Intro To ADO.NET

.NET Cohort

SOFTWARE GUILD

Lesson Goals

- Learn about the ADO.NET data providers and namespaces
- Learn how to work with connections, commands, and data readers

What is ADO.NET

- ADO stands for *Active Data Objects* and it pre-dates the .NET Framework. The original ADO was the common set of classes for interacting with databases. ADO.NET is the evolution of ADO.
- ADO.NET was built to serve both persistent and disconnected data needs.
- We will find the core ADO.NET classes in the System.Data namespace.

Three Layers of ADO.NET

- *Connected*
  - These classes explicitly connect and disconnect from the database. It is the most flexible, but also takes the most work to set up.
- *Disconnected*
  - These classes allow you to store data in Tables and Sets disconnected from the database and handle bulk updates.
- *Entity Framework*
  - The Entity Framework is an ORM (object relational mapper) tool which can automate the creation of classes that map to tables and allows manipulation through LINQ queries. For straightforward apps, this saves a lot of time.

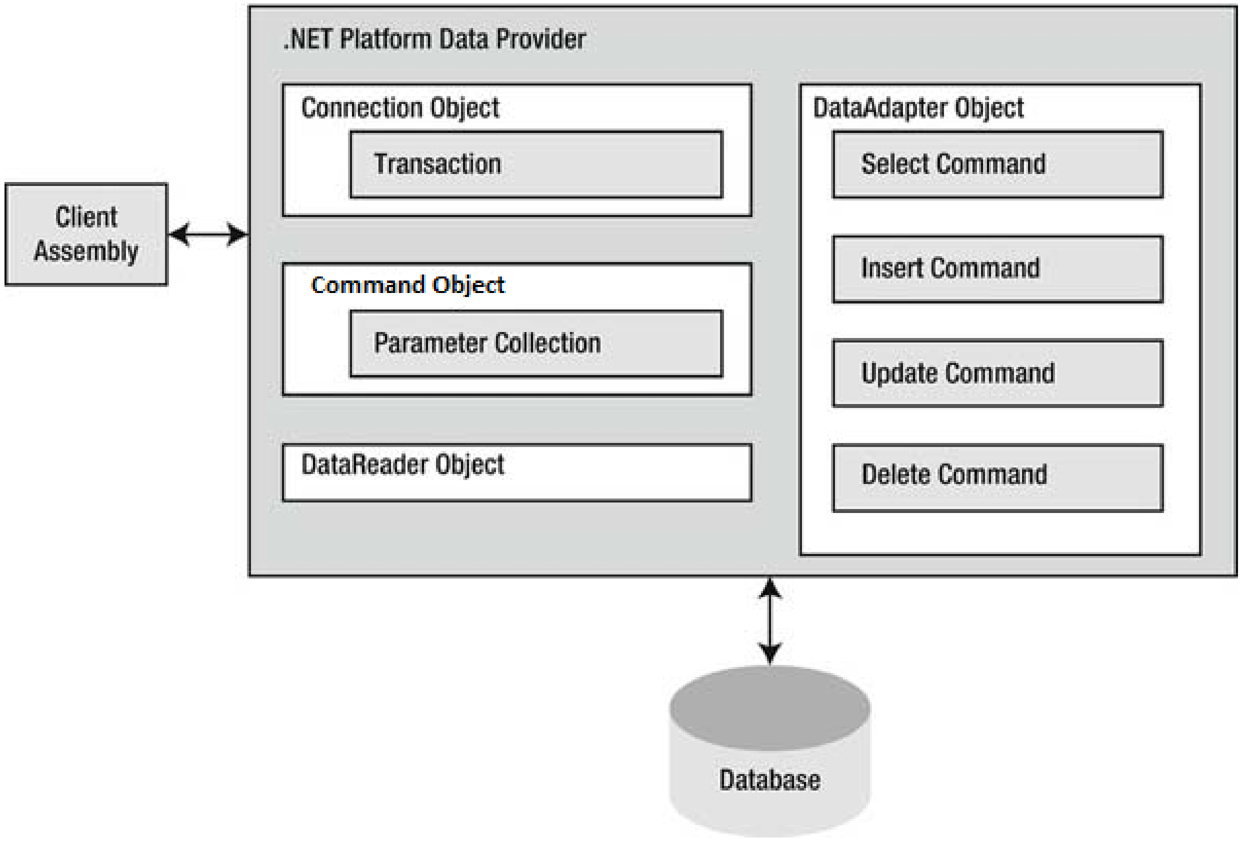
There are MANY other choices

- Dapper, NHibernate, LLBGen, etc. etc. etc.
- The ADO.NET core uses interfaces to define all the key components, so developers can easily roll out their own data access frameworks and provide libraries for connecting to other databases.
  - These libraries are called *drivers*. So, if you want to connect C# to say, MySQL, just search for the MySQL .NET Driver and reference the DLL.

Core ADO.NET Classes

Type	Base Class	Interfaces	Usage
Connection	DbConnection	IDbConnection	Connect and disconnect from a data store, register transactions
Command	DbCommand	IDbCommand	Handle the execution of SQL Queries or Stored Procedures
DataReader	DbDataReader	IDataReader, IDataRecord	Provides **forward-only**, **read-only** access to data. Persistent connection.
DataAdapter	DbDataAdapter	IDataAdapter, IDbDataAdapter	Transfers sets of data between caller and data store
Parameter	DbParameter	IDataParameter, IDbDataParameter	Transfers parameters to and from the server in queries
Transaction	DbTransaction	IDbTransaction	Handles database transactions

ADO.NET Providers Illustrated



Out-of-the-Box Providers

- OLEDB (System.Data.OleDb)
  - This is a classic COM based protocol. Used for old legacy access databases, pervasive, etc.
- SQL Server (System.Data.SqlClient)
  - Highly optimized for SQL Server databases, only connects to SQL Server
- Oracle (System.Data.OracleClient)
  - Do not use. Oracle provides its own implementation now, go download it.
- ODBC (System.Data.Odbc)
  - Most databases support ODBC, so you can use this if there is no .NET provider available.

SOFTWARE GUILD

Reading Data From a SQL Server Database

When using basic ADO.NET, there are a few steps to connecting to a SQL Server Database:

1. Create a connection
2. Create a command
3. Loop through the data reader
4. Close the reader
5. Close the connection

Creating a Connection

- First, we need to add a using statement for System.Data.SqlClient.
- Second, we need to be able to provide our connection with a *connection string.*
  - Connection strings tell the connection class which server to connect to, what database to use, and specifies credentials (login/pwd) if necessary.

Aside: Configuration Files

- In a real production environment, we will often have Dev, QA, and Production servers and we need to be able to change things, like which database we connect to, without having to recompile the application.
- Configuration files allow us to change settings slightly by modifying a text file that lives outside the program.
- These files are XML format, and have two sections we can define:
  - *appSettings* is for any setting, like file paths, etc.
  - *connectionStrings* is for database connections.

SOFTWAREGUILD

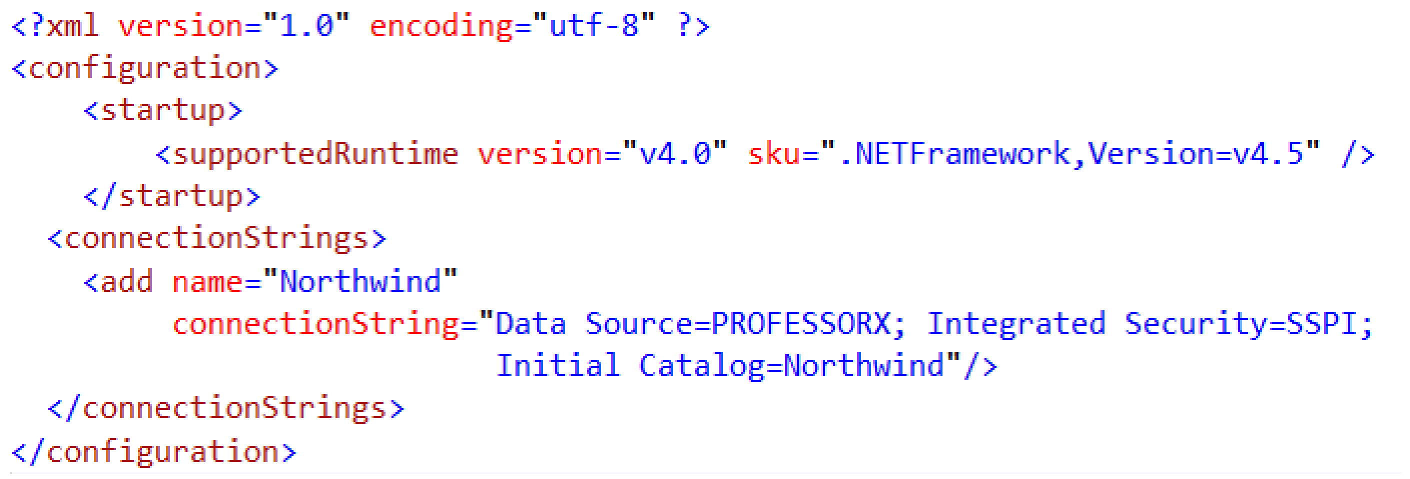
Note: Security

- In a real application, you may want to consider encrypting your config file's appSettings and connectionStrings.

http://msdn.microsoft.com/en-us/library/zhhddkxy.aspx.

Adding a ConnectionStrings Section

- Open your app.config file in the console app. (Or, if one isn't there, go to Add → New Item and choose Application Configuration File.)
- Add the following text:



```xml
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
    </startup>
  <connectionStrings>
    <add name="Northwind"
         connectionString="Data Source=PROFESSORX; Integrated Security=SSPI;
                           Initial Catalog=Northwind"/>
  </connectionStrings>
</configuration>
```

Connection String Elements

- Server — Name or IP address of the server
- Database — Name of the database on the server
- User Id — user name to log in (if not using Windows authentication)
- Password — password for user id
- Trusted_Connection — true if using Windows authentication

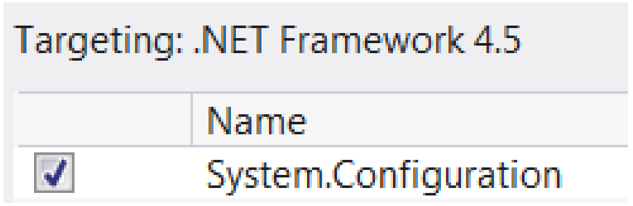
For more, see www.connectionstrings.com

Configuration Manager

We can read data from the configuration file by key or name by using the configuration manager in the system.configuration namespace.

1. Add a reference to system.configuration.
2. Add a using statement.
3. The configuration manager is a static class, so we don't have to new it up. We can access the connection strings by name.

Note: in a website, the file would be web.config instead of app.config.



Targeting: .NET Framework 4.5

	Name
✓	System.Configuration

```
using System.Configuration;
```

```
string connectionString;
connectionString = ConfigurationManager
                       .ConnectionStrings["Northwind"]
                       .ConnectionString;
```

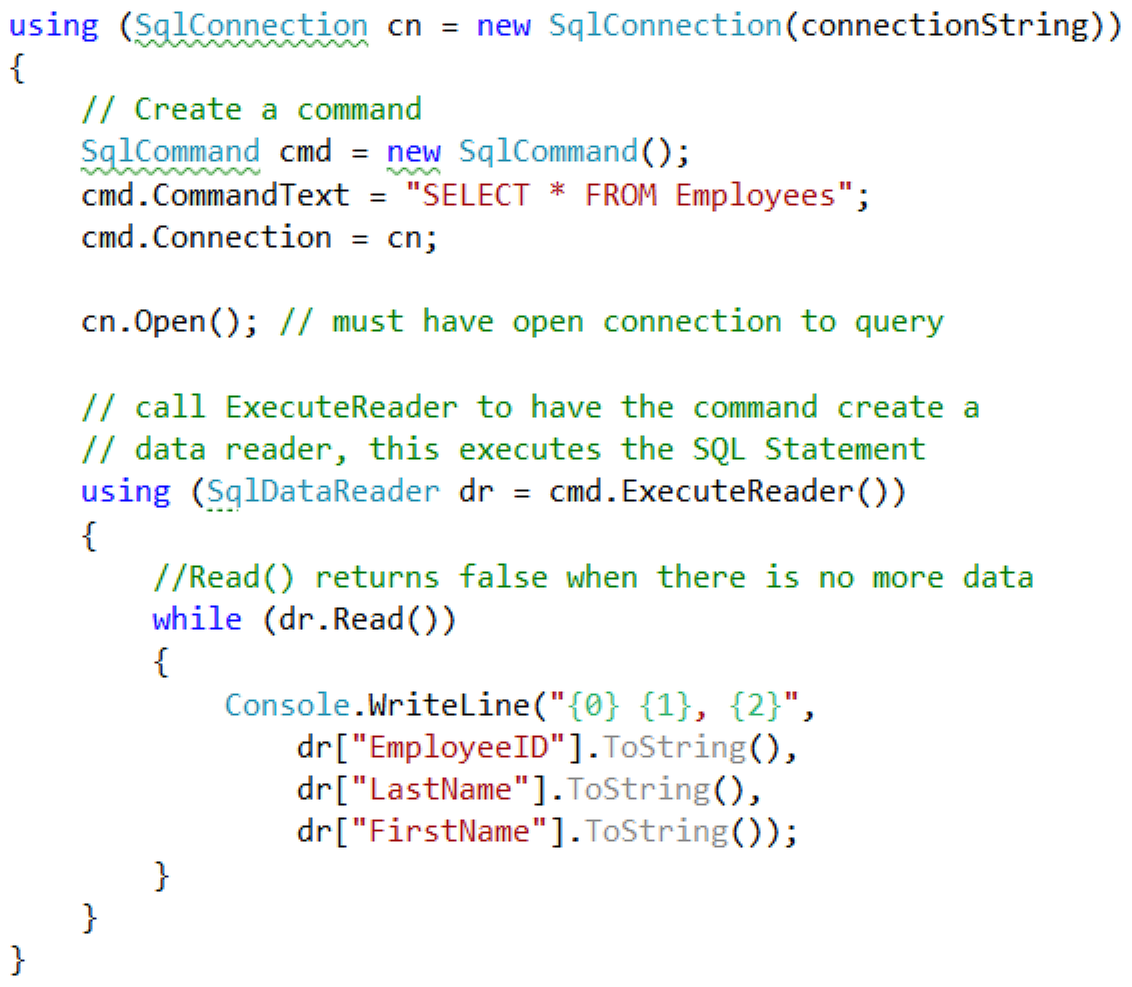
Running a SELECT Statement

Here, we are creating a connection for our connection string, configuring a command with some SQL we want to execute, and looping through a data reader to print each result row out to the console.

The using statements automatically close the connection and reader for us.

DataReader columns are accessed by name and are objects by default. So, if we load a class, we have to cast the column to the right type. For example:

int id = (int)dr[“EmployeeID”]



```
using (SqlConnection cn = new SqlConnection(connectionString))
{
    // Create a command
    SqlCommand cmd = new SqlCommand();
    cmd.CommandText = "SELECT * FROM Employees";
    cmd.Connection = cn;

    cn.Open(); // must have open connection to query

    // call ExecuteReader to have the command create a
    // data reader, this executes the SQL Statement
    using (SqlDataReader dr = cmd.ExecuteReader())
    {
        //Read() returns false when there is no more data
        while (dr.Read())
        {
            Console.WriteLine("{0} {1}, {2}",
                dr["EmployeeID"].ToString(),
                dr["LastName"].ToString(),
                dr["FirstName"].ToString());
        }
    }
}
```

Command Execute Methods

- ExecuteReader — runs the SQL statement and returns a DataReader with all the result rows in it.
- ExecuteNonQuery — runs a SQL statement where there is no result set (e.g. Update, Insert, Delete)
- ExecuteScalar — runs a SQL statement where only one value is returned (e.g. Count)

Parameterized Queries

- We should NEVER append text to create SQL statements. This opens the door to SQL Injection Attacks, which are one of the most common hacking techniques.
- SQL Server will automatically wrap parameters so that injection attacks are blocked.
- Parameters are variables added to the command object and, in SQL, they start with the @symbol.

SOFTWARE GUILD

Demo Goal

- Be able to get a list of employees and their manager (if they have one)
- Be able to load a single employee by their ID
- Handle null data using nullable properties
- Pass parameters to commands

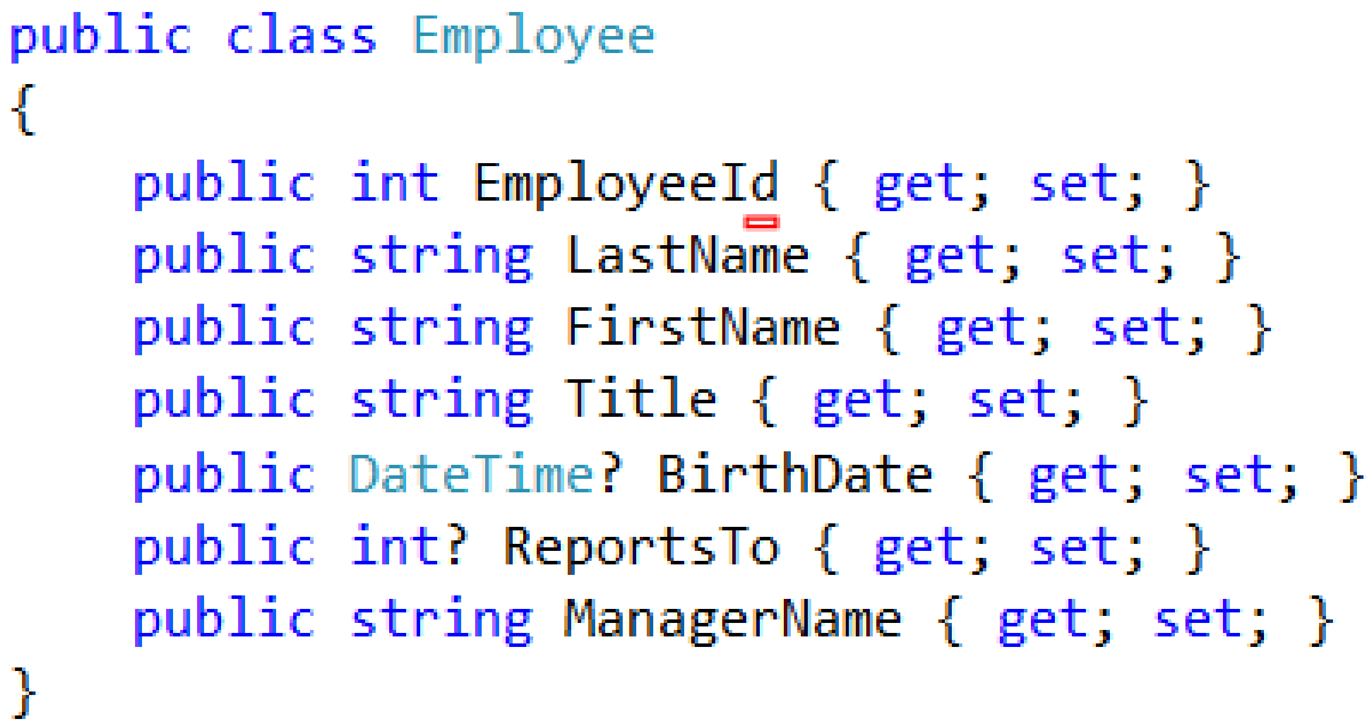
Reading data into classes & parameterized queries

DEMO

Nullable Types

- Any type that doesn’t support null inherently (numeric types, booleans, etc.) will default to 0 (or false).
- In databases where we can have null columns, we want to be able to have a null number for optional fields — 0 is not valid
- Solution: append a question mark to the type declaration.
    - We have to take extra care when working with these fields or we will get a null reference exception!

Employee Class With Nullables



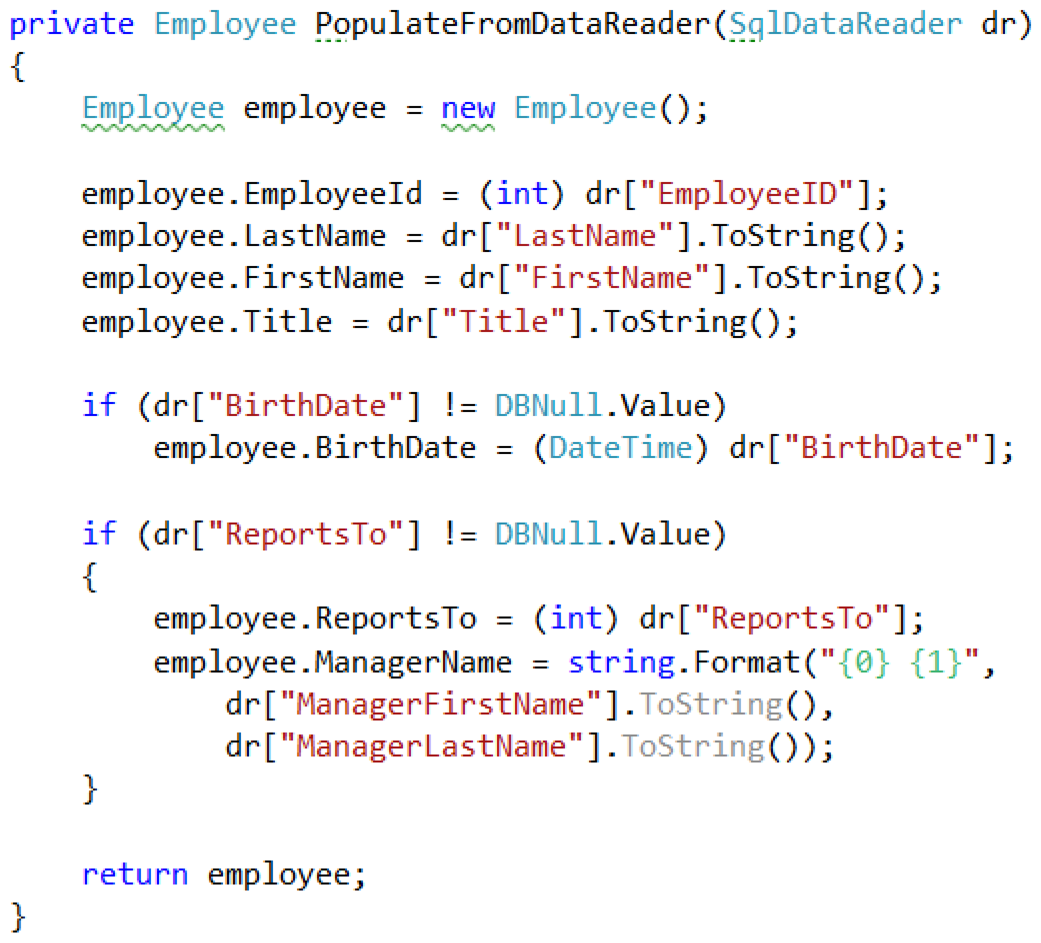
```
public class Employee
{
    public int EmployeeId { get; set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
    public string Title { get; set; }
    public DateTime? BirthDate { get; set; }
    public int? ReportsTo { get; set; }
    public string ManagerName { get; set; }
}
```

Method to Create Employee from DataReader

Because we have to read each row from the data reader, it’s a common practice to create a method that takes the data reader in and returns an employee.

DataReader columns are of type object, so we have to cast them.

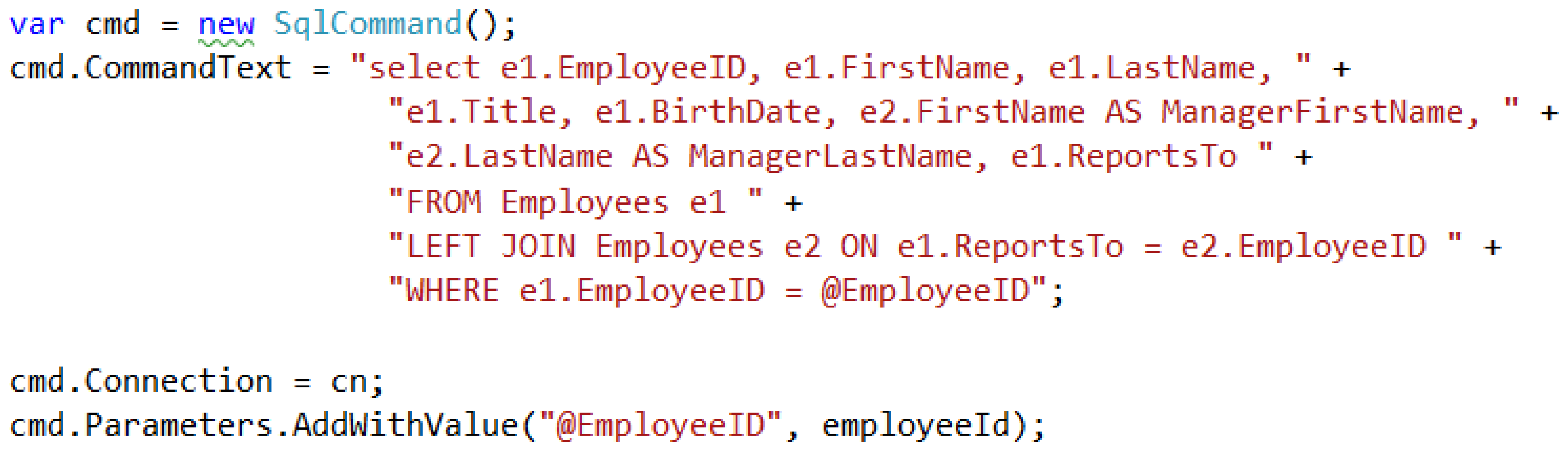
If we want to see if a column is null, we have to compare it to DBNull.Value



```
private Employee PopulateFromDataReader(SqlDataReader dr)
{
    Employee employee = new Employee();

    employee.EmployeeId = (int) dr["EmployeeID"];
    employee.LastName = dr["LastName"].ToString();
    employee.FirstName = dr["FirstName"].ToString();
    employee.Title = dr["Title"].ToString();

    if (dr["BirthDate"] != DBNull.Value)
        employee.BirthDate = (DateTime) dr["BirthDate"];

    if (dr["ReportsTo"] != DBNull.Value)
    {
        employee.ReportsTo = (int) dr["ReportsTo"];
        employee.ManagerName = string.Format("{0} {1}",
            dr["ManagerFirstName"].ToString(),
            dr["ManagerLastName"].ToString());
    }

    return employee;
}
```

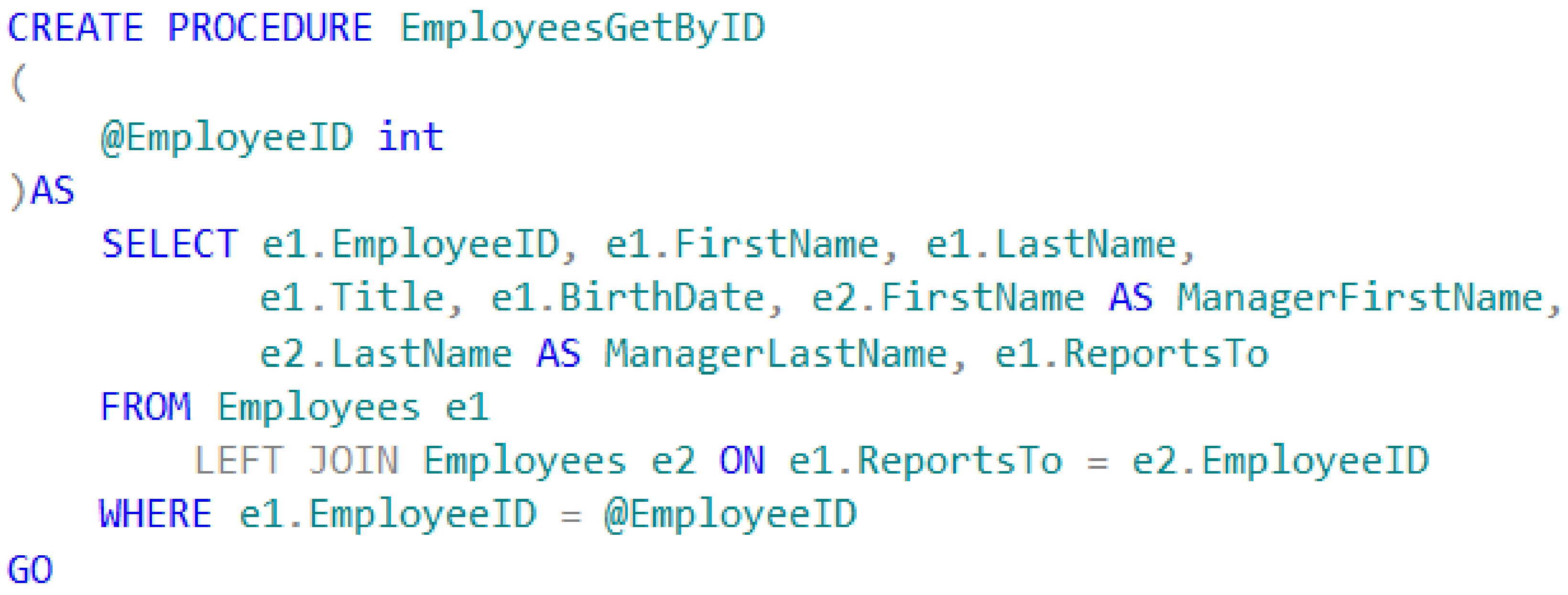
Parameterized SQL

- As we covered in stored procedures, we need to parameterize our SQL to avoid SQL Injection attacks.
- The command object has an AddWithParameter to pass parameters to SQL Statements and Stored Procedures.



```
var cmd = new SqlCommand();
cmd.CommandText = "select e1.EmployeeID, e1.FirstName, e1.LastName, " +
                    "e1.Title, e1.BirthDate, e2.FirstName AS ManagerFirstName, " +
                    "e2.LastName AS ManagerLastName, e1.ReportsTo " +
                    "FROM Employees e1 " +
                    "LEFT JOIN Employees e2 ON e1.ReportsTo = e2.EmployeeID " +
                    "WHERE e1.EmployeeID = @EmployeeID";

cmd.Connection = cn;
cmd.Parameters.AddWithValue("@EmployeeID", employeeId);
```

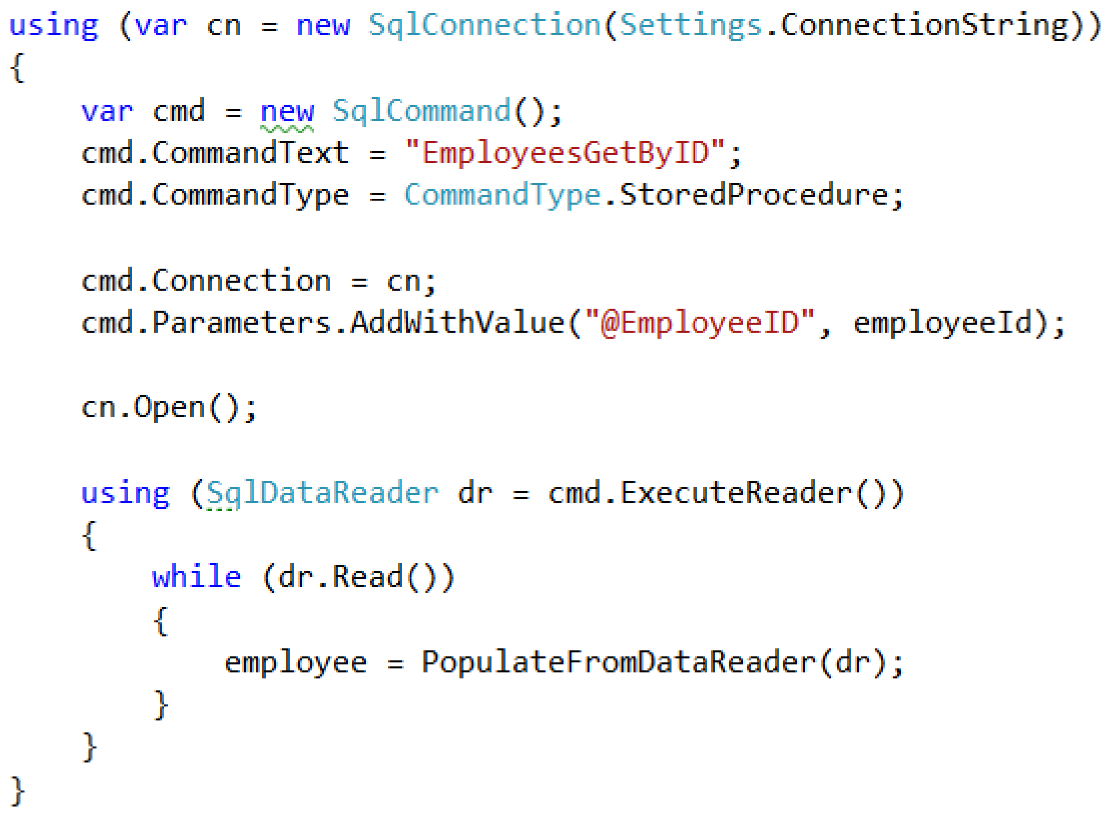
Let’s make a SPROC



```sql
CREATE PROCEDURE EmployeesGetByID
(
    @EmployeeID int
)AS
    SELECT e1.EmployeeID, e1.FirstName, e1.LastName,
           e1.Title, e1.BirthDate, e2.FirstName AS ManagerFirstName,
           e2.LastName AS ManagerLastName, e1.ReportsTo
    FROM Employees e1
        LEFT JOIN Employees e2 ON e1.ReportsTo = e2.EmployeeID
    WHERE e1.EmployeeID = @EmployeeID
GO
```

Calling Stored Procedures

- Pretty much the same as writing SQL directly — we just put in the procedure name and set the command type to StoredProcedure.



```
using (var cn = new SqlConnection(Settings.ConnectionString))
{
    var cmd = new SqlCommand();
    cmd.CommandText = "EmployeesGetByID";
    cmd.CommandType = CommandType.StoredProcedure;

    cmd.Connection = cn;
    cmd.Parameters.AddWithValue("@EmployeeID", employeeId);

    cn.Open();

    using (SqlDataReader dr = cmd.ExecuteReader())
    {
        while (dr.Read())
        {
            employee = PopulateFromDataReader(dr);
        }
    }
}
```

Enhance our RSVP sample to store data in the database and display a guest list

DEMO

Conclusion

Any SQL Statement we can run in SQL Studio, we can also run from C# to select data.

Commands allow us the flexibility to call statements, call procedures, and add data to our calls to pass to the database.