The **SqlClient** namespace interacts with SQL Server. It allows the development of data-driven applications in the C# language. It creates database connections with SqlConnection. It inserts data programmatically with SqlCommand. It handles rows with SqlDataReader. It is an essential tool for building programs that interact with databases.

**Introduction**



This tutorial is based on the local computer, but its steps could be easily applied on the network simply by using a remote database connection string. You will need to use a SQL database.

**In Visual Studio:**Click on Data > Add New Data Source.  
Select "Database" and click "New Connection."



**Using database connection to SQL Server.** In the Data Source box, select Microsoft SQL Server. This will hook up the SQL Database on your local machine. In the Server Name pulldown menu, select SQLExpress. This is present on new installations of Visual Studio 2008 Professional.

**Then:**Save the connection string in the next dialog. You can access this string through your program's settings, which are automatically generated and easy to use. You do not need to manually create a connection string in this tutorial.

**Add namespaces**

If your are working in a console program, you will only need three namespaces. You need to add the System.Data.SqlClient namespace manually. This will give you easy access to the SqlConnection, SqlCommand, and other Sql\* classes.

**Namespaces to include [C#]**

using System; // For system functions like Console.

using System.Collections.Generic; // For generic collections like List.

using System.Data.SqlClient; // For the database connections and objects.

**Create table**

You cannot INSERT a new database table in SQL Server if one by that name already exists. If you do that, the program will throw an exception, which may look like the following one:

**Error:**Unhandled Exception: System.Data.SqlClient.SqlException:  
There is already an object named... in the database.

**Working around the SqlException.** The simplest way to work around this problem is to simply wrap the SqlCommand in a try/catch block. This will tell the runtime to ignore the exception. For more complex programs, you will prefer other techniques. Here is the method that creates the Dogs1 table.

**Program that uses SqlCommand [C#]**

/// <summary>

/// This method attempts to create the Dogs1 SQL table.

/// If will do nothing but print an error if the table already exists.

/// </summary>

static void TryCreateTable()

{

using (SqlConnection con = new SqlConnection(

ConsoleApplication1.Properties.Settings.Default.masterConnectionString))

{

con.Open();

try

{

using (SqlCommand command = new SqlCommand(

"CREATE TABLE Dogs1 (Weight INT, Name TEXT, Breed TEXT)", con))

{

command.ExecuteNonQuery();

}

}

catch

{

Console.WriteLine("Table couldn't be created.");

}

}

}

**CREATE TABLE method.** The above method will create the Dogs1 table with three columns. We are storing the Weight of the dog, the Name of the dog, and the Breed of the dog. The weight is stored as an int, while the other two fields are text fields (strings).

**Description of ExecuteNonQuery.** The ExecuteNonQuery method is ideal for when you are inserting data, not reading it or querying for a specific result. Also, look at how we call the Open instance method on the SqlConnection.



**Using statements.** The using statement in the C# language is excellent for handling important system resources. If you do not clean of the SqlConnection and SqlCommands, your program will have catastrophic failures over time. Also, note the second parameter to the SqlCommand constructor is the opened connection.

[**Using Statement Calls Dispose**](http://www.dotnetperls.com/using)

**Add variables**

Here we see how you can extract the code to insert objects into your SQL database into a method. If your program uses object models, this method could accept those objects. The method here allows Dogs1 data to be inserted by simply passing parameters to AddDog. It uses similar code with SqlConnection and SqlCommand.

**Program that uses SqlParameter [C#]**

/// <summary>

/// Insert dog data into the SQL database table.

/// </summary>

/// <param name="weight">The weight of the dog.</param>

/// <param name="name">The name of the dog.</param>

/// <param name="breed">The breed of the dog.</param>

static void AddDog(int weight, string name, string breed)

{

using (SqlConnection con = new SqlConnection(

ConsoleApplication1.Properties.Settings.Default.masterConnectionString))

{

con.Open();

try

{

using (SqlCommand command = new SqlCommand(

"INSERT INTO Dogs1 VALUES(@Weight, @Name, @Breed)", con))

{

command.Parameters.Add(new **SqlParameter**("Weight", weight));

command.Parameters.Add(new **SqlParameter**("Name", name));

command.Parameters.Add(new **SqlParameter**("Breed", breed));

command.ExecuteNonQuery();

}

}

catch

{

Console.WriteLine("Count not insert.");

}

}

}

**Every SqlCommand instance** has a Parameter collection, which is a SqlParameterCollection and accessed with the Parameters property. It is empty by default, so use the Add instance method to add new parameters. In the Add method, we call the new SqlParameter constructor.

**Notes on SqlParameter constructor.** One of the overloads of SqlParameter, and the one shown here, receives two arguments. First is the name of the variable in your command text, and second is the value object.

**Using object parameters.** In the C# language, every variable inherits from System.Object, so when you need a object parameter as in the SqlParameter constructor, you can use any variable instance.

[**SqlParameter Example**](http://www.dotnetperls.com/sqlparameter)

**SQL injection**



The AddDog method shown above uses variables in the SqlCommand text. This means it is immune to SQL injection attacks. If you are running an ASP.NET website, you will get these every day.

**Tip:**Never insert data into a database table in raw form. Always use SqlParameter and variable names, or question markers.

**Object array**

Your C# programs will almost certainly use object-oriented models for encapsulating data. This means that in our data-driven dog database, we need a Dog class. For simplicity, this class has three public properties, which correspond to the three columns in the database.

**Definition of Dog class [C#]**

/// <summary>

/// Encapsulates data for dog objects.

/// </summary>

public class Dog

{

public int Weight { get; set; }

public string Name { get; set; }

public string Breed { get; set; }

public override string **ToString()**

{

return string.Format("Weight: {0}, Name: {1}, Breed: {2}",

Weight, Name, Breed);

}

}

**We see the** ToString() method in this class. This is overrided and makes it easy for us to display the contents of the class Dog. It is not critical to the database logic in this tutorial.

[**ToString Method**](http://www.dotnetperls.com/tostring)

**SqlDataReader**

Usually, the fastest way to read data out of a table is to use the SqlDataReader class. This class lets you extract individual cells from the database. Because our data is ordered with Weight first, which is an int, we can call GetInt32() with the parameter of 0.

**Program that uses SqlDataReader [C#]**

/// <summary>

/// Read in all rows from the Dogs1 table and store them in a List.

/// </summary>

static void DisplayDogs()

{

List<Dog> dogs = new List<Dog>();

using (SqlConnection con = new SqlConnection(

ConsoleApplication1.Properties.Settings.Default.masterConnectionString))

{

con.Open();

using (SqlCommand command = new SqlCommand("SELECT \* FROM Dogs1", con))

{

**SqlDataReader reader = command.ExecuteReader();**

while (reader.Read())

{

int weight = reader.GetInt32(0); // Weight int

string name = reader.GetString(1); // Name string

string breed = reader.GetString(2); // Breed string

dogs.Add(new Dog() { Weight = weight, Name = name, Breed = breed });

}

}

}

foreach (Dog dog in dogs)

{

Console.WriteLine(dog);

}

}

**In this example,** we use the SqlConnection and SqlCommand ADO.NET patterns. Look at the string parameter to the SqlCommand constructor, and it is an SQL command that selects all cells from each SQL table row. The star in the command means "all". The "Dogs1" table is also specified.

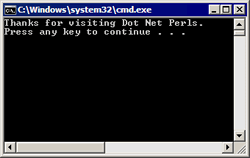
**SqlReader is fast,** but is not ideal for all situations. In Windows Forms, where you usually bind database tables to Controls, you can use SqlDataAdapter and the DataSource property on the Controls.

[**SqlDataReader**](http://www.dotnetperls.com/sqldatareader)[**SqlDataAdapter**](http://www.dotnetperls.com/sqldataadapter)

**Finally:**The foreach near the end of the example above displays the List collection that was filled with the data in the "Dogs1" table.

**Tip:**When you execute the program, you will see lines of each dog's data that have put into the database. The ToString() method is called implicitly when you specify the parameter to Console.WriteLine.

**Console example**



For the tutorial here, we use a Console program with an event loop. What this does is allow you to input data into the program and have it dynamically INSERTed into the SQL Server database. This code is only useful for this demonstration. However, when developing other data-driven applications, you can use similar logic for inserting user data, such as that entered on web pages.

**Program that calls AddData [C#]**

static void Main()

{

TryCreateTable();

while (true)

{

Console.WriteLine("INPUT TYPE:\tWeight,Name,Breed\tor\tSELECT:");

string[] input = Console.ReadLine().Split(',');

try

{

char c = char.ToLower(input[0][0]);

if (c == 's')

{

DisplayDogs();

continue;

}

int weight = int.Parse(input[0]); // The dog weight.

string name = input[1]; // The name string.

string breed = input[2]; // The breed string.

AddDog(weight, name, breed); // Add the data to the SQL database.

}

catch

{

Console.WriteLine("Input error");

}

}

}



**This Main method,** which could be placed in the Program class in the Console application, first makes sure the database table we are using has been created. This is done by calling the TryCreateTable method. That method is shown above.

**The parsing code.** The next few lines in the Main method above parse the Console input from the ReadLine method. You should be familiar with the basics of the Split method and the int.Parse method.

**Note:**See the relevant articles on the site for more information.

[**Split String Examples**](http://www.dotnetperls.com/split)[**int.Parse for Integer Conversion**](http://www.dotnetperls.com/int-parse)

**Tip:**The methods shown here could be applied to any new data-driven application. If your program uses lots of data, you can use Visual Studio to add indexes on columns that can be selected on. This can vastly improve performance.

**Input, output**

The program described and developed in the first 10 steps of this tutorial is fun to play with. The lines from the output of the program shown next demonstrate how the rows are input into the database, and then SELECTed and iterated through.

**Input**

INPUT TYPE: Weight,Name,Breed or SELECT:

57,Koko,Shar Pei

INPUT TYPE: Weight,Name,Breed or SELECT:

130,Fido,Bullmastiff

INPUT TYPE: Weight,Name,Breed or SELECT:

93,Alex,Anatolian Shepherd Dog

INPUT TYPE: Weight,Name,Breed or SELECT:

25,Charles,Cavalier King Charles Spaniel

INPUT TYPE: Weight,Name,Breed or SELECT:

7,Candy,Yorkshire Terrier

INPUT TYPE: Weight,Name,Breed or SELECT:

s

Weight: 57, Name: Koko, Breed: Shar Pei

Weight: 130, Name: Fido, Breed: Bullmastiff

Weight: 93, Name: Alex, Breed: Anatolian Shepherd Dog

Weight: 25, Name: Charles, Breed: Cavalier King Charles Spaniel

Weight: 7, Name: Candy, Breed: Yorkshire Terrier

INPUT TYPE: Weight,Name,Breed or SELECT:

**I painstakingly entered** five dogs into the database. Next, I got tired and exited the program. The text that we see next demonstrates that the data was persisted successfully in the database. The dogs are now in Dog objects.

**Output**

Table couldn't be created.

INPUT TYPE: Weight,Name,Breed or SELECT:

s

Weight: 57, Name: Koko, Breed: Shar Pei

Weight: 130, Name: Fido, Breed: Bullmastiff

Weight: 93, Name: Alex, Breed: Anatolian Shepherd Dog

Weight: 25, Name: Charles, Breed: Cavalier King Charles Spaniel

Weight: 7, Name: Candy, Breed: Yorkshire Terrier

INPUT TYPE: Weight,Name,Breed or SELECT:

**Program**

The complete program source is available below. Before you can use the source code, you will need to change the namespace for the program. You will also need to follow the steps in part 1 to create your connection string.

**Complete program [C#]**

using System;

using System.Collections.Generic;

using System.Data.SqlClient;

class Program

{

static void Main()

{

TryCreateTable();

while (true)

{

Console.WriteLine("INPUT TYPE:\tWeight,Name,Breed\tor\tSELECT:");

string[] input = Console.ReadLine().Split(',');

try

{

char c = char.ToLower(input[0][0]);

if (c == 's')

{

DisplayDogs();

continue;

}

int weight = int.Parse(input[0]);

string name = input[1];

string breed = input[2];

AddDog(weight, name, breed);

}

catch

{

Console.WriteLine("Input error");

}

}

}

/// <summary>

/// This method attempts to create the Dogs1 SQL table.

/// If will do nothing but print an error if the table already exists.

/// </summary>

static void TryCreateTable()

{

using (SqlConnection con = new SqlConnection(ConsoleApplication1.Properties.Settings.Default.masterConnectionString))

{

con.Open();

try

{

using (SqlCommand command = new SqlCommand("CREATE TABLE Dogs1 (Weight INT, Name TEXT, Breed TEXT)", con))

{

command.ExecuteNonQuery();

}

}

catch

{

Console.WriteLine("Table couldn't be created.");

}

}

}

/// <summary>

/// Insert dog data into the SQL database table.

/// </summary>

/// <param name="weight">The weight of the dog.</param>

/// <param name="name">The name of the dog.</param>

/// <param name="breed">The breed of the dog.</param>

static void AddDog(int weight, string name, string breed)

{

using (SqlConnection con = new SqlConnection(ConsoleApplication1.Properties.Settings.Default.masterConnectionString))

{

con.Open();

try

{

using (SqlCommand command = new SqlCommand("INSERT INTO Dogs1 VALUES(@Weight, @Name, @Breed)", con))

{

command.Parameters.Add(new SqlParameter("Weight", weight));

command.Parameters.Add(new SqlParameter("Name", name));

command.Parameters.Add(new SqlParameter("Breed", breed));

command.ExecuteNonQuery();

}

}

catch

{

Console.WriteLine("Count not insert.");

}

}

}

/// <summary>

/// Read in all rows from the Dogs1 table and store them in a List.

/// </summary>

static void DisplayDogs()

{

List<Dog> dogs = new List<Dog>();

using (SqlConnection con = new SqlConnection(ConsoleApplication1.Properties.Settings.Default.masterConnectionString))

{

con.Open();

using (SqlCommand command = new SqlCommand("SELECT \* FROM Dogs1", con))

{

SqlDataReader reader = command.ExecuteReader();

while (reader.Read())

{

int weight = reader.GetInt32(0); // Weight int

string name = reader.GetString(1); // Name string

string breed = reader.GetString(2); // Breed string

dogs.Add(new Dog() { Weight = weight, Name = name, Breed = breed });

}

}

}

foreach (Dog dog in dogs)

{

Console.WriteLine(dog);

}

}

}

/// <summary>

/// Encapsulates data for dog objects.

/// </summary>

public class Dog

{

public int Weight { get; set; }

public string Name { get; set; }

public string Breed { get; set; }

public override string ToString()

{

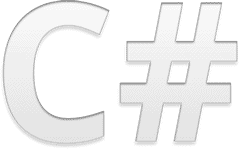
return string.Format("Weight: {0}, Name: {1}, Breed: {2}",

Weight, Name, Breed);

}

}

**Summary**

**[](http://www.dotnetperls.com/data)**

We saw a detailed tutorial for using the System.Data.SqlClient namespace and Visual Studio to develop a data-driven console program. We also built object models that can be used in other C# code for data processing and logic.

**Review:**These patterns of using SqlConnection and SqlCommand will result in secure code that cannot be hacked. You can conserve system resources.