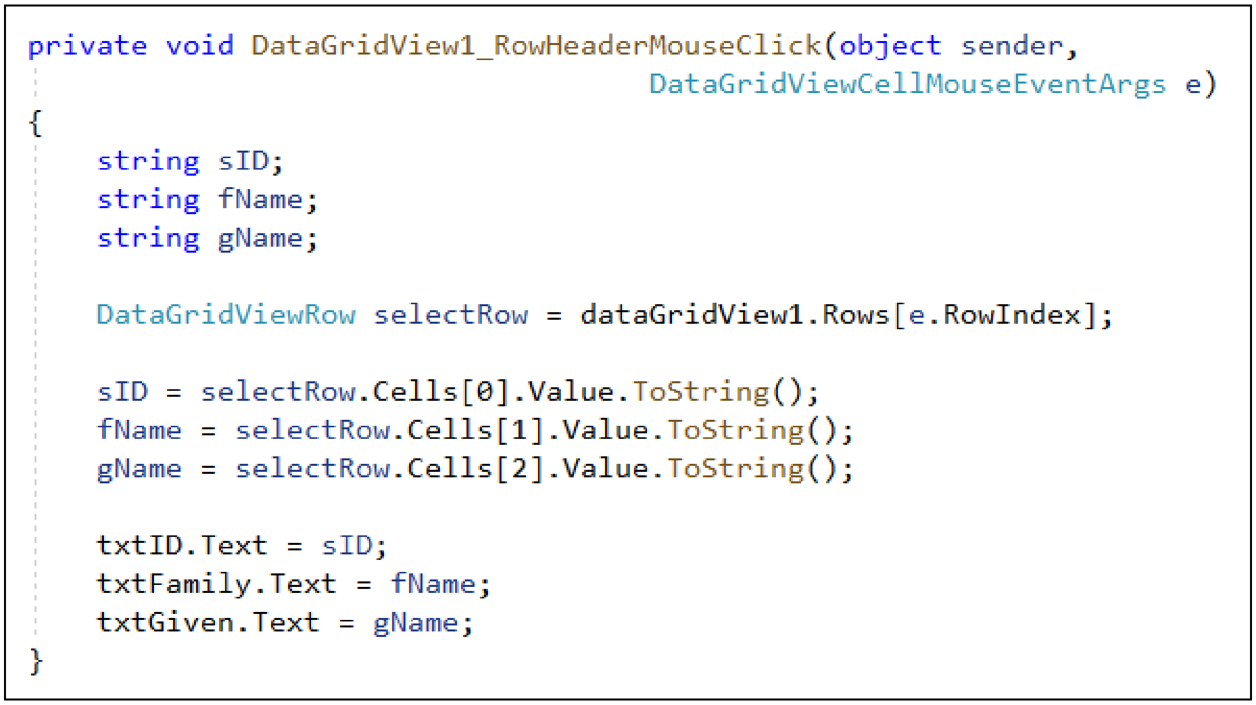
C#

```
     fName = selectRow.Cells[1].Value.ToString();
     gName = selectRow.Cells[2].Value.ToString();
```

Finally, we can place the cell values in the text boxes:

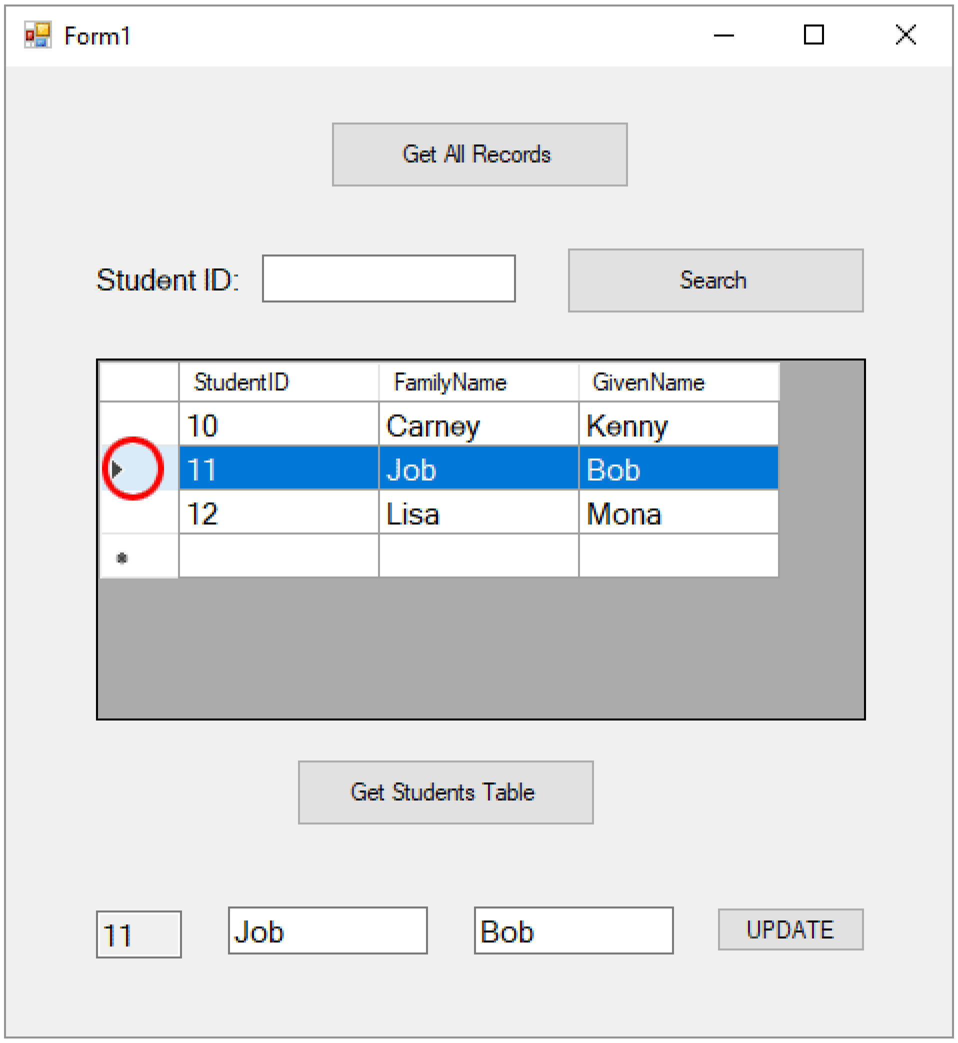
```
txtID.Text = sID;
txtFamily.Text = fName;
txtGiven.Text = gName;
```

Your code should look like this in C#:



```
private void DataGridView1_RowHeaderMouseClick(object sender,
                                     DataGridViewCellMouseEventArgs e)
{
    string sID;
    string fName;
    string gName;

    DataGridViewRow selectRow = dataGridView1.Rows[e.RowIndex];

    sID = selectRow.Cells[0].Value.ToString();
    fName = selectRow.Cells[1].Value.ToString();
    gName = selectRow.Cells[2].Value.ToString();

    txtID.Text = sID;
    txtFamily.Text = fName;
    txtGiven.Text = gName;
}
```

Try it out. Run your form and click your **Get Students Table** button. Now click on one of the row headers, circled in red in the image below:



	StudentID	FamilyName	GivenName
	10	Carney	Kenny
▶	11	Job	Bob
	12	Lisa	Mona
*			

11 Job Bob

When you click on a row header, you should see the row data appear in the text boxes.

Now that row cell data is in the text boxes, it can be edited more easily. After an edit, you can click the UPDATE button. Let's do that now.

Updating the Students Table

First, we can get the data from the text boxes:

C#

```
int studentIdValue = int.Parse(txtID.Text);
string fName = txtFamily.Text;
string gName = txtGiven.Text;
```

Ideally, you'd want to do some error checking here, before going ahead with the rest of the code. You can add an if statement and check that none of the text boxes are blank. But we'll leave that part off.

The next lines will be familiar to you now, and you can copy and paste from elsewhere in your code. Just remember to change the name of your Stored Procedure.

C#

```
string conString;
conString = Properties.Settings.Default.SDB;
SqlConnection con = new SqlConnection(conString);
con.Open();
SqlCommand cmd = new
SqlCommand("UpdateStudentsTable", con);
cmd.CommandType =
CommandType.StoredProcedure;
```

Now we need to add the parameters. There are three this time:

C#

```
cmd.Parameters.Add(new SqlParameter("@sID",
studentIdValue));
cmd.Parameters.Add(new SqlParameter("@Fam",
fName));
cmd.Parameters.Add(new SqlParameter("@Giv",
gName));
```

To actually update the database table, you only need one line of code. Add this (without the semicolon on the end, for VB Net users):

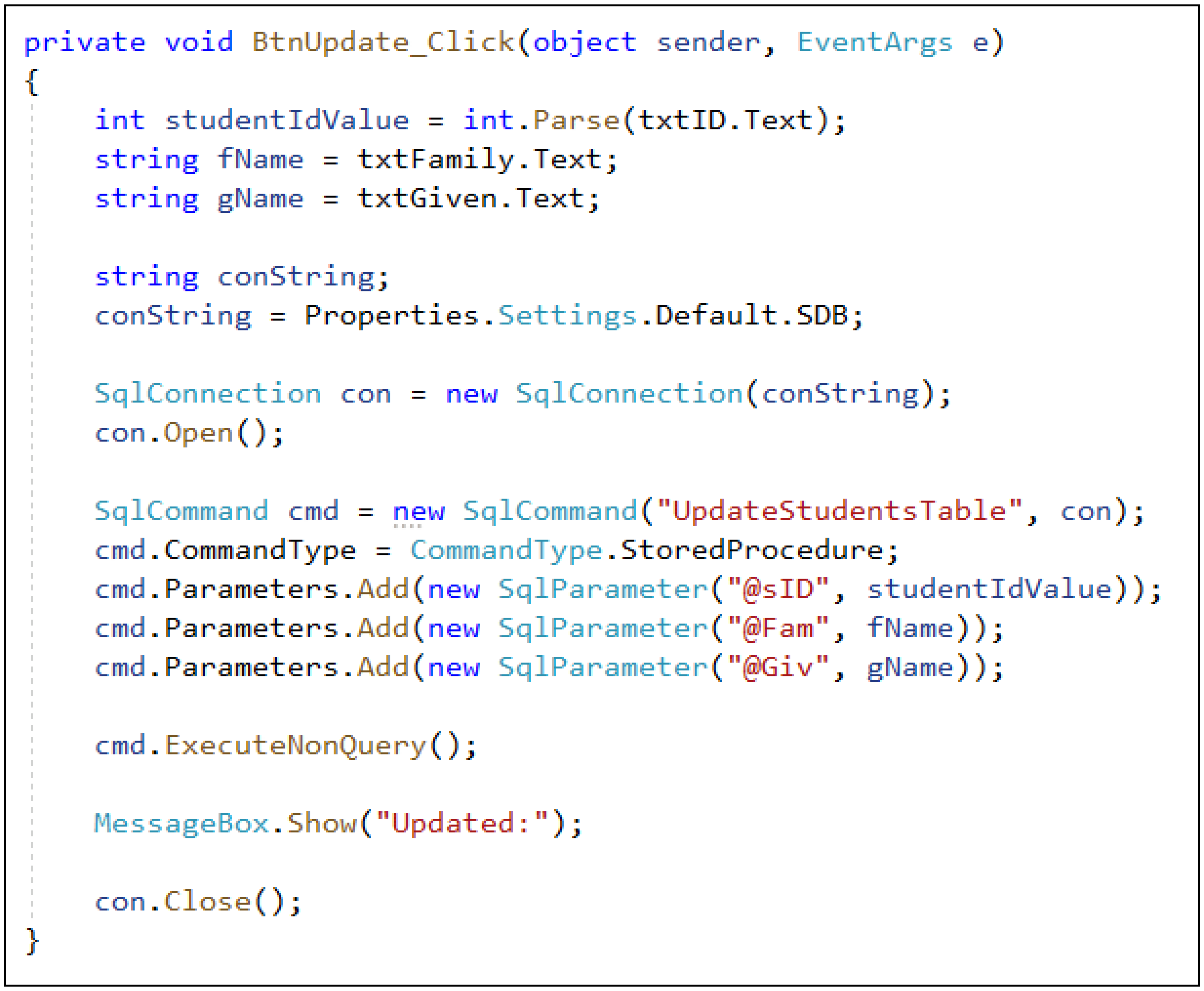
cmd.ExecuteNonQuery();

So it just the method ExecuteNonQuery after your SqlCommand object. That's enough to update the table using your parameter information.

Finally, you can add a message box and close the connection:

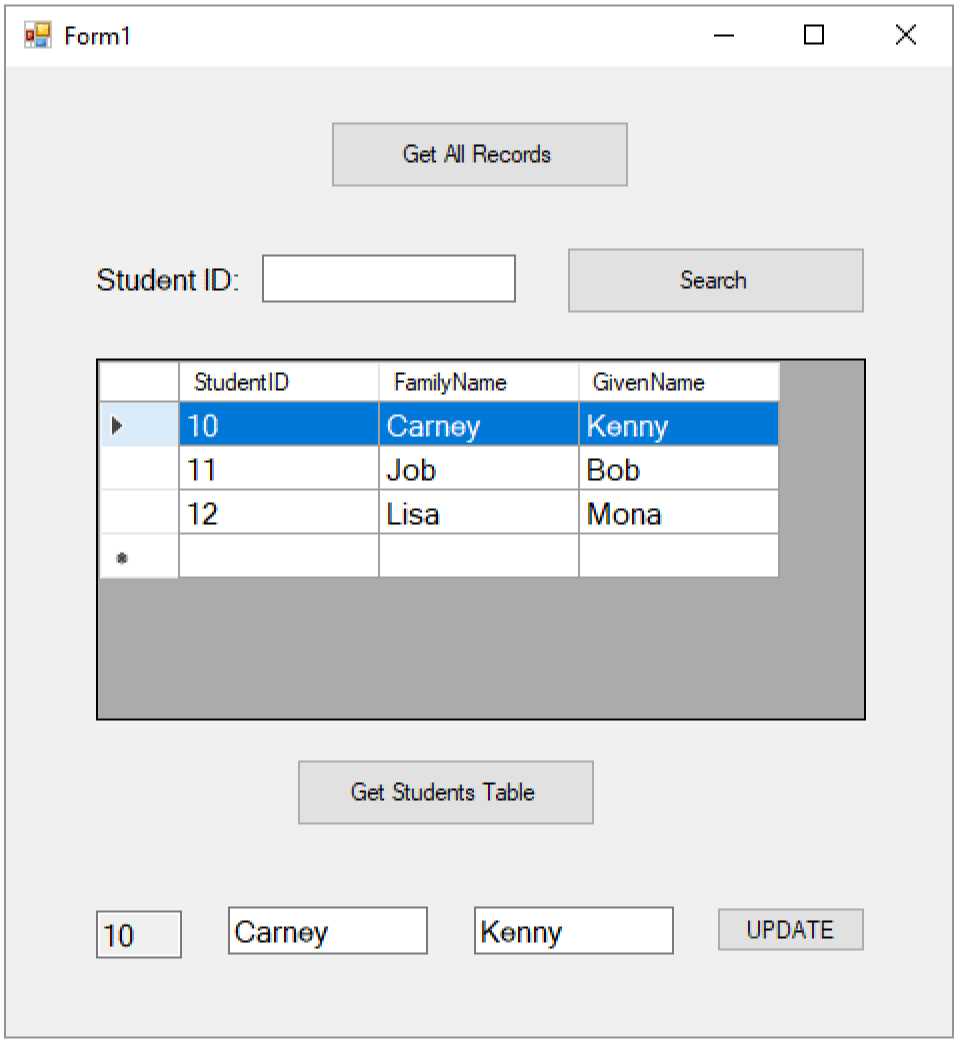
MessageBox.Show("Updated:");
con.Close();

Your code should look like this in C#:



```csharp
private void BtnUpdate_Click(object sender, EventArgs e)
{
    int studentIdValue = int.Parse(txtID.Text);
    string fName = txtFamily.Text;
    string gName = txtGiven.Text;

    string conString;
    conString = Properties.Settings.Default.SDB;

    SqlConnection con = new SqlConnection(conString);
    con.Open();

    SqlCommand cmd = new SqlCommand("UpdateStudentsTable", con);
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.Add(new SqlParameter("@sID", studentIdValue));
    cmd.Parameters.Add(new SqlParameter("@Fam", fName));
    cmd.Parameters.Add(new SqlParameter("@Giv", gName));

    cmd.ExecuteNonQuery();

    MessageBox.Show("Updated:");

    con.Close();
}
```

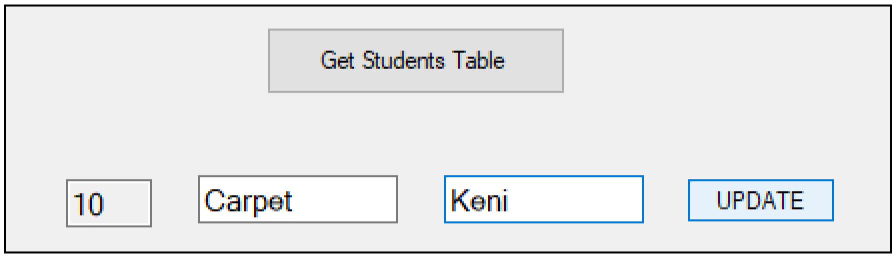
You can give it a try now. Run your form and click your **Get Students Table** button. Click on one of your rows:



	StudentID	FamilyName	GivenName
▶	10	Carney	Kenny
	11	Job	Bob
	12	Lisa	Mona
*			

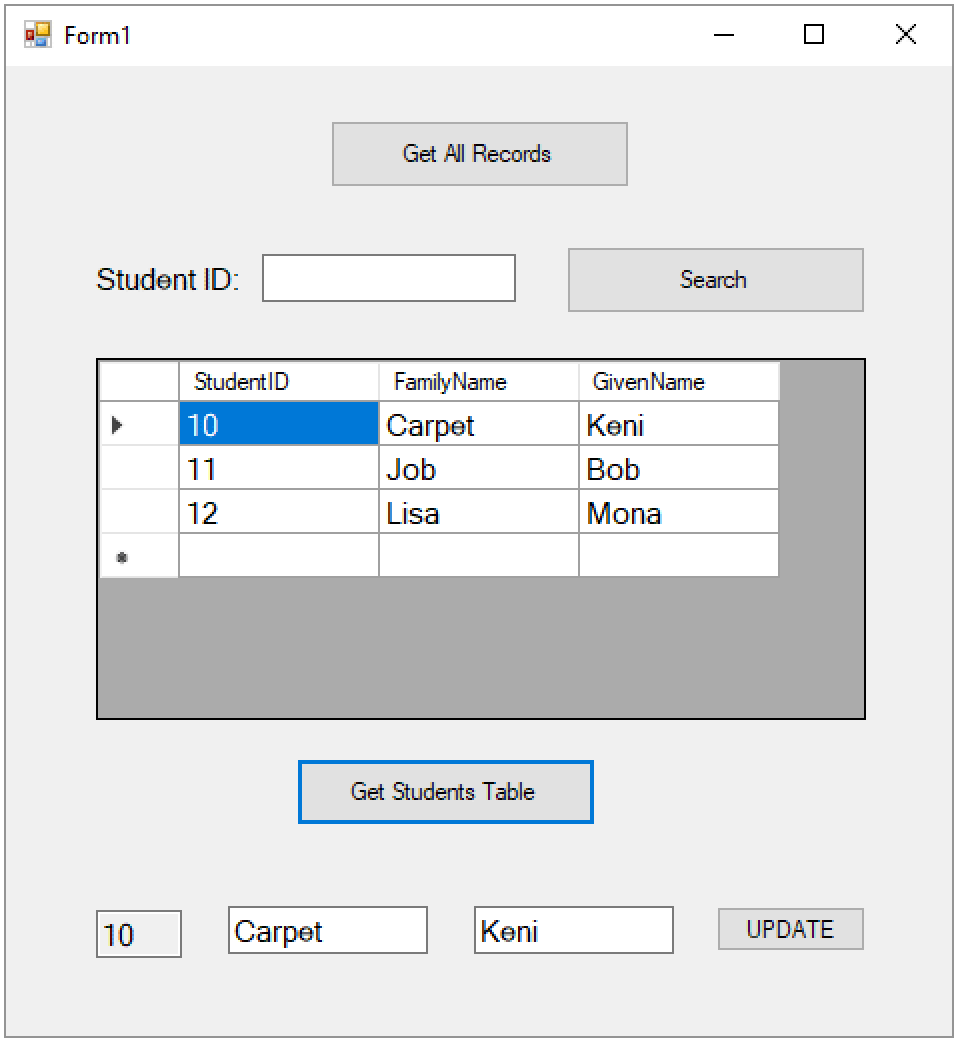
10	Carney	Kenny	UPDATE

Now make changes to the name:



10	Carpet	Keni	UPDATE

Click your UPDATE button to commit the changes to the database.
Now click your **Get Students Table** button again:



	StudentID	FamilyName	GivenName
▶	10	Carpet	Keni
	11	Job	Bob
	12	Lisa	Mona
*			

10 Carpet Keni

Close down your form and relaunch it. You should find the changes you made are still there.

And that's it for this section. You should now have a good idea how to create more than one table in SQL Server, how to add a relationship, how to run queries, and how to create Stored Procedures. You should also have a good working knowledge on how to code for SQL Server databases.