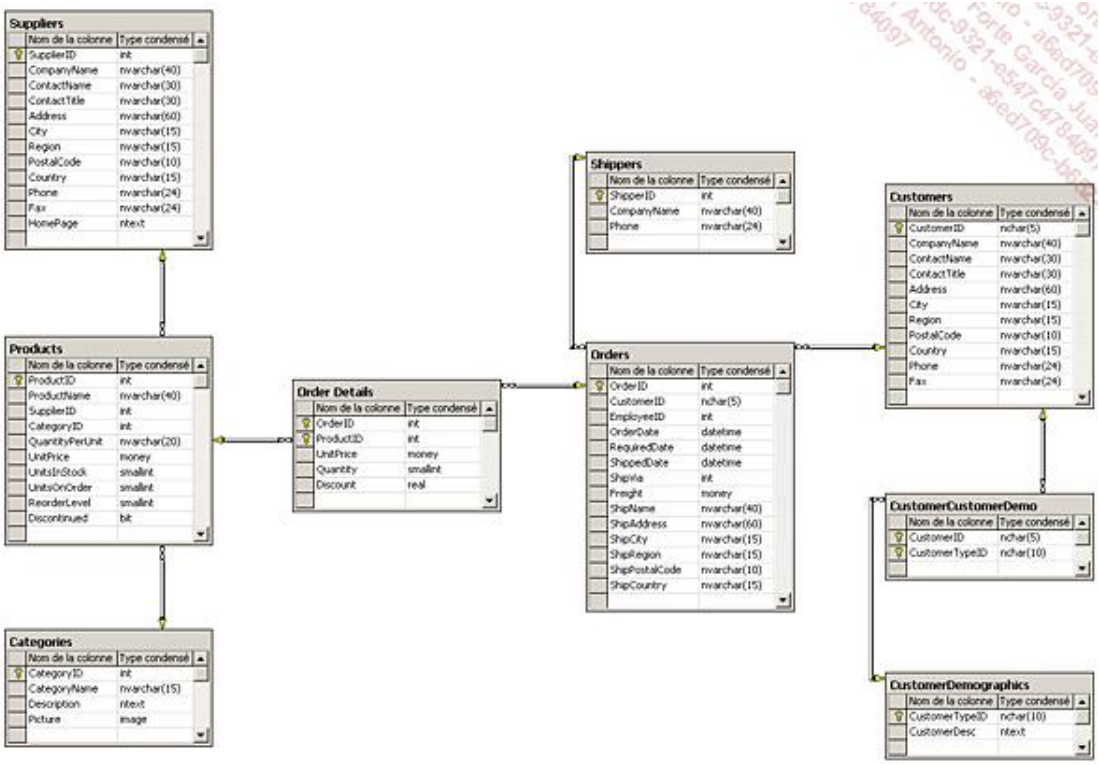
Utilización del modo conectado

En este capítulo, vamos a tratar las operaciones que pueden ejecutarse en una base de datos usando el modo conectado. Ciertas nociones estudiadas en este capítulo también serán útiles para el funcionamiento en modo desconectado. Para probar las diferentes funcionalidades estudiadas en este capítulo, utilizaremos un servidor SQL Server 2008. La base de datos usada será la base Northwind, que se puede instalar gracias al script instrund.sql disponible en los archivos que se descargan. El siguiente esquema muestra una parte de la estructura de la base.



1. Conexión a una base

Para poder trabajar con un servidor de base de datos, una aplicación debe establecer una conexión de red con el servidor. La clase `SqlConnection` es capaz de gestionar una conexión hacia un servidor SQL Server versión 7.0 o posterior. Como para cualquier objeto, en primer lugar debemos declarar una variable.

```
SqlConnection ctn;
```

Luego, debemos crear la instancia de la clase e inicializarla llamando a un constructor. La inicialización va a consistir esencialmente en indicar los parámetros utilizados para establecer la conexión con el servidor. Estos parámetros se definen en forma de cadena de caracteres. Pueden ser indicados durante la llamada del constructor o modificados luego por la propiedad `ConnectionString`.

a. Cadena de conexión

El formato estándar de una cadena de conexión está constituido por una serie de pares clave/valor separados con punto y coma. El signo = se usa para la asignación de un valor a una palabra clave. El análisis de la cadena se efectúa durante la asignación de la cadena a la propiedad `ConnectionString`. A continuación se extraen los valores asociados a las palabras clave y se asignan las diferentes propiedades de la conexión. Si se encuentra un error de sintaxis, entonces se genera una excepción inmediatamente y no se modifica ninguna propiedad. Por el contrario, sólo se pondrán controlar algunas propiedades durante la apertura de la conexión. Sólo en este momento se activará una excepción si la cadena de conexión contiene un error. Únicamente se puede modificar la cadena de conexión si la conexión está cerrada. Las siguientes palabras están disponibles para una cadena de conexión:

Connect Timeout

Tiempo en segundos durante el cual la aplicación esperará una respuesta del servidor a su petición de conexión. Pasado este plazo, se activa una excepción.

Data Source

Nombre o dirección de red del servidor hacia el cual se establece la conexión. El número del puerto se puede especificar después del nombre o de la dirección de red. Si no está indicado, el número de puerto es igual a 1433.

Initial Catalog

Nombre de la base de datos a la que se desea conectar.

Integrated Security

Si este valor es **false**, entonces se debe facilitar un nombre de usuario y una contraseña en la cadena de conexión. Si es **true**, la cuenta local Windows del usuario se usará como autentificación.

Persist Security Info

Si se coloca este valor en **true**, entonces el nombre del usuario y su contraseña serán accesibles por la conexión. Por razones de seguridad, se debe colocar este valor en **false**. De hecho, es el caso si usted no indica nada en su cadena de conexión.

Pwd

Contraseña asociada a la cuenta SQL Server utilizada para la conexión. Si no existe contraseña asociada a una cuenta, se puede omitir esta información en la cadena de conexión.

User ID

Nombre de la cuenta SQL Server utilizada para la conexión.

Connection LifeTime

Indica la duración de vida de una conexión en un pool de conexiones. Un valor igual a cero indica una duración ilimitada.

Connection Reset

Indica si la conexión se reinicializa al ser devuelta al pool.

Max Pool Size

Número máximo de conexiones en el pool.

Min Pool Size

Número mínimo de conexiones en el pool.

Pooling

Indica si es posible extraer una conexión de un pool de conexiones.

Una cadena de conexión toma así la forma mínima siguiente:

```
ctn.ConnectionString= "Data Source=localhost;Initial Catalog=Northwind;
Integrated Security=true";
```

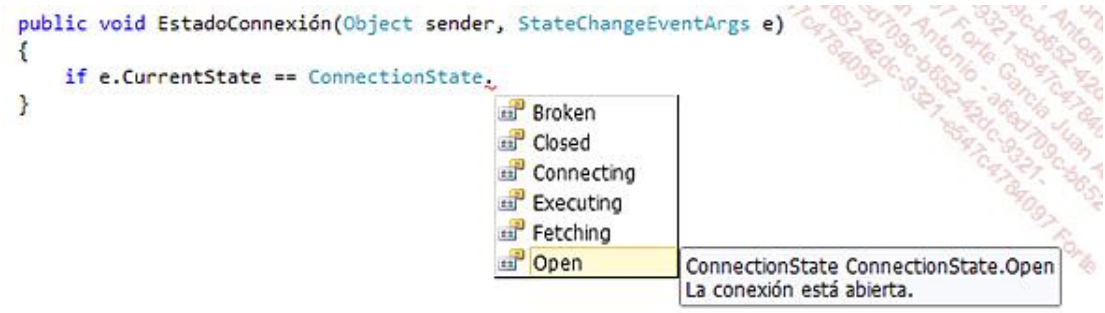
b. Pool de conexiones

Los pools de conexiones permiten mejorar las prestaciones de una aplicación evitando la creación de conexiones adicionales. Cuando una conexión está abierta, se crea un pool de conexiones utilizando un algoritmo basado en la cadena de conexión. Así, cada pool está asociado a una cadena de conexión particular. Si se abre una nueva conexión y no existe pool que corresponda exactamente a su cadena de conexión, entonces se crea un nuevo pool. Los pools de conexiones así creados existirán hasta el final de la aplicación. Durante la creación del pool, otras conexiones pueden crearse automáticamente para satisfacer el valor `Min Pool Size` indicado en la cadena de conexión. Se podrán añadir otras conexiones al pool hasta alcanzar el valor `Max Pool Size` de la cadena de conexión. Cuando se requiere una conexión, ésta se puede obtener desde un pool de conexión (si existe uno que corresponda exactamente a las características de la conexión requerida). Por supuesto, hace falta que el pool contenga una disponible y activa.

Si se alcanza el número máximo de conexión en el pool, la petición se colocará en cola hasta que haya una conexión libre. Se devuelve una conexión al pool tras su cierre o invocando el método `Dispose` sobre la conexión. Por esta razón, se recomienda cerrar explícitamente las conexiones cuando ya no se utilizan en la aplicación. Se retira una conexión del pool cuando el sistema detecta que no hay más conexiones desde hace un cierto tiempo, indicado por el valor `ConnectionLifeTime` de la cadena de conexión. También se retira del pool si detecta que la conexión con el servidor se ha interrumpido.

c. Eventos de conexión

La clase `SQLConnection` propone dos eventos que le permiten recibir una advertencia cuando el estado de la conexión cambia o cuando un mensaje de información se envía a través del servidor. El evento `StateChanged` se activa durante un cambio de estado de la conexión. El gestor de este evento recibe un parámetro de tipo `StateChangeEventArg`, que permite obtener, con la propiedad `CurrentState`, el estado actual de la conexión, y con la propiedad `OriginalState`, el estado de la conexión antes de la desactivación del evento. Para probar el valor de estas dos propiedades, puede utilizar la enumeración `ConnectionState`.



El evento `InfoMessage` se activa cuando el servidor le informa de una situación anormal, pero que no justifica la activación de una excepción (gravedad del mensaje inferior a 10). El gestor de eventos asociado recibe un parámetro de tipo `InfoMessageEventArgs`. Por la propiedad `Errors` de este parámetro, tiene acceso a objetos `SqlErrors` que corresponden a la información enviada por el servidor. El siguiente código muestra en la consola los mensajes de información que provienen del servidor.

```
public static void ctn_InfoMessage(object sender, System.Data.SqlClient.SqlInfo
MessageEventArgs e)
    {
         foreach ( SqlError info in e.Errors)
         {
            Console.WriteLine(info.Message);
```

```
        }
    }
```

2. Ejecución de un comando

Después de haber establecido una conexión hacia un servidor de base de datos, usted le puede transmitir instrucciones SQL. Se utiliza la clase `SqlCommand` para pedir al servidor la ejecución de un comando SQL. Esta clase contiene varios métodos que permiten la ejecución de diferentes tipos de consultas SQL. Se puede instanciar la clase `SqlCommand` de manera clásica usando uno de sus constructores o se puede obtener una instancia con el método `CreateCommand` de la conexión.

a. Creación de un comando

La primera posibilidad para crear un `SqlCommand` consiste en utilizar uno de los constructores de la clase. La utilización del constructor por defecto le obliga a utilizar luego diferentes propiedades para facilitar la información relativa a la instrucción SQL que se ha de ejecutar.

La propiedad `CommandText` contiene el texto de la instrucción SQL que se ha de ejecutar. La propiedad `Connection` debe hacer referencia a una conexión válida hacia el servidor de base de datos. El siguiente código resume estas operaciones:

```
SqlCommand cmd;
cmd = new SqlCommand();
cmd.Connection = ctn;
cmd.CommandText = "select * from products";
```

La segunda solución consiste en utilizar un constructor sobrecargado aceptando como parámetros la instrucción SQL con la forma de una cadena de caracteres y la conexión utilizada por esta `SqlCommand`. Así se puede resumir el código anterior a la siguiente fila:

```
cmd = new SqlCommand("select * from products", ctn);
```

La tercera solución consiste utilizar el método `CreateCommand` de la conexión. En este caso, sólo se tiene que especificar a continuación la instrucción SQL con la propiedad `CommandText`.

```
SqlCommand cmd;
cmd = ctn.CreateCommand();
cmd.CommandText = "select * from products";
```

b. Lectura de información

A menudo, la instrucción SQL de un `SqlCommand` selecciona un conjunto de registros en la base, o eventualmente un valor único que es resultado de un cálculo efectuado sobre valores contenidos en la base. Una instrucción SQL, que devuelve un conjunto de registros, debe ser ejecutada por el método `ExecuteReader`. Este método devuelve un objeto `DataReader` que va a permitir luego la lectura de la información que proviene de la base de datos. Si la instrucción sólo devuelve un valor único, el método `ExecuteScalar` se encarga de la ejecución y devuelve él mismo el valor que proviene de la base de datos.

El siguiente código permite la recuperación del número de pedidos que ha hecho un cliente:

```
SqlConnection ctn;
SqlCommand cmd;
ctn = new SqlConnection();
ctn = new SqlConnection("Data Source=localhost;Initial
```

```
                        Catalog=Northwind;Integrated Security=True");
ctn.Open();
cmd = ctn.CreateCommand();
cmd.CommandText = "select count(orderid) from orders where
                   customerid='FRANK'";
Console.WriteLine("el cliente ALFREDO ha pasado {0} pedido(s)",
cmd.ExecuteScalar());
```

El caso de instrucciones que devuelven varios registros es un poco más complejo. Después de haber ejecutado la instrucción con el método `ExecuteReader` y recuperado el objeto `DataReader`, puede utilizar este último para recorrer los resultados devueltos. El método `Read` de la clase `DataReader` permite el desplazamieto en el conjunto de los registros devueltos. Este método devuelve un booleano que indica si queda aún un registro. El desplazamiento sólo es posible desde el primero al último registro. Este tipo de desplazamiento se llama `Forward Only`. La información contenida en el registro corriente está accesible por uno de los métodos `Get...` de la clase `DataReader`. Estos métodos permiten extraer los datos del registro y convertirlos en un tipo de datos .NET. Existe una versión para cada tipo de datos del Framework .NET. Por supuesto, hace falta que la información presente en el registro se pueda convertir en el tipo correspondiente. Si la conversión es imposible, se activa una excepción. Los métodos `Get...` esperan como parámetro el número del campo a partir de la cual se recupera la información. Por defecto, también puede utilizar la propiedad del `DataReader` indicando el nombre del campo en cuestión. En este caso, no hay conversión y el valor devuelto es de tipo `Object`.

El siguiente código visualiza la lista de todas las categorías de productos disponibles:

```
public static void testDataReader()
{
    SqlCommand cmd;
    SqlConnection ctn;
    SqlDataReader lector;
          ctn = new SqlConnection();
          ctn.ConnectionString = "Data Source=localhost;Initial
                                 Catalog=Northwind;Integrated Security=true";
          ctn.Open();
          cmd = new SqlCommand();
          cmd.Connection = ctn;
          cmd.CommandText = " select * from categories";
          lector = cmd.ExecuteReader();
          while (lector.Read())
          {
               Console.WriteLine("numero de la categoria:{0}" + "\t" +
"Nombre:{1}", lector.GetInt32(0),    lector["CategoryName"]);
           }
           lector.Close();
           ctn.Close();
 }
```

La utilización de una conexión por un `DataReader` se efectúa de manera exclusiva. Para que la conexión esté de nuevo disponible para otro comando, debe cerrar obligatoriamente `DataReader` después de su utilización.

c. Modificaciones de la información

La modificación de la información en una base de datos se efectúa principalmente con las instrucciones `SQL INSERT`, `UPDATE`, `DELETE`. Estas instrucciones no devuelven registros de la base de datos. Para utilizar estas instrucciones, debe crear un `SqlCommand` y luego pedir la ejecución de este pedido a través del método `ExecuteNonQuery`. Este método devuelve el número de registros a los que afecta la ejecución de la instrucción SQL contenida en el `SqlCommand`. Si la propiedad `CommandText` contiene varias instrucciones SQL, entonces el valor devuelto por el método `ExecuteNonQuery`

corresponde al número total de filas a las que afectan todas las instrucciones SQL del `SqlCommand`.

El siguiente código añade una nueva empresa de entrega en la tabla **Shippers**:

```
public static void  TestExecuteNonQuery()
    {
        SqlCommand cmd;
        SqlConnection ctn;
        ctn = new SqlConnection();
        ctn.ConnectionString = "Data Source=localhost;Initial
                               Catalog=Northwind;Integrated Security=true";
            ctn.Open();
            cmd = new SqlCommand();
            cmd.Connection = ctn;
            cmd.CommandText = "Insert into shippers (companyname,phone)
                               values ('DHL','02 40 41 42 43')";
            Console.WriteLine("{0} linea(s) añadida(s) en la tabla",
                              cmd.ExecuteNonQuery());
            ctn.Close();
    }
```

d. Utilización de parámetros

El manejo de instrucciones SQL puede resultar más fácil si se crean parámetros. Éstos permiten construir instrucciones SQL genéricas que se pueden reutilizar fácilmente. El principio de funcionamiento es similar al de los procedimientos y funciones de Visual C#. Una alternativa a la utilización de parámetros podría ser la construcción dinámica de una instrucción SQL por concatenación de cadenas de caracteres.

A continuación, un ejemplo que utiliza esta técnica y que permite la búsqueda de un cliente según su código (luego veremos cómo mejorar este código usando parámetros):

```
public static void TestRequeteConcat()
    {
        SqlCommand cmd;
        SqlConnection ctn;
        SqlDataReader lector;
        string codigoCliente;
            ctn = new SqlConnection();
            ctn.ConnectionString = "Data Source=localhost;Initial
                      Catalog=Northwind;Integrated Security=true";
            ctn.Open();
            cmd = new SqlCommand();
            cmd.Connection = ctn;
            Console.Write("introducir el código del cliente buscado:");
            codigoCliente = Console.ReadLine();
            cmd.CommandText = " SELECT * from Customers WHERE CustomerID =
                              '" + codigoCliente + "'";
            lector = cmd.ExecuteReader();
            while (lector.Read())
            {
                Console.WriteLine("apellido del cliente:{0}",
                lector["ContactName"]);
            }
            lector.Close();
            ctn.Close();
```

```
        }
    }
```

La parte importante de este código corresponde al momento en el que se asigna un valor a la propiedad `CommandText`. Una instrucción SQL correcta debe construirse por concatenación de cadenas de caracteres. En nuestro caso, es relativamente simple, ya que sólo hay un valor variable en la instrucción SQL, pero si fueran varios, hay una multitud de concatenaciones que realizar. Los errores clásicos en estas concatenaciones son:

- el olvido de un espacio,
- el olvido de los caracteres ` ` para enmarcar un valor de tipo cadena de caracteres,
- un número de caracteres ` impar.

Todos estos errores tienen el mismo efecto: la creación de una instrucción SQL no válida que será rechazada por el servidor durante la ejecución.

La utilización de los parámetros simplifica considerablemente la escritura de este tipo de consulta. Se utilizan los parámetros para marcar una ubicación en una consulta donde estará colocado, en el momento de la ejecución, un valor literal de cadena de caracteres o numérico. Los parametros pueden ser nominados o anónimos. Un parámetro anónimo es introducido en una consulta por el carácter ?. Los parámetros nombrados se especifican mediante el carácter @ seguido del parámetro.

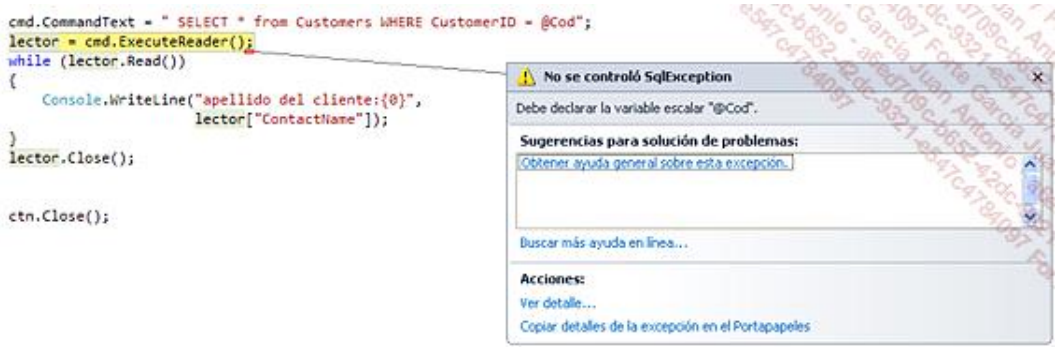
La consulta de nuestro ejemplo anterior puede tomar las siguientes formas:

```
cmd.CommandText = " SELECT * from Customers WHERE CustomerID = ?";
```

o

```
cmd.CommandText = " SELECT * from Customers WHERE CustomerID = @Code”;
```

La ejecución del `SqlCommand` fracasa ahora si no se facilita información alguna para el parámetro o los parámetros.



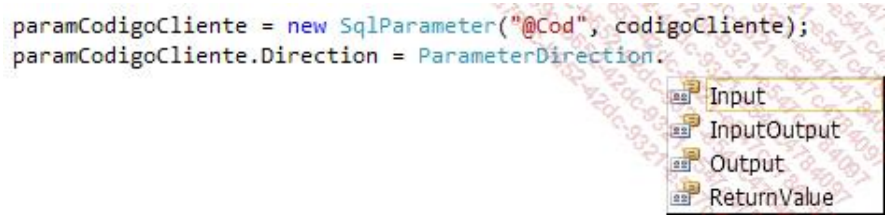
El `SqlCommand` debe tener una lista de valores utilizados para reemplazar los parámetros en el momento de la ejecución. Se almacena esta lista en la colección **Parameters** del `SqlCommand`. Antes de la ejecución del `SqlCommand`, hace falta crear los objetos `SqlParameter` y añadirlos a la colección. Para cada `SqlParameter`, hay que proveer:

- el nombre del parámetro;
- el valor del parámetro;
- la dirección de utilización del parámetro.

Se indica las dos primeras informaciones durante la construcción del objeto:

```
SqlParameter paramCodigoCliente;
paramCodigoCliente = new SqlParameter("@code", codigoCliente);
```

La dirección de utilización indica si la información contenida en el parámetro se pasa al código SQL para su ejecución (`Input`) o si le corresponde a la ejecución del código SQL modificar el valor del parámetro (`Output`), o ambas cosas (`InputOutput`). La propiedad `Direction` de la clase `SqlParameter` indica el modo de utilización del parámetro.



El parámetro está ahora preparado para añadirse a la colección **Parameters**. A este nivel, conviene estar al tanto si la consulta utiliza parámetros anónimos, ya que se deben añadir a la colección dichos parámetros obligatoriamente en el orden de su aparición en la consulta. Si se utilizan los parámetros nominados, no es necesario respetar esta regla, pero es prudente atenerse a ella, por si un día el código SQL se modifica y deja de utilizar los parámetros nominados. Podría ser el caso si usted debe cambiar de tipo de proveedor de datos y el nuevo no acepta los parámetros nominados en una instrucción SQL. Ahora el `SqlCommand` está listo para la ejecución. Hay que observar que, con esta solución, no tenemos que preocuparnos del tipo de valor esperado por la instrucción SQL para saber si debemos enmarcarlo con caracteres `'`. Si se usan parámetros en la salida de la instrucción SQL, sólo estarán disponibles después del cierre del `DataReader`. El siguiente ejemplo muestra, además del nombre del cliente, el número de pedidos ya pasados:

```
public static void TestRequeteParam()
    {
        SqlCommand cmd;
        SqlConnection ctn;
        SqlDataReader lector;
        string codigoCliente;
        SqlParameter paramCodigoCliente;
        SqlParameter paramNumPedidos;
            ctn = new SqlConnection();
            ctn.ConnectionString = "Data Source=localhost;Initial
                      Catalog=Northwind;Integrated Security=true";
            ctn.Open();
            cmd = new SqlCommand();
            cmd.Connection = ctn;
            Console.Write("introducir el código del cliente buscado:");
            codigoCliente = Console.ReadLine();
            cmd.CommandText = " SELECT * from Customers WHERE CustomerID =
                            @Code;select @nbCmd=count(orderid) from
                            orders where customerid=@code";
            paramCodigoCliente = new SqlParameter("@Code", codigoCliente);
            paramCodigoCliente.Direction = ParameterDirection.Input;
            cmd.Parameters.Add(paramCodigoCliente);
            paramNumPedidos = new SqlParameter("@nbCmd",null);
            paramNumPedidos.Direccion = ParameterDirection.Output;
            paramNumPedidos.SqlDbType=SqlDbType.Int;
            cmd.Parameters.Add(paramNumPedidos);
            lector = cmd.ExecuteReader();
            while (lector.Read())
            {
                Console.WriteLine("apellido del cliente:{0}",
                                lector["ContactName"]);
            }
```

```
            lector.Close();
            Console.WriteLine("este cliente ha pasado {0} pedido(s)",
                              cmd.Parameters["@nbCmd"].Value);
            ctn.Close();
    }
```

e. Ejecución de procedimientos almacenados

Los procedimientos almacenados son componentes de una base de datos que corresponden a un conjunto de instrucciones SQL, los cuales pueden ejecutarse simplemente invocando su nombre. Son verdaderos programas SQL que pueden recibir parámetros y devolver valores. Además, los procedimientos almacenados se registran en la memoria caché del servidor en forma compilada durante su primera ejecución, lo que aumenta las prestaciones para las ejecuciones siguientes. Otra ventaja de los procedimientos almacenados es que centralizan en el servidor de base de datos todo el código SQL de una aplicación. Si se deben aportar modificaciones en las instrucciones SQL, sólo tendrá que efectuar modificaciones en el servidor, sin necesidad de retomar el código de la aplicación ni tener que volver a generar y desplegar la aplicación.

La llamada a un procedimiento almacenado a partir de Visual C# es prácticamente similar a la ejecución de una instrucción SQL. La propiedad `CommandText` contiene el nombre del procedimiento almacenado. También puede modificar la propiedad `CommandType` con el valor `CommandType.StoredProcedure` para indicar que la propiedad `CommandText` contiene el nombre de un procedimiento almacenado. Como para una instrucción SQL, un procedimiento almacenado puede utilizar parámetros de entrada o salida. Hay un tercer tipo de parámetro disponible para los procedimientos almacenados en el tipo `ReturnValue`. Este tipo de parámetro sirve para recuperar el valor devuelto por la instrucción `Return` del procedimiento almacenado (mismo principio que una función Visual C#). Para probar estas nuevas nociones, vamos a utilizar el procedimiento almacenado siguiente, que devuelve el importe total de todos los pedidos hechos por un cliente.

```
CREATE PROCEDURE TotalCliente  @code nchar(5) AS
declare @total money
select @total=sum(UnitPrice*Quantity*(1-Discount)) from Orders,[Order Details]
where customerid=@code and Orders.orderid=[order details].orderid
return @total
GO
```

A nivel de código Visual C#, debemos indicar que se trata de la ejecución de un procedimiento almacenado y añadir un parámetro para recuperar el valor de retorno del procedimiento almacenado. Este parámetro debe llamarse `RETURN_VALUE`.

```
public static void TestProcedimientoAlmacenado()
        {
            SqlCommand cmd;
            SqlConnection ctn;
            SqlParameter paramCodigoCliente;
            SqlParameter paramImporte;
            string codigoCliente;
            Console.Write("introducir el código del cliente buscado:");
            codigoCliente = Console.ReadLine();
            ctn = new SqlConnection();
            ctn.ConnectionString = "Data Source=localhost;Initial
                      Catalog=Northwind;Integrated Security=true";
            ctn.Open();
            cmd = new SqlCommand();
            cmd.Connection = ctn;
            cmd.CommandText = "TotalCliente";
            cmd.CommandType = CommandType.StoredProcedimiento;
```

```
            paramCodigoCliente = new SqlParameter("@Code", codigoCliente);
            paramCodigoCliente.Direccion = ParameterDirection.Input;
            cmd.Parameters.Add(paramCodigoCliente);
            paramImporte = new SqlParameter("RETURN_VALUE",
                                            SqlDbType.Decimal);
            paramImporte.Direccion = ParameterDirection.ReturnValue;
            cmd.Parameters.Add(paramImporte);
            cmd.ExecuteNonQuery();
            Console.WriteLine("Este cliente ha efectuado pedidos por importe
                              de {0} Euros", paramImporte.Value);
            ctn.Close();
        }
```