Guillermo "Guille" Som

Conexiones y lectores en ADO.NET 2.0

En el taller de este mes continuaremos hablando sobre el acceso a datos, centrándonos en cómo usar de forma más correcta algunas de las clases definidas en ADO.NET (varias de ellas exclusivas de la versión 2.0), sin olvidarnos de cómo minimizar los problemas cuando se produzcan errores gracias a la instrucción `using`. Asimismo, veremos cómo usar un objeto de tipo `DataReader` para leer datos de forma secuencial.

» **Al igual que hicimos** en el último taller, seguiremos escribiendo código; es decir, no vamos a echar mano de los asistentes de Visual Studio 2005, entre otras cosas porque nos vamos a centrar en cómo usar o mejorar el uso de algunas de las clases y sus métodos que ADO.NET pone a nuestra disposición. Y no solo nos basaremos en usar la "mejor" clase, sino que explicaremos cómo usar las clases que tengamos que usar, pero para sacarles un mayor rendimiento (o al menos para que no nos den un susto, por ejemplo, porque se produzca un error al utilizarla, ya que siempre es preferible que un problema nos "pille confesaos" a que nos tengamos que arrepentir de no haber estados prevenidos). Pero no solo veremos cómo acceder de forma "segura" a los datos, sino también qué clases podemos usar para mejorar el rendimiento e incluso trataremos de ver código que nos permita ejecutar de forma más fácil las tareas que habitualmente realizamos a la hora de acceder a los datos.

Comenzaremos viendo cómo preparar las cadenas de conexión y cómo obtener información sobre las instancias de SQL Server a las que tenemos acceso, para después entrar de lleno en cómo mejorar nuestro código o al menos intentar que sea menos propenso a "catástrofes". Por último, veremos cómo podemos usar un objeto de tipo `DataReader` para leer la información de una o varias series de datos

Guillermo "Guille" Som es Microsoft MVP de Visual Basic desde 1997. Es redactor de dotNetManía, miembro de *Ineta Speakers Bureau Latin America*, mentor de *Solid Quality Learning* Iberoamérica y autor del libro *Manual Imprescindible de Visual Basic .NET*.
http://www.elguille.info

Preparar la cadena de conexión

En cualquier aplicación de acceso a datos necesitaremos una cadena de conexión para conectar con la fuente de datos. Como ya vimos en números anteriores, esa cadena de conexión la podemos generar ayudados por los asistentes de Visual Studio 2005 o bien de forma manual. Pero entre las clases de ADO.NET 2.0 tenemos una que nos ayuda a crear esa cadena de conexión, a la que le indicaremos cada uno de los elementos que forman dicha cadena para que la genere por nosotros; estamos refiriéndonos a `DbConnectionStringBuilder`. La ventaja de esta clase –y particularmente de las derivadas de ella para cada uno de los proveedores de datos– es que nos resultará más "amigable" asignar la información, ya que dispone de propiedades del tipo adecuado según la información que queramos aportar. Por ejemplo, si queremos utilizar la seguridad integrada, asignaremos un valor verdadero a la propiedad `IntegratedSecurity` del objeto de tipo `SqlConnectionStringBuilder` instanciado para crear una cadena de conexión a una base de datos de SQL Server; de la misma forma, asignaremos el resto de datos, los cuales serán del tipo que corresponda según la información que tengamos que asignar. En el código del fuente 1 podemos ver cómo construir la cadena de conexión que hemos estado usando en los ejemplos de nuestro artículo anterior; como se puede apreciar, indicamos cada uno de los valores de una forma más amigable que creando la cadena como una ristra de parejas `nombre=valor`.

```
SqlConnectionStringBuilder csb = new
SqlConnectionStringBuilder();
csb.DataSource = @"(local)\SQLEXPRESS";
csb.InitialCatalog = "pruebaGuille";
csb.IntegratedSecurity = true;
```

Fuente 1. Crear una cadena de conexión usando la clase `SqlConnectionStringBuilder`

La cadena de conexión que posteriormente usaremos para conectar al servidor de base de datos la obtenemos por medio del método `ToString` o bien usando la propiedad `ConnectionString`. A esta propiedad también le podemos asignar una cadena de conexión que ya tengamos, y ella se encargará de dar a cada una de las demás propiedades de la clase los valores que correspondan, resultándonos mucho más fácil hacer luego cualquier cambio y volver a construir la cadena de conexión. Por ejemplo, en caso de que en lugar de la seguridad integrada de Windows queramos indicar un usuario y una contraseña.

Los proveedores de SQL Server disponibles

Otra de las clases que se han incorporado en .NET 2.0 al espacio de nombres `System.Data.Sql` es la clase `SqlDataSourceEnumerator`, por medio de la que podemos obtener una tabla con los proveedores accesibles en nuestra red local. Esta clase es una clase sellada (no heredable) de la que no podemos crear nuevas instancias, por lo que para crear un objeto de la clase debemos usar la propiedad estática `Instance`. A las instancias de SQL Server disponibles podremos acceder por medio del método `GetDataSources`, que devuelve un objeto del tipo `DataTable` con una fila por cada una de las instancias accesibles. Cada fila ofrece cuatro columnas que nos dan la información que necesitamos, principalmente el nombre del servidor (`ServerName`) y el nombre de la instancia (`InstanceName`), que puede ser una cadena vacía en el caso de la instancia predeterminada.

En el fuente 2 podemos ver cómo acceder a esas instancias y devolver un array de cadenas con cada nombre de servidor concatenado con el de la correspondiente instancia, valores que habitualmente usaremos para asignar al parámetro `DataSource` de una cadena de conexión.

```
private string[] instanciasSQL()
{
  List<string> instancias = new List<string>();

  SqlDataSourceEnumerator dse =
      SqlDataSourceEnumerator.Instance;
  DataTable dt = dse.GetDataSources();
  foreach( DataRow dr in dt.Rows )
  {
    string s = dr["ServerName"].ToString();
    if( !string.IsNullOrEmpty(
                  dr["InstanceName"].ToString()))
    {
      s += @"\" + dr["InstanceName"].ToString();
    }
    instancias.Add(s);
  }
  return instancias.ToArray();
}
```

Fuente 2. Acceder a las instancias de los servidores de SQL disponibles en nuestra red local

El problema que tiene esta clase es que, según indica la propia documentación de Visual Studio 2005, no es fiable al 100%, ya que no siempre garantiza que se devolverán todas las instancias de todos los servidores de SQL Server que haya accesibles en la red local. Incluso es posible que no siempre devuelva los mismos valores en llamadas consecutivas, además de que la obtención de esa información puede ser algo lenta.

Si lo que nos interesa es saber qué instancias hay disponibles en el equipo en el que se está ejecutando la aplicación, y el usuario tiene permisos para acceder al registro de Windows, podemos acceder a `InstalledInstances` de la clave `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Microsoft SQL Server`, que nos dirá qué instancias de SQL Server hay instaladas. La instancia predeterminada vendrá indicada como "MSSQLSERVER", que para uso normal podemos convertir en la cadena "(local)", tal como podemos ver en el código del fuente 3.

```
List<string> instancias = new List<string>();
string claveSQL=@"SOFTWARE\Microsoft\Microsoft SQL Server";
RegistryKey rk;
rk = Registry.LocalMachine.OpenSubKey(claveSQL,
false);
string[] instanciasRegistro;
instanciasRegistro =
      ((string[])rk.GetValue("InstalledInstances"));

foreach( string s in instanciasRegistro )
{
    if( s == "MSSQLSERVER" )
        instancias.Add("(local)");
    else
        instancias.Add(@"(local)\" + s);
}
return instancias.ToArray();
```

Fuente 3. Acceder a las instancias locales usando el registro

Debido a que es posible que el usuario no pueda acceder a esa información, ese código deberíamos incluirlo dentro de un bloque `try/catch`. En el código de ejemplo que acompaña a este artículo se tiene en cuenta esa posibilidad, y en caso de que falle el acceso al registro utilizamos el código mostrado en el fuente 2.

Al conectar, más vale prevenir...

Cuando nos conectamos al servidor de la base de datos debemos prevenir cualquier contratiempo. El primero y principal es que se produzca un error al realizar la conexión; por tanto, siempre que hagamos una conexión al servidor deberíamos incluir nuestro código de conexión dentro de un bloque `try/catch`. Lo habitual es que también tengamos un bloque `finally` que nos asegure que dicha conexión se cierre tanto si

se produce una excepción como si no se produce, ya que –como sabemos– el código que incluyamos dentro de un bloque `finally` siempre se ejecuta, haya o no error. En el fuente 4 podemos ver el código que deberíamos usar habitualmente.

```
SqlConnection cnn = null;
try
{
    cnn = new SqlConnection(cadenaConexion);
    cnn.Open();
    // resto del código
}
catch( Exception exception )
{
    MessageBox.Show("Error al conectar: " +
                    exception.Message);
}
finally
{
    if( cnn != null )
    {
        cnn.Close();
    }
}
```

Fuente 4. Incluir la conexión a la base de datos en un bloque `try`, sin olvidar cerrar la conexión en el bloque `finally`.

Mejor usando using

En este código, el punto importante es lo que hay en el bloque `finally`, pero C# (e incluso Visual Basic 2005) nos ofrece una alternativa, digamos, más elegante y en cierto modo más segura, que es usando la instrucción `using`. Esta instrucción deberá utilizarse indicando siempre un objeto que implemente la interfaz `IDisposable`; al implementar esa interfaz, las clases que podemos usar con la instrucción `using` incorporan un método llamado `Dispose`, que es el que se encarga de liberar los recursos no administrados que el objeto esté manejando. En el caso de la clase `SqlConnection`, ese recurso es la conexión a la base de datos; por tanto, al finalizar la ejecución del código que hayamos escrito dentro del bloque de la instrucción `using`, ésta se encargará de llamar a ese método, con lo cual nos aseguraremos de que si la conexión estaba abierta, se cerrará, liberando esos recursos. La instrucción `using` garantiza que si en el código que contiene se produce una excepción (incluso si no la tenemos controlada), podremos tener la tranquilidad de que el método `Dispose` será llamado. Por tanto, el código que antes hemos mostrado en el fuente 4 lo podemos cambiar por el que vemos en el fuente 5, y así nos aseguraremos de que, aunque nos olvidemos, la conexión se cerrará. Sí, seguramente no nos olvidaremos de cerrar esa conexión y, en el caso de no usar la instrucción `using`, no nos vamos a olvidar de incluir el código del bloque `finally`; pero como nunca está de más cualquier precaución, al usar el bloque `using` cualquier olvido nuestro no afectará a la conexión que tengamos abierta.

Por supuesto, el que utilicemos `using` no nos librará de que se produzcan excepciones, por tanto es una buena práctica incluir esa instrucción dentro de un bloque `try/catch`, aunque, como vemos en el fuente 5, ya no es necesario escribir el bloque `finally`, al menos si en ese bloque solo nos aseguramos de que se cierre la conexión.

```
SqlConnection cnn = null;

try
{
  using( cnn = new SqlConnection(cadenaConexion) )
  {
      cnn.Open();
      // resto del código
  }
}
catch( Exception ex )
{
  MessageBox.Show("Error al conectar: " + ex.Message);
}
```

Fuente 5. Utilizando un bloque `using` nos aseguramos de que la conexión no quedará abierta.

El uso de la instrucción `using` también es aplicable al resto de objetos que utilicen recursos no administrados por el propio *runtime* de .NET (CLR) y, en el caso de las bases de datos, sería igualmente válido usarlo con los objetos `SqlCommand` o cualquier otro objeto de acceso a datos que implemente `IDisposable`; si el objeto utiliza recursos, debemos liberarlos lo más pronto posible, y cuando usamos la instrucción `using`, esa liberación se realiza al finalizar el bloque de código.

Anidación de bloques using

Como hemos visto en el fuente 5, al utilizar la instrucción `using` indicamos el objeto (u objetos) que queremos controlar, y tal como hemos dicho, ese objeto tiene un ámbito o duración que termina cuando finaliza dicho bloque de código. La ventaja añadida es que si ese objeto no se crea (por ejemplo, en el código del fuente 5, porque la cadena de conexión no sea válida), el código que contiene el bloque no se ejecuta. Esto nos permite anidar varias instrucciones `using`, las cuales dependen de las anteriores para que todo funcione como es debido.

Por ejemplo, en el fuente 6 tenemos tres bloques `using` anidados. El primero es el que mantiene la conexión a la base de datos; el segundo depende de que ese objeto esté creado, ya que es un comando que traerá la información de la base de datos. Este bloque `using` a su vez contiene otro en el que creamos un objeto del tipo `SqlDataReader` que será el encargado de mostrar la información, en este caso, en una caja de texto.

```
StringBuilder sb = new StringBuilder();
SqlConnection cnn = null;
try
{
 using( cnn = new SqlConnection(CadenaConexion) )
 {
   cnn.Open();
   SqlCommand cmd = null;
   using(cmd=new SqlCommand("SELECT * FROM Prueba",cnn))
   {
     using( SqlDataReader reader=cmd.ExecuteReader())
     {
        while( reader.Read() )
        {
            sb.AppendFormat("{0}-{1}, {2}, {3}\r\n",
              reader[0], reader["Nombre"],
              reader["e-mail"], reader["FechaAlta"]);
        }
     }
   }
 }
 this.textBox1.Text = sb.ToString();
}
catch( Exception ex )
{
 MessageBox.Show("Error al conectar: " + ex.Message);
}
```

Fuente 6. Ejemplo de anidación de bloques `using`

¿Cómo funciona esta anidación de bloques using?

Si la conexión no se realiza correctamente, bien porque la cadena de conexión sea incorrecta, porque el servidor de SQL Server no esté disponible o por cualquier otra causa, el bloque de código completo no se ejecutará; por tanto, el resto de bloques `using` tampoco.

Si la conexión se realiza satisfactoriamente, se ejecutará el código que contiene el primer bloque, pasando a la apertura de la conexión (nuevamente, si ésta falla, se sale del bloque) y creando el comando con la selección indicada, el cual a su vez forma parte de otro bloque `using` que contiene como código un nuevo bloque `using` en el que se crea un `DataReader`. Como podemos suponer, cualquier error producido en cualquiera de estos bloques `using` hará que se cancele todo lo que reste por ejecutarse, pero con la certeza de que cada bloque `using` hará la llamada correspondiente al método `Dispose`, liberando los recursos que se estuvieran usando.

¿Realmente hacen falta tantos using?

Llegados a este punto, podríamos pensar que es una exageración utilizar tantos `using` cuando con bloques `try/catch/finally` conseguiríamos el mismo resultado. Y en parte podríamos pensar que así es, pero en realidad no es lo mismo. Es cierto que un bloque `using` se puede simular con un bloque `try/catch/finally` como el mostrado en el fuente 4, pero no debemos olvidar que en el fuente 6 estamos "controlando" tres objetos, no uno, por tanto, un código similar al del fuente 4 nos obligaría a usar tres bloques `try/catch/finally` o uno solo en el que tuviésemos que anidar comprobaciones de qué objetos están creados para no incurrir en una nueva excepción al intentar usar un método de uno que no ha llegado a instanciarse; además de que podemos obviar el uso del método `Close`, ya que el método `Dispose` cierra todo lo que hubiese abierto –al menos así es en los objetos de las clases que estamos usando–, particularmente en `SqlConnection` y `SqlDataReader`; esa misma funcionalidad debemos darla nosotros siempre que implementemos un método `Dispose`.

Ventajas y desventajas de usar un DataReader

Ya que en el código del fuente 6 hemos usado un objeto del tipo `SqlDataReader` para obtener los datos de ejemplo, veamos un poco cómo funciona esta clase y cuándo puede ser más eficiente su uso frente a un objeto de tipo `DataAdapter` con su correspondiente `DataTable` o `DataSet`.

Lo primero que debemos saber es que para usar un objeto `DataReader` (para SQL sería `SqlDataReader`, para OLE DB sería `OleDbDataReader`, etc.) debemos contar con una conexión abierta a la base de datos, y que el objeto `DataReader` no se puede instanciar, sino que lo obtenemos por medio del método `ExecuteReader` de un objeto `Command` (`SqlCommand`, `OldBCommand`, etc.), tal como vimos en el fuente 6. Esa conexión la tenemos que mantener abierta mientras estemos usando el "lector de datos" y cuando no necesitemos seguir usándolo debemos asegurarnos de que llamamos al método `Close` del `DataReader`; si tampoco necesitamos la conexión, debemos hacer también una llamada al método `Close` del objeto `Connection`.

Cuando usamos un `DataReader`, éste se "adueña" de la conexión y no la libera hasta que cerramos el lector; es decir, se comporta como un "exclusivista", tanto de la conexión como del comando que está usando. Por ejemplo, si el comando devuelve un valor, éste no se recibirá hasta que el `DataReader` se haya cerrado, y si necesitamos usar esa misma conexión para otros comandos, o incluso la estamos usando con otros objetos `Command`, éstos no harán su trabajo hasta que se cierre el `DataReader`.

Sigamos con las "pegas" de esta clase. Como acabamos de comentar, el

> **NOTA**
>
> En el código de ejemplo que acompaña al artículo hay una variante del código del fuente 6 en el que en lugar de usarse las clase `SqlConnection` y `SqlCommand` se utilizan otras clases personalizadas que juegan el mismo papel (o casi), pero desde las que se lanzan eventos cada vez que se llama al método `Dispose` que cada una de esas clases implementa. De esa forma podemos comprobar cómo se realiza esa "limpieza" se produzcan o no errores. En ese mismo código se indican las pruebas que podemos hacer para que se produzcan los errores, y así comprobar que todo funciona como es debido.

DataReader necesita una conexión abierta y activa. En realidad, este objeto no lee toda la información de una vez y la mantiene de forma local (como sería el caso de, por ejemplo, un DataTable), sino que obtiene una fila cada vez que llamamos al método Read; es decir, este método le pide a la base de datos que le proporcione la información de la fila actual, y así continuará para cada una de las filas que resulten de la ejecución del comando, avanzando cada vez a la siguiente hasta que ya no queden más datos, en cuyo caso el método Read devolverá el valor false, que podremos usar, como hacemos en el código del fuente 6, para saber que ya no quedan más datos que obtener.

Un objeto DataReader, como su nombre indica (aunque en inglés), solo lee datos, es decir, no lo podemos usar para realizar actualizaciones o cualquier otro proceso que no sea el de lectura. Además, esos datos solo los podemos leer secuencialmente en una dirección: avanzando del primero al último; es decir, no podemos movernos hacia atrás en los registros.

Con todo lo dicho hasta ahora podemos resumir que:

1. El objeto DataReader usa siempre una conexión a la base de datos, y ésta es exclusiva hasta que el lector se cierra.
2. Solo lo podemos usar para leer datos y solo avanzando en una dirección, sin posibilidad de volver a un registro anterior; cuando queremos avanzar a registros posteriores, no podremos hacerlo sin antes pasar por los que haya entre el actual y ellos.
3. Debemos cerrar el DataReader para poder seguir usando la misma conexión o los comandos relacionados con esa conexión.

Pero como es de esperar, no todo van a ser pegas. Veamos ahora algunas ventajas. A diferencia de un DataAdapter, el cual se conecta a la base de datos, trae toda la información que hemos solicitado y se desconecta, liberando la conexión, el DataReader mantiene la conexión abierta, pero con la ventaja de que no recupera toda la información y la guarda en la memoria de nuestro equipo. Esto es útil cuando el número de registros es muy grande, ya que no tenemos que esperar a que se recupere toda la información, acción que podría ocasionar dos problemas: el primero es que si los datos son muchos, la memoria ocupada podría suponer un inconveniente; el segundo es el tiempo que esa lectura podría acarrear.

Si vamos a acceder a una cantidad grande de datos y solo queremos navegar en una dirección (del primero al último), podemos decantarnos por usar un objeto del tipo DataReader

Con un mismo DataReader podemos indicar más de una selección de registros, aunque para poder usar el segundo bloque de datos, debemos leer el primero o bien forzar a pasar al segundo (o al siguiente), usando el método NextResult, que devolverá un valor verdadero si hay más datos, o falso si ya se leyeron todos los datos que había que leer. Para poder recuperar más de un grupo de datos, la cadena de selección debe contener cuantas sentencias necesitemos (no tienen por qué ser datos de la misma tabla ni con una misma estructura), separando cada una de ellas con un punto y coma, por ejemplo:

```
cmd.CommandText =
  "SELECT * FROM Prueba " +
  "WHERE FechaAlta > @Fecha1; " +
  "SELECT Nombre, [e-mail], FechaAlta " +
  "FROM Prueba WHERE FechaAlta < @Fecha2";
```

En estos casos (en los que pueden variar los nombres o el número de columnas que intervienen en cada grupo de resultados del DataReader) nos puede interesar saber cuántas columnas contiene cada fila o bien los nombres de las columnas que intervienen (ahora veremos cómo), ya que para acceder a los valores de cada columna de la fila actual tenemos que indicar el índice en base cero del campo o bien el nombre de la columna, tal como hicimos en el código del fuente 6:

```
reader[0], reader["Nombre"]
```

Además de usar el indizador (o la propiedad predeterminada Item en Visual Basic) del objeto DataReader para acceder a los valores, también podemos usar funciones específicas según el tipo de datos que vamos a recuperar: Por ejemplo, si sabemos que el contenido de la columna 1 es de tipo cadena, podemos usar el método GetString, al que le indicaremos el número de la columna:

```
reader.GetString(1)
```

Al usar esos métodos "especializados" solo podemos indicar el índice numérico de la columna a la que queremos acceder, es decir, no podemos indicar el nombre de la columna. Hay funciones de este tipo para prácticamente todos los tipos de datos predefinidos de .NET Framework y, en el caso de las clases SqlDataReader y OracleDataReader, también hay métodos que devuelven tipos definidos en SQL o en Oracle, por ejemplo, GetSqlBinary o GetOracleLob.

Como comentábamos, para saber los nombres de las columnas que intervienen u obtener otro tipo de información acerca de esas columnas, como el tipo de datos, el tamaño o si admiten nulos, podemos usar el método GetSchemaTable del objeto DataReader. La tabla que produce ese método contiene una fila por cada uno de los datos informativos acerca de cada columna; por ejemplo, el primer dato (fila con índice cero) nos informará del nombre de la columna, el segundo (índice 1), el índice de dicha columna, el tercero el tamaño, etcétera. Para comprender mejor esto, el fragmento de código del fuente 7 muestra cómo acceder a esos grupos de datos y a cada uno de los valores de las columnas que intervienen en el resultado devuelto.

```
using( cmd = new SqlCommand() )
{
  cmd.Connection = cnn;
  cmd.CommandText =
       "SELECT * FROM Prueba WHERE FechaAlta > @Fecha1;"+
       "SELECT Nombre, [e-mail], FechaAlta " +
       "FROM Prueba WHERE FechaAlta < @Fecha2";
  cmd.Parameters.AddWithValue("@Fecha1",
       DateTime.Parse("31/05/2006"));
  cmd.Parameters.AddWithValue("@Fecha2",
       DateTime.Parse("01/06/2006"));
  using( SqlDataReader reader = cmd.ExecuteReader() )
  {
    bool hayDatos = true;
    int n = -1;
    string[] grupos = cmd.CommandText.Split(';');
    while( hayDatos )
    {
      // El esquema de cada grupo de datos
      DataTable dt = reader.GetSchemaTable();
      n++;
      sb.AppendFormat("Resultado número {0}", n);
      sb.AppendLine();
      sb.Append( grupos[n].Replace(
          cmd.Parameters[n].ParameterName,
          cmd.Parameters[n].Value.ToString()).TrimStart());
      sb.AppendLine();
      while( reader.Read() )
      {
        // La primera columna de cada fila
        // tiene el nombre de la columna actual.
        foreach( DataRow dr in dt.Rows )
        {
            sb.AppendFormat("{0}: {1}\r\n",
                dr[0], reader[dr[0].ToString()]);
        }
        sb.AppendLine();
      }
      hayDatos = reader.NextResult();
    }
  }
}
```

Fuente 7. Acceso a la estructura de un `DataReader`

En el código mostrado en el fuente 7 podemos ver lo comentado anteriormente –en la propiedad `CommandText` del objeto `SqlCommand` asignamos más de una instrucción `SELECT`, creamos el `SqlDataReader` y recorremos cada uno de los resultados–. Cuando hemos terminado de leer los datos de uno de esos grupos de datos, utilizamos el método `NextResult` para saber si aún quedan datos en el `DataReader`; de ser así, mostramos esos datos usando las columnas indicadas en el esquema del resultado, esquema que conseguimos por medio del método `GetSchemaTable`. Cada columna que interviene nos la proporciona la colección `Rows` de la tabla obtenida; como el indizador del objeto `DataReader` espera un nombre de columna o un índice, debemos usar `dr[0].ToString()` para saber el nombre. Sin embargo, para añadir ese nombre de columna al objeto `StringBuilder` no es necesario usar el método `ToString`, ya que dicho método es utilizado de forma predeterminada por el método `AppendFormat` del `StringBuilder`. Sin embargo, el indizador del `DataReader` no es tan flexible y nos obliga a indicar el nombre o el índice numérico de la columna; en este último caso tendríamos que usar el valor de la columna con índice 1 de la fila del esquema:

```
sb.AppendFormat("{0}: {1}\r\n",
      dr[0], reader[(int)dr[1]]);
```

Ya para concluir, comentaremos que el acceso por índice numérico a las columnas del `DataReader` es más rápido que el acceso por nombre; seguramente unos pocos ticks de reloj, pero más rápido porque el índice numérico va "a tiro fijo" (siempre que ese valor esté dentro del rango de índices permitidos); entre otras cosas, porque no es necesario comprobar si el nombre es correcto o no.

Resumiendo un poco y sin querer ser concluyente, ya que el mejor resultado es el que a la larga nos resulte más cómodo de usar (independientemente de lo que nos cuenten, porque a favor o en contra del uso de `DataReader`, como en casi todo, hay opiniones muy variadas), si vamos a acceder a una cantidad grande de datos y solo queremos navegar en una dirección (del primero al último), podemos decantarnos por usar un objeto del tipo `DataReader`. Pero si lo que nos interesa es obtener una serie de datos (aunque solo sea para lectura) y queremos hacer selecciones sobre los datos obtenidos, posiblemente nos interese más usar un `DataAdapter` y rellenar un `DataTable` con esos datos, a los que podremos aplicar luego el método `Select` para obtener las filas que cumplan la condición que indiquemos. Si no nos hace falta una conexión (u otros comandos) para realizar otras tareas mientras estamos accediendo a los datos, podemos usar un `DataReader`, pero si después de conseguir los datos queremos dejarlos a un lado para procesar otras cosas, podemos usar un adaptador. Si sabemos que el usuario no se irá a tomar café desde que pida los datos con el método `ExecuteReader`, podemos usar el `DataReader`; pero si es posible que la conexión se quede abierta por más tiempo del deseado, posiblemente nos interese más solicitar los datos con el objeto `DataAdapter`. Así podríamos seguir con más supuestos, pero como siempre suele suceder, al final tendremos que decidir nosotros solos qué es lo que más nos conviene.

Conclusiones

Confiamos que todo lo aquí comentado sirva como ayuda para cuando tengamos que decidir qué usar, y si ahora sabemos un poco más en ese sentido, pues el objetivo está conseguido. No nos referimos solo al `DataReader`, sino también al resto de temas que hemos tratado, y en particular al relacionado con los bloques de código `using`.

Como viene siendo habitual en estos artículos de *dnm.inicio*, en el ZIP con el código de ejemplo se ofrece tanto el código para C# como para Visual Basic 2005, donde, como hemos comentado, también es posible usar la instrucción `Using`.