12 Programación Funcional

La interface `IComparer` (Listado 12-1) presentada en el Capítulo 11 consiste de un solo método que recibe dos parámetros `x` y `y` de un mismo tipo `T` y "los compara", devolviendo un valor `int` que se supone cumple con el convenio de que si `x` es menor que `y` se devuelve un valor menor que `0`, si `x` es mayor que `y` se devuelve un valor mayor que `0` y si son iguales se devuelve `0`.

```
public interface IComparer<in T>
{
  int Compare(T x, T y);
}
```

Listado 12-1 Interface IComparer

Gracias a esta interface se puede definir de manera genérica un método de ordenación (Listado 12-2) sobre la base de que se le pase un objeto instancia de una clase que implemente la interface `IComparer`.

```
static void Ordenar<T>(IList<T> items, IComparer<T> comp)
{
  for (int k = 0; k < items.Length - 1; k++)
    for (int j = k + 1; j < items.Length; j++)
      if (comp.Compare(items[j],items[k]) < 0)
      {
        T temp = items[j];
        items[j] = items[k];
        items[k] = temp;
      }
}
```

Listado 12-2 Método Ordenar usando un IComparer

De este modo, para ordenar una lista de elementos de tipo `T` podemos darle a un método de ordenación diferentes objetos comparadores, con lo que logramos el efecto de ordenar la lista por diferentes criterios de comparación. Así para un tipo `Email` como el del Listado 12-3 se pueden tener diferentes implementaciones de `IComparer<Email>`, es decir de "objetos comparadores de `Email`" (Listado 12-4)

```
class Email
{
  public string Sender { get; set; }
  public string Subject { get; set; }
  public long Size { get; set; }
  public DateTime Received { get; set; }
  public DateTime Sent { get; set; }
  //...
}
```

Listado 12-3 Implementación de un tipo Email

```
class EmailSenderComparer: IComparer<Email>
{
  //...
}

class EmailSubjectComparer : IComparer<Email>
{
  //...
}

class EmailSizeComparer : IComparer<Email>
{
  //...
}

class EmailReceivedComparer : IComparer<Email>
{
  //...
}

class EmailSentComparer : IComparer<Email>
{
  //...
}
```

Listado 12-4 Implementación de un tipo Email

Como se dice en el Capítulo 11 (Interfaces), parece excesivo tener que definir toda una clase para expresar solo un criterio de comparación, clase de la que además solo se va a crear una instancia.

Vamos a ver la posibilidad de darle un mayor rango o categoría al criterio de método (función) para asignarlos como valor y pasarlos como parámetros al mismo nivel que hacemos con los objetos. De esta manera, a un método como el de `Ordenar` en lugar de pasarle un objeto comparador instancia del tipo `IComparer` le podremos pasar directamente el método como tal.

C# dispone de un recurso para caracterizar este concepto de "tipo método" del mismo modo que como lenguaje orientado a objetos dispone de recurso (las clases e interfaces) para definir tipos de objetos. Este recurso son los **delegados** (*delegates*).

12.1 Delegados

Si una clase define un "tipo de objeto", un **delegado** define un "tipo de método". Para un caso como el del ejemplo anterior de comparador, un delegado debe caracterizar a un tipo de método que recibe dos parámetros de un mismo tipo y devuelve un entero (`int`). Un tal tipo, llamémosle `Compare`, se puede definir entonces de la forma

```
public delegate int Compare<T>(T x, T y);
```

Una nueva palabra clave `delegate` caracteriza a esta forma de definición. Note que también se ha aplicado aquí la genericidad. Si queremos caracterizar a un método que compara `Email`, entonces se puede definir como de tipo `Compare<Email>`, que caracteriza a métodos que comparan dos emails. Un método como el `Ordenar` del Listado 12-2 puede ahora programarse como se muestra en el Listado 12-5

```
static void Ordenar<T>(IList<T> items, Compare<T> compara)
{
  for (int k = 0; k < items.Length - 1; k++)
    for (int j = k + 1; j < items.Length; j++)
      if (compara(items[j],items[k]) < 0)
      {
        T temp = items[j];
        items[j] = items[k];
        items[k] = temp;
      }
}
```

Listado 12-5 Método Ordenar usando un delegado Compare

Fíjese que en el Listado 12-2 se comparaba haciendo `comp.Compare(items[j],items[k])`, es decir, se le pedía al objeto `comp` instancia de `IComparer` que aplicase el método `Compare` que como instancia de `IComparer` debía tener. Ahora en el Listado 12-5 se aplica directamente el método al hacer `compara(items[j],items[k])`, es decir, el valor del parámetro `compara` es propiamente un método de comparación y por tanto se aplica usando la misma sintaxis `compara(…parámetros…)` que la que se usa para invocar a un método.

12.1.1 Creación de instancias de delegados

Ya tenemos cómo definir tipos delegados. Para crear una "instancia" del tipo delegado hay que aportar entonces un método que cumpla con las exigencias del tipo del delegado (es decir, con los mismos tipos de los parámetros y tipo de retorno). Similarmente a la creación de una instancia de una clase en la que se escribía `new NombreDeLaClase(…)`, una instancia de delegado se crea haciendo `new NombreDelTipoDelegado(…)`, en este caso dando como parámetro el nombre de un método que corresponda a la signatura definida por el delegado.

Si se consideran los métodos definidos en el Listado 12-6, entonces si `emailList` es una lista de emails y `Ordenar` es el método del Listado 12-5 podemos ordenar la lista con un código como el que se muestra en el Listado 12-7.

```
public static class Utils
{
  public static int CompareSender(Email x, Email y)
  {
    return x.Sender.CompareTo(y.Sender);
  }
  public static int CompareSubject(Email x, Email y)
  {
    return x.Subject.CompareTo(y.Subject);
  }
  public static int CompareSent(Email x, Email y)
  {
    return x.Sent.CompareTo(y.Sent);
  }
  public static int CompareReceived(Email x, Email y)
  {
    return x.Received.CompareTo(y.Received);
  }
```

```
  public static int CompareSize(Email x, Email y)
  {
    return x.Size.CompareTo(y.Size);
  }
  //...
}
```

Listado 12-6 Métodos comparadores de Email

```
IList<Email> emailList = new List<Email>();
//...Poblando lista de emails

Console.WriteLine("\nOrdenar por el Sender ...");
Ordenar(emailList, new Compare<Email>(Utils.CompareSender));

Console.WriteLine("\nOrdenar por la Fecha de Envío ...");
Ordenar(emailList, new Compare<Email>(Utils.CompareSent));
```

Listado 12-7 Invocar a Ordenar usando delegados

Es interesante destacar aquí cómo encaja consistentemente la genericidad. En la definición genérica del método **`Ordenar`** el primer parámetro es de tipo `IList<T>` y el segundo de tipo `Compare<T>`, de modo que si ahora se invoca con `emaiList`, que es de tipo `IList<Email>`, entonces el segundo parámetro tiene que ser de tipo `Compare<Email>`, y por eso la creación de la instancia es `new Compare<Email>(…)`. El parámetro que se le pasa a este constructor tiene que ser entonces un método que tenga la signatura caracterizada por el tipo delegado `Compare<Email>`, es decir, un método de dos parámetros de tipo `Email` y que devuelva un `int`. Este es el caso de los métodos `Utils.CompareSender` y `Utils.CompareSent` que se han utilizado en el Listado 12-7.

El siguiente tipo delegado nos expresa métodos que sirven de predicado, es decir que aplicados a un objeto de tipo `T` nos devuelven un valor de tipo `bool` y que por tanto podemos usar para determinar si

12.1.2 Delegados Func y Action

Note que lo único que garantiza la definición del delegado `Compare<T>` es que éste caracteriza a métodos de dos parámetros del mismo tipo `T` que devuelven un `int`. El nombre `Compare` que hemos utilizado solo ayuda en todo caso en cuanto a la legibilidad; no hay nada que formalmente asegure lo que se quiere hacer bajo ese nombre, es decir, que en este ejemplo los métodos que se le asocien al tal delegado hagan realmente "comparación".

Para más facilidad y para uniformidad, C# ofrece el delegado `Func`

```
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
```

Este delegado caracteriza a un método que recibe dos parámetros (de tipos `T1` y `T2`) y devuelve un resultado de tipo `TResult`. El método `Ordenar` podría definirse usando directamente este delegado `Func` (Listado 12-1) en lugar de tener que definir el delegado `Compare`.

```
public static void Ordenar<T>(IList<T> a, Func<T, T, int> comp)
```

```
{
  for (int i = 0; i < a.Count - 1; i++)
    for (int j = i + 1; j < a.Count; j++)
      if (comp(a[j], a[i]) == -1)
      {
        T temp = a[i]; a[i] = a[j]; a[j] = temp;
      }
}
```

Listado 12-8 Método Ordenar usando el delegado Func

En este caso se ha utilizado `Func`, pero indicando un mismo `T` para los parámetros `T1` y `T2` (con lo que se especifica que el método que se utilice para crear el delegado para comparar tiene que tener los dos parámetros del mismo tipo) y el tipo `int` como parámetro concreto al parámetro genérico `TResult` (con lo que se indica que el método debe devolver un `int`). El código para invocar al método `Ordenar` del Listado 12-7 quedaría ahora como se muestra en el Listado 12-9.

```
IList<Email> emailList = new List<Email>();
//...Poblando lista de emails
Console.WriteLine("\n Ordena por Sender ...");
Ordenar(emailList, new Func<Email, Email, int>(Utils.CompareSender));

Console.WriteLine("\nOrdena por Fecha de Envío ...");
Ordenar(emailList, new Func<Email, Email, int>(Utils.CompareSent));
```

Listado 12-9 Llamada a Ordenar usando el delegado Func

C# tiene una gran cantidad de sobrecargas del delegado `Func` para expresar los métodos que devuelven un valor a partir de diferentes cantidades de parámetros (Listado 12-10). Note que siempre el parámetro genérico más a la derecha es el que como convenio se ha utilizado para indicar el tipo de resultado.

Un delegado `Func` muy utilizado es el que tiene como parámetro `TResult` al tipo `bool`, es decir utilizado de la forma `Func<T1, T2, …, Tn, bool>` estaría caracterizando a un método que devuelve un valor `bool` a partir de una secuencia de parámetros de los tipos `T1, T2, …Tn`. Un tal delegado se le suele denominar **predicado** o **condición** y como se verá más adelante tiene mucha utilidad usado combinadamente con los iteradores.

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T, out TResult>(T arg);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
public delegate TResult Func<in T1, in T2, in T3, out TResult>(T1 arg1, T2 arg2,
T3 arg3);
...
```

Listado 12-10 Sobrecargas del delegado Func

De manera similar a la de `Func` se tienen los delegados `Action` (Listado 12-11) para caracterizar a los métodos que representan una acción a realizar a partir de una secuencia de parámetros, pero que no devuelven ningún valor.

```
public delegate void Action();
public delegate void Action<in T>(T arg);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
public delegate void Func<in T1, in T2, in T3>(T1 arg1, T2 arg2, T3 arg3);
...
```

Listado 12-11 Sobrecargas del delegado Action

12.1.3 Delegados Anónimos

Como se muestra en los subepígrafes anteriores, para crear un delegado hay que especificar el nombre concreto de un método que cumpla con la signatura del delegado. Esto es lo que se hace, por ejemplo, en el Listado 12-7 y el Listado 12-9 cuando nos referimos a métodos que comparan dos emails. Podemos preguntarnos ¿dónde colocar tales métodos? ¿Cómo métodos estáticos de la propia clase `Email`? Esto no sería una buena práctica, porque quien define `Email` no necesariamente es quien decide tener una lista de email ni el que decide ordenar ésta, ni tampoco tiene que prever todas las formas de comparar emails que nos puedan interesar.

De modo que es mejor colocar estos métodos sin tener que atarlos a la definición del tipo que están comparando. En este ejemplo colocamos los métodos en una clase `Utils` (Listado 12-6).

Sin embargo, formar todo este tinglado de tener que definir un método estático para luego poder usarlo en la creación de un delegado, por ejemplo tener que hacer

```
Ordenar(emailList, new Func<Email, Email, int>(Utils.CompareSender));
```

es además muy verboso.

C# tiene también el recurso de **delegado anónimo**, en este caso el código del método que corresponda al delegado se escribe en el mismo código en que se quiere usar el delegado (Listado 12-12). Se le dice anónimo porque note que no tiene que haber sido definido aparte como un método de ninguna clase.

```
Console.WriteLine("\nOrdenar por Subject ...");
Ordenar(emailList, delegate(Email e1, Email e2)
                    { return e1.Subject.CompareTo(e2.Subject); });
Console.WriteLine("\nOrdenar por Size ...");
Utils.Ordena(emailList, delegate(Email e1, Email e2)
                    { return e1.Size.CompareTo(e2.Size);});
```

Listado 12-12 Usando delegados anónimos

12.2 Expresiones Lambda

Aun cuando se usen delegados anónimos, la sintaxis del Listado 12-12 puede seguir resultando verbosa. Sobre todo muy aparatosa para una situación tan sencilla como ésta, en la que el método consiste en una simple expresión.

C# dispone de una notación aún más simple mediante el recurso de **expresión lambda**.

El término expresión lambda ha sido tomado de la lógica matemática, en la que usa la letra griega lambda para denotar a una función. Es comúnmente usado en los lenguajes de programación considerados como funcionales y ha sido utilizado desde tiempos tan lejanos como los años 1960s con el lenguaje LISP.

Una expresión lambda tiene la forma

```
(parámetros) => expresión que computa un valor usando los parámetros
```

Con este recurso el código del Listado 12-12 puede expresarse de forma más abreviada como se muestra en el Listado 12-13

```
Console.WriteLine("\nOrdenar por Subject ...");
Ordenar(emailList, (Email e1, Email e2) => e1.Subject.CompareTo(e2.Subject));
Console.WriteLine("\nOrdenar por Size ...");
Ordenar(emailList, (Email e1, Email e2) => e1.Size.CompareTo(e2.Size));
```

Listado 12-13 Llamada a Ordenar usando expresiones lambda

Cuando una expresión lambda se asigna como parte derecha a una variable tipo delegado, como cuando se hace

```
Func<Email, Email, int) comp =
  (Email e1, Email e2) => e1.Size.CompareTo(e2.Size);
```

O cuando se pasa como parámetro real a un parámetro formal de tipo delegado (caso del método `Ordenar`), el compilador transforma la expresión lambda en un delegado y genera el código correspondiente al delegado.

Una expresión lambda también puede ser asignada como valor a una variable o parámetro de tipo `Expression` (ver Capítulo 15). En este caso no se genera el código del delegado, sino que la expresión lambda se representa en forma de un árbol. Tal árbol puede ser recorrido y analizado en ejecución. Incluso hay operaciones para construir tales árboles con la misma funcionalidad de lo que puede hacerse escribiendo directamente código C#. Esto permite lograr una suerte de dinamismo en tiempo de ejecución.

12.2.1 Inferencia del tipo de los parámetros en la expresión lambda

Para abreviar más aún la sintaxis en la expresión lambda, no hay que poner explícitamente el tipo de los parámetros siempre que el compilador lo pueda inferir a partir del contexto en que ésta se utiliza.

Por ejemplo, el código del Listado 12-13 se puede escribir como se muestra en el Listado 12-14, en donde se omitido indicar que el tipo de `e1` y `e2` es `Email`. En este caso el compilador de C# detecta que se está haciendo una llamada al método `Ordenar` con un primer parámetro (`emailList`) que es de tipo `IList<Email>`. Según la definición del método `Ordenar` (Listado 12-8), si el primer parámetro es de tipo `IList<T>` entonces el segundo parámetro es de tipo `Func<T, T, int>`, lo que quiere decir en este caso que `T` se concreta en el tipo `Email`. Por tanto, los dos parámetros `e1` y `e2` deben ser también de tipo `Email` y así se les considera en consecuencia al analizar el código de la expresión, ya que esta expresión se aplicará a los elementos del `IList<Email>`.

```
Console.WriteLine("\nOrdenar por Subject ...");
Ordenar(emailList, (e1, e2) => e1.Subject.CompareTo(e2.Subject));
Console.WriteLine("\nOrdenar por Size ...");
Ordenar(emailList, (e1, e2) => e1.Size.CompareTo(e2.Size));
```

Listado 12-14 Usando expresiones lambda con Inferencia de Tipos

Visual Studio también conoce de esta inferencia de tipo y por tanto nos puede ayudar cuando se usan los parámetros de la expresión (como se puede ver en la Figura 12-1)

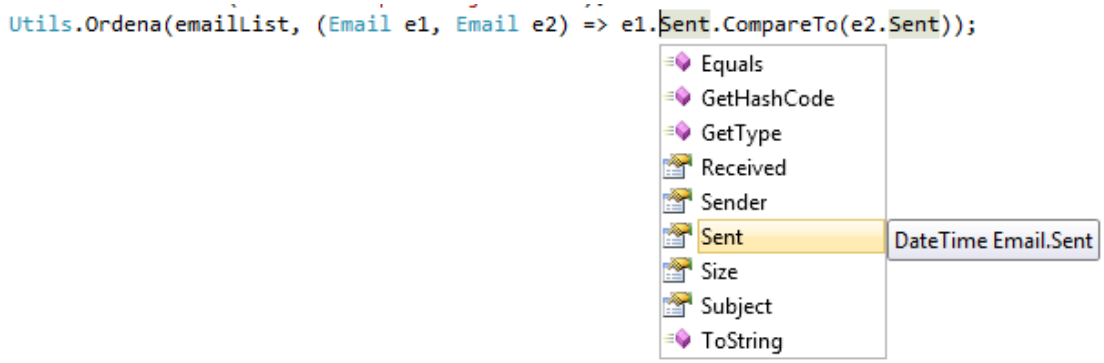

Figura 12-1 Infiriendo el tipo del parámetro de una expresión lambda

Inferir el tipo de una entidad (variable o parámetro) a partir de la construcción que calcula el valor que se le asocia (parte derecha de una asignación o parámetro concreto que se le pasa al parámetro formal en una llamada a un método) es una característica de la programación funcional y de los lenguajes de programación considerados funcionales. C# ha incluido este recurso como forma de facilitar la programación cuando es tedioso volver a escribir el nombre de un tipo para declarar el tipo de la entidad (variable, parámetro). También es útil cuando el tipo de la parte derecha es estático pero creado anónimamente por el compilador; en este caso el programador no conoce el nombre, pero sí conoce lo que quiere usar del tipo (ver subepígrafe 12.5.3).

12.3 Genericidad, iteradores y delegados: un perfecto "menâge a trois"

La combinación de objetos que sean `IEnumerable<T>` con delegados basados en el tipo `T` permite una gran expresividad en la obtención de nuevos enumerables o para obtener resultados basados en el procesamiento del enumerable fuente.

12.3.1 Métodos sobre enumerables que producen enumerables

Una combinación muy útil es la de aplicar un delegado del tipo `Func<T, bool>` a un `IEnumerable<T>` para "filtrar" el enumerable original. Si interpretamos el delegado como "una condición", entonces podemos devolver aquellos objetos que al aplicarle el delegado se obtiene el valor `true`. El método extensor `Where` definido en el Listado 12-15 implementa esta combinación.

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> fuente,
                                      Func<T, bool> condicion)
{
  foreach (T x in fuente)
    if (condicion(x)) yield return x;
}
```

Listado 12-15 Método Where para "filtrar" un IEnumerable según el cumplimiento de una condición

Continuando con el ejemplo de la lista de emails `emailList` y usando como expresión lambda la comparación de dos fechas, se puede escribir entonces

```
emailList.Where( (m) => m.Sent.CompareTo(new DateTime(2015,5,1))>0)
```

Esto nos da un enumerable con todos los emails recibidos después del 1/5/2015.

Si se hace

```
emailList.Where( (m) => m.Size>100)
```

tenemos un enumerable con todos los mails de tamaño mayor de `100`.

Otra combinación muy útil de genericidad, enumerables y delegados es la que implementa el método `Select` del **Listado 12-16**. Note que en este caso la fuente original de datos es un `IEnumerable<T>` y el delegado es de tipo `Func<T, R>` (es decir, una función que dado un parámetro de tipo `T` devuelve un resultado de tipo `R`). La aplicación del `Select` hace el efecto de una "transformación": el enumerable de valores de tipo `T` se ha "transformado" en un enumerable de valores de tipo `R`.

```
public static IEnumerable<R> Select<T, R>(this IEnumerable<T> fuente,
                                          Func<T, R> selector)
{
  foreach (T x in fuente)
    yield return selector(x);
}
```

Listado 12-16 Método Select para "transformar" los elementos de un IEnumerable

Si se aplica al enumerable `emailList` la transformación `(x) => x.Sender` obtenemos un enumerable de `string` con los nombres de todos los remitentes.

```
foreach (string s in emailList.Select((x) => x.Sender))
  Console.WriteLine(s);
```

Es interesante destacar aquí el proceso de deducción que hace el compilador con los tipos genéricos `T` y `R`. El tipo `T` (que en este ejemplo es `Email`) es el tipo de los elementos del enumerable original. Como el `Func` usado en la definición de `Select` es `Func<T,R>`, el compilador sabe entonces que la expresión lambda tiene que recibir un parámetro de tipo `Email`. Por lo tanto el código `x.Sender` es correcto, el compilador puede deducir entonces que `x.Sender` es `string`, y por tanto el `R` que devuelve el delegado es `string` y el enumerable que devuelve entonces el `Select` es `IEnumerable<string>`.

A su vez, ambos métodos `Where` y `Select` se pueden usar combinadamente. El código a continuación nos da un enumerable con los nombres de todos aquellos que han enviado mails sin Subject

```
emailList.Where(x => x.Subject.Length == 0).Select( x => x.Sender))
```

El lector puede ver en la documentación de la clase `Enumerable` una gran variedad de métodos que combinados con la aplicación de delegados (expresiones lambda) se aplican a enumerables para obtener otros enumerables.

La acción realizada por el método `Select` es conocida en matemática como **mapping**, es decir, un método que recibe como parámetros una secuencia de datos y una función y devuelve como resultado la secuencia formada de aplicarle la función a cada elemento de la secuencia original. Algunos lenguajes de programación (Python, por ejemplo) tienen construcciones sintácticas propias del lenguaje para hacer este tipo de transformaciones.

12.3.2 Funcionales de agregación.

Se pueden combinar los enumerables y los delegados para obtener resultados simples producto de aplicar el delegado a los elementos del enumerable.

La propia clase `Enumerable` ofrece el método `Aggregate`

```
public static TSource Aggregate<TSource>(this IEnumerable<TSource> source,
                                         Func<TSource, TSource, TSource> func);
```

Note que el delegado `Func` describe a toda función que recibe dos parámetros del mismo tipo y devuelve un resultado de ese tipo.

La funcionalidad de este método `Aggregate` se interpreta de la siguiente manera:

Si el enumerable `source` tiene un solo elemento, el resultado es dicho elemento. Si `source` tiene dos o más elementos, se aplica acumulativamente el delegado a los elementos de la secuencia.

De modo que si `listaEnteros` es un `IEnumerable<int>`, la ejecución del código siguiente da como resultado la suma de todos los enteros, es decir `15`.

```
IEnumerable<int> listaEnteros = new List<int> { 1, 2, 3, 4, 5 };
Console.WriteLine(listaEnteros.Aggregate((k, j) => k + j));
```

Note el poder expresivo de este funcional `Aggregate` según sea el valor del delegado `Func`. La aplicación siguiente da como resultado el mayor de la secuencia

```
listaEnteros.Aggregate((k, j) => (k<j)?j:k)
```

El Listado 12-17 muestra una posible implementación del método `Aggregate`.

```
static class Utils
{
  //...
  public static T MiAggregate<T>(this IEnumerable<T> fuente, Func<T, T, T> f)
  {
    T acumulado = fuente.First();
    foreach (T x in fuente.Skip(1))
      acumulado = f(acumulado, x);
    return acumulado;
  }
}
```

Listado 12-17 Una implementación de Aggregate

El método `Aggregate` puede sobrecargarse para cuando `Func` describe a una función que recibe dos parámetros, uno de tipo `R` y otro de tipo `T`, y devuelve un resultado de tipo `R`.

```
TAccumulate Aggregate<TSource, TAccumulate>(this IEnumerable<TSource> source,
        TAccumulate seed, Func<TAccumulate, TSource, TAccumulate> func);
```

Al método `Aggregate` se le pasa además un valor inicial de tipo `TAccumulate` que se da como "semilla" y un delegado que recibe un parámetro `TAccumulate` y un parámetro de tipo `TSource` y da como resultado un valor de tipo `TAccumulate` que se convierte en el nuevo acumulado para seguir repitiendo el proceso con todos los elementos de la secuencia.

Por ejemplo, la siguiente aplicación de `Aggregate` da como resultado la suma de las longitudes de todos los elementos de una secuencia de cadenas. Aquí los elementos de la secuencia son de tipo `string`, el valor a acumular es de tipo `int` y el valor que sirve de semilla es el `0`.

```
IEnumerable<string> listaCadenas = new List<string> {"rojo", "azul", "blanco"};
Console.WriteLine(listaCadenas.Aggregate(0, (k, s) => k + s.Length));
```

Piense el lector en cuál es la ventaja del código anterior en lugar de hacer

```
Console.WriteLine(listaCadenas.Select( s => s.Length).Aggregate((k, j) => k +
j));
```

Se deja al lector la implementación de esta sobrecarga del método `Aggregate`.

12.4 Funcionales

Al elevar las funciones al rango de "ciudadanos de primera clase" se aumenta el poder expresivo de la programación. Hasta ahora los valores que se manejaban o eran objetos o eran valores simples; ahora también se pueden manipular las funciones como valores, y si se puede pasar una función como valor de un parámetro, también se puede devolver una función como resultado.

Se denominan **funcionales** a aquellas funciones u operaciones que actúan sobre funciones para formar nuevas funciones; dicho en términos de C#, métodos que reciben delegados como parámetros y devuelven como resultado un nuevo delegado.

12.4.1 Composición de funciones

Un ejemplo trivial es el de la composición de funciones. Si `F` es un delegado del tipo `Func<T1, T2>` y `G` es un delegado del tipo `Func<T2,R>`, un método `Compose` nos devolvería como resultado un delegado de tipo `Func<T1, R>`, es decir que reciba un parámetro de tipo `T1`, le aplique el delegado `F` para obtener un resultado de tipo `T2` y a ese resultado le aplique el delegado `G` para obtener un resultado de tipo `R`. El Listado 12-18 nos muestra el código de un tal método `Compose` y un ejemplo de aplicación.

```
static Func<T1, R> Compose<T1, T2, R>(Func<T1, T2> F, Func<T2, R> G)
{
  return (x) => G(F(x));
}
//...
Func<string, string> Reverso =   (s) => { string result = "";
                                          for (int k = s.Length-1; k>=0; k--)
                                            result = String.Concat(result, s[k]);
                                          return result;};
Func<string, string> Mayuscula = (s) => s.ToUpper();
Func<string, string> ReversoEnMayuscula = Compose(Reverso, Mayuscula);
```

Listado 12-18 Composición de funciones

Mucho ruido y pocas nueces puede que piense el lector. La composición de funciones es algo que siempre hemos podido hacer, si tengo dos métodos `F` y `G` y el tipo del resultado de uno es el tipo del parámetro del otro, cuando quiera aplicar los dos se puede escribir `G(F(x))` sin tener que usar un método como `Compose` para obtener un nuevo método. El delegado `ReversoEnMayuscula` se podía sencillamente haber obtenido escribiendo directamente

```
Func<string, string> ReversoEnMayuscula = (s) => Mayuscula(Reverso(s));
```

Sin embargo, la sencillez de este ejemplo ha servido para ilustrar cómo escribir un método que devuelva un delegado a partir de otro delegado. El ejemplo del epígrafe a continuación ilustra mejor esta capacidad de definir funcionales.

12.4.2 Decoradores

Se le suele denominar **decorador** a un funcional que "decora" un método con una determinada funcionalidad. La intención es que el método "decorado" sea utilizado en lugar del método original de tal modo, que cuando se llame al método decorado éste realice alguna acción previa y luego llame al método original.
El funcional `MideTiempo` del Listado 12-19 recibe como parámetro una función (un delegado o expresión lambda) de un parámetro de tipo `T` y de valor de retorno de tipo `R` y devuelve como resultado una función de un parámetro de tipo `T` pero de valor de retorno un tuplo formado por un valor de tipo `R` (el valor que hubiese devuelto la función original) y un valor de tipo `long` que es el tiempo en milisegundos que demoró la ejecución de la función original.

```
static Func<T, Tuple<R, long>> MideTiempo<T, R>(Func<T, R> f)
{
  return (T x) =>
  {
    Stopwatch crono = new Stopwatch();
    crono.Restart();
    R result = f(x);
    crono.Stop();
    return Tuple.Create(result, crono.ElapsedMilliseconds);
  };
}
```

Listado 12-19 Decorador para medir el tiempo de ejecución de una función

El código del Listado 12-20 nos muestra la aplicación del funcional `MideTiempo` para obtener una función que calcule la función de Fibonacci y a la vez mida el tiempo que ésta demoró. Suponga una implementación recursiva de la función de Fibonacci como la que se ha incluido en el listado.

```
static long FibRecursivo(int k)
{
  if (k <= 2) return 1;
  else return FibRecursivo(k - 2) +
              FibRecursivo(k - 1);
}
...
var FibDecorado = MideTiempo(new Func<int, long>(FibRecursivo));
var t = FibDecorado(35);
Console.WriteLine("Fibonacci de {0} es {1} calculado en {2} ms",
                  35, t.Item1, t.Item2);
```

Listado 12-20 Midiendo el tiempo de calcular la función de Fibonacci

Con toda intención se ha utilizado aquí una implementación recursiva ineficiente para el cálculo de Fibonacci (ver Capítulo 8 de Recursividad) para apreciar mejor el tiempo que demora el cálculo y porque en el epígrafe siguiente veremos cómo utilizar un decorador para mejorar esta ineficiencia.

Note la utilización en este ejemplo de la inferencia de tipo al aplicar el especificador `var`. Al escribir

```
var FibDecorado = MideTiempo(new Func<int, long>(FibRecursivo));
```

nos evitamos tener que escribir explícitamente

```
Func<int, Tuple<long, long>> FibDecorado
```

y dejamos que sea el compilador quien lo infiera de la parte derecha. De igual modo, en la asignación `var r = FibDecorado(35);` el compilador infiere que el tipo de `t` es `Tuple<long, long>` y por tanto es correcto referirse a sus componentes con `t.Item1` y `t.Item2`.

12.4.3 Implementación del patrón memorización usando un decorador

El método recursivo `FibRecursivo` del Listado 12-20 para calcular el enésimo término de la sucesión de Fibonacci es ineficiente porque las llamadas recursivas repiten cálculo innecesario cuando dentro de la ramificación de la recursividad se vuelven hacer llamadas con valores que ya fueron calculados.

Con una solución tradicional imperativa estos cálculos redundantes podrían evitarse si almacenamos en un array los valores de los elementos de la sucesión que ya fueron calculados para utilizarlos posteriormente sin necesidad de volverlos a calcular (Listado 12-21).

```
static long[] memoria = new long[1000];
static long Fibonacci(int n)
{
  long result;
  if (memoria[n] != 0)
    return memoria[n];
  if (n <= 2)
    result = 1;
  else
    result = Fibonacci(n - 2) + Fibonacci(n - 1);
  memoria[n] = result;
  return result;
}
```

Listado 12-21 Método de Fibonacci memorizando en un array

Esta solución tiene la limitación de que está atada específicamente al método `Fibonacci`, aplicar el mismo patrón a otro método implicaría escribir un código similar en dicho método. Por otra parte, nada indica en el código que el array `memoria` debe ser solo para usar en el código del método `Fibonacci`. Vamos a ver una solución más genérica usando programación funcional.

El decorador `Memorize` del Listado 12-22 puede aplicarse a una función de tipo `Func<T, R>`. Note que el método internamente tiene un diccionario `dicc` para guardar pares `T`, `R`, es decir según el valor de tipo `T` para el que quiera calcularse la función original `f` se guarda en el diccionario el valor de tipo `R` calculado. De este modo, en un próximo intento de calcular la función para el mismo valor éste se busca primero en el diccionario para no recalcularlo.

```
static Func<T, R> Memorize<T, R>(Func<T, R> f)
{
```

```
  Dictionary<T, R> dicc = new Dictionary<T, R>();
  return ((T value) => {
            R result;
            if (dicc.TryGetValue(value, out result))
              return result;
            else
            {
              result = f(value);
              dicc.Add(value, result);
              return result;
            }
          });
}
```

Listado 12-22 Funcional que recibe una función y devuelve la función decorada con la capacidad de memorización

Note que la creación del diccionario `dicc` se hace dentro del código del método `Memorize`, es decir, su ámbito es local a dicho método y por tanto se puede usar dentro de la función lambda que se define dentro del método y que retorna como resultado. El compilador de C# se encarga de que después de terminar la ejecución del método `Memorize` el diccionario queda atado a la función que se retorna sin ser accesible desde otra parte del código[1].

El Listado 12-23 nos muestra cómo usar este método `Memorize` para solucionar el problema de Fibonacci. Lo importante es que internamente la acción de memorizar queda encapsulada y transparente al programador. Para usar este patrón el programador no tiene que conocer las interioridades del `Memorize`, sino que debe concentrarse en plantear su solución recursiva como le resulte más simple y aplicarle luego la memorización si piensa que puede haber cálculos redundantes.

```
Func<int, long> Fibonacci = null;
Fibonacci = (int n) =>
  {
    if (n > 1)
      return Fibonacci(n - 1) + Fibonacci(n - 2);
    else
      return n;
  };
Fibonacci = Memorize(Fibonacci);
```

Listado 12-23 Aplicación de la memorización a la función de Fibonacci

[1] En la terminología de la programación funcional se dice que el diccionario forma parte de la **clausura** de la función.

Una limitante de esta solución es que el método `Memorize` se aplica a funciones de un parámetro. Si se quisiera aplicar el mismo patrón a funciones de dos parámetros, habría que repetir un código similar y con un diccionario con dos llaves y así sucesivamente para las diferentes cantidades de parámetros. El problema es que la genericidad utilizada en la especificación del tipo `Func` exige poner explícitamente una cantidad de parámetros, y por tanto no se puede definir un único método `Memorize` que sirva para todos los casos.
Una solución más flexible usando tipado dinámico se puede ver en el Capítulo 15

Más que resolver el problema de Fibonacci, para el que sabemos hay una solución iterativa muy eficiente, la intención de este ejemplo ha sido ilustrar cómo se pueden usar los recursos de programación funcional de C# para escribir funcionales, es decir, funciones que devuelvan funciones.

Para decidir aplicar este patrón de memorización se debe valorar si buscar y guardar en un diccionario vale la pena en comparación con calcular de nuevo la función. En este ejemplo se manifiesta un viejo dilema del diseño y análisis de algoritmos: "Memoria vs Procesamiento". Aunque en la actualidad, en términos de hardware, la memoria es más barata que el procesador, esto no significa que el programador la malgaste inconscientemente, por lo que deberá considerar inteligentemente el uso o no de la memorización según el problema de que se trate.

12.5 LINQ

Hasta la fecha no es secreto que la mayor parte de las aplicaciones de la industria y la academia se desarrollan hoy día con el paradigma de la Programación Orientada a Objetos.

Paralelamente, en el universo de los datos, han surgido algunos SisSubjects Bases de Datos Orientadas a Objetos, y aunque por ahí sobreviven algunos, lo cierto es que no se han impuesto masivamente. En la actualidad, en el mundo de los datos (al menos en los empresariales) se siguen imponiendo las **Bases de Datos Relacionales**, dominado por sus señores de la guerra: los servidores SQL. Con el desarrollo de la Web, también han empezado a proliferar las llamadas Bases de Datos non-SQL y diferentes formatos para los datos (como **XML**, con un vehículo bastante común de entendimiento).

De modo que ha habido una impedancia entre el lenguaje en que se representan, almacenan y acceden los datos y el lenguaje en el que se programa la lógica del negocio que utiliza a los mismos. El intento de facilitar a las aplicaciones que tengan acceso a datos no es nuevo. En muchos lenguajes se pueden colocar sentencias SQL para comunicarse con bases de datos SQL, aunque desde el punto de vista del lenguaje de programación son sólo oscuras cadenas que son analizadas en tiempo de ejecución por los servidores SQL, lo que hace incómoda la integración entre los datos y la lógica de la aplicación. Se han desarrollado diferentes APIs para "conectar" los lenguajes con los datos (por ejemplo en .NET pueden mencionarse `System.XML` y `ADO.NET`. También se tienen diferentes "mapeadores" relacionales a objetos (conocidos por ORM, por Object Relational Mapping). Pero se echaba en falta la existencia de una forma de trabajo simple, consistente y uniforme que integre ambos mundos.

Es en este contexto en que se desarrolla el lenguaje de consulta LINQ (Language INtegrated Query)[2] para integrarlo a la plataforma .NET y, entre otros, al lenguaje C#.

Un estudio en profundidad de LINQ se sale de los objetivos de este libro (el lector puede encontrar una buena cantidad de títulos de interés listados en la Bibliografía). Sin embargo, LINQ es un buen ejemplo de la integración de algunos de los conceptos y recursos que se presentan en este libro, como el tipado estático, genericidad, iteradores y programación funcional. Por estas razones, el resto de este capítulo está dedicado a presentar e ilustrar algunas de las características fundamentales de LINQ.

12.5.1 Una consulta al estilo imperativo tradicional

Considere un modelo de datos como el que expresa con el código del Listado 12-24, con el que se representan datos de cine. Un género tiene un nombre y un conjunto de films de ese género. Un director tiene un nombre y un conjunto de films dirigidos por ese director, y un film tiene un título, un género, un director, el año de producción y si el film ha sido ganador de algún Oscar[3]. Un objeto de tipo `CineDB` tiene entonces una colección de films, una colección de directores y una colección de géneros.

```
class Genre{
  public string Name { get; }
  public IEnumerable<Film> Films { get; }
  //...
}
class Film{
  public string Title { get; set; }
  public Genre Genre { get; set; }
  public Director Director { get; set; }
  public bool Oscar { get; set; }
  public int Rating { get; set; }
  public int Year { get; set; }
  //...
}
class Director {
  public string Name { get; set; }
  public IEnumerable<Film> Films { get; }
  //...
}
class CineDB{
  public IEnumerable<Film> Films { get; }
  public IEnumerable<Director> Directors { get; }
  public IEnumerable<Genre> Genres { get; }
  //...
}
```

Listado 12-24 Clases para representar datos de cine

Suponga que tiene un objeto `cineDB` que se ha poblado con datos y que desea obtener y listar una colección formada por el título y el director de aquellos films que sean del

[2] LINQ fue lanzado oficialmente en el .NET Framework con Visual Studio 2008.
[3] No lo tome a mal si usted es un cinéfilo. Esta es una simplificación con propósitos didácticos; todos sabemos que un film puede ser de más de un género, que puede tener más de un director, y que puede haber ganado más de un Oscar, según las categorías.

género `Drama`. Una solución imperativa tradicional, que se empeñe en no utilizar ninguno de los recursos que hasta ahora debemos haber aprendido, se muestra en el Listado 12-25.

```
class MiFilm
{
  public string Titulo{get; set;}
  public string Director{get; set;}
}
...
class Program
  static IEnumerable<Film> Dramas(CineDB cineDB)
  {
    List<Film> dramas = new List<Film>();
    foreach (Film f in cineDB.Films) {
      if (f.Genre.Name == "Drama")
        dramas.Add(new MiFilm(Titulo = f.Title, Director = f.Director.Name));
  }
...
```

Listado 12-25 Método imperativo para obtener los films de género Drama

Si se quisiera listar el título y el director de cada drama, se podría escribir un código como el del Listado 12-26

```
CineDB cineteca = new CineDB();
...poblar cineteca...
IEnumerable<MiFilm> dramas = Dramas(cineteca);
Console.WriteLine("DRAMAS");
foreach (MiFilm f in dramas)
  Console.WriteLine("{0,-24}  {2})", f.Titulo, f.Director);
```

Listado 12-26 Listar el título y el director de los films de género Drama

Este enfoque tiene múltiples defectos:

- Es muy rígido, solo vale para obtener los dramas. Si se quiere hacer cualquier otra consulta habría que escribir un método propio para la misma.
- El método `Dramas` construye explícitamente en memoria una lista con el resultado, con independencia de lo que haga después el que quiere procesar el resultado de la consulta.
- Hemos tenido que definir un tipo `MiFilm` para expresar los valores que nos interesan en el resultado.

12.5.2 Consulta más simple, genérica, reusable y eficiente

Utilizando los métodos extensores `Where` y `Select` del epígrafe 12.3 podemos expresar la consulta de forma más reusable y con la notación punto de la programación orientada a objetos.

La consulta se puede ir escribiendo con la notación punto de la orientación a objetos

```
cineDB.Films.Where(f => f.Genre.Name == "Drama")
```

La notación objetual de izquierda a derecha permite que nos pueda ayudar el Intellisense, ya que al saber el tipo de la parte izquierda nos puede mostrar los recursos de ese tipo que pueden ser utilizados.

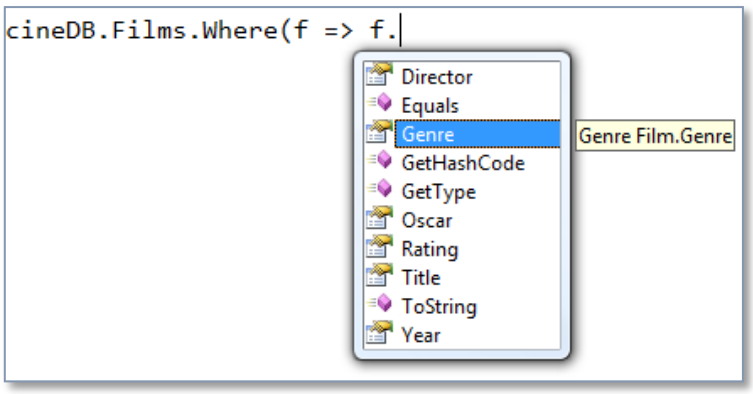

Figura 12-2 Usando Intellisense en una consulta

La consulta para seleccionar el título y el director se puede completar entonces escribiendo

```
var dramas = cineDB.Films.Where(f => f.Genre.Name == "Drama")
               .Select(f => new MiFilm{ Titulo = f.Title,
                                        Director = f.Director.Name });
```

Note que la utilización de la especificación `var` para declarar la variable `dramas` es una comodidad sintáctica, porque nos libera de tener que escribir explícitamente

```
var IEnumerable<MiFilm> dramas = …
```

Como los métodos `Where` y `Select` han sido definidos utilizando el operador `yield`, su ejecución es perezosa (*lazy*), y el enumerable que se devuelve como resultado no ha construido físicamente una colección en memoria. Los objetos resultantes del `Where` `Select` se van formando por demanda según el resto de la aplicación lo pida, como por ejemplo cuando se hace

```
foreach (MiFilm f in dramas)
  Console.WriteLine("{0,-24}  {2})", f.Titulo, f.Director);
```

En este caso se necesita ir teniendo en la variable `f` los objetos concretos, ya que se les pide los valores de sus propiedades `f.Titulo` y `f.Director`.

12.5.3 Tipos Anónimos

Aun cuando el enfoque anterior mejore significativamente las limitaciones de la solución "cableada" del epígrafe 12.5.1, sigue siendo un incordio tener que definir un nuevo tipo `MiFilm` solo con el propósito de utilizarlo como resultado de la consulta.

Se tiene el recurso de definir un **tipo anónimo** y dejar que sea el compilador que sea quien le dé un nombre. A fin de cuentas, para usarlo como en el ejemplo anterior lo único que nos interesa saber es que tiene una propiedad `Titulo` que es de tipo `string` y una propiedad `Director` que también es `string`.

En este caso sí se justifica la necesidad de disponer del especificador `var` para escribir

```
var dramas = cineDB.Films.Where(f => f.Genre.Name == "Drama")
```

```
                .Select(f => new { Titulo = f.Title,
                                   Director = f.Director.Name });
```

porque no sabemos realmente cuál es el nombre que el compilador asignó y por tanto no podemos escribir

```
IEnumerable<??> dramas = ...
```

El tipo sí existe estáticamente y el compilador lo conoce y por tanto se puede inferir el tipo de dramas de la parte derecha de la asignación. Pruebe a hacer `dramas.GetType().Name`.

Los tipos anónimos pueden ser también útiles para definir componentes internas de otros tipos. El código del Listado 12-27 es correcto porque el compilador usa el mismo tipo anónimo para `triangulo1` que para `triangulo2`. Al igual que los tipos de las componentes internas vértices son también los mismos

```
var triangulo1 = new
{
  V1 = new { X = 100, Y = 200 },
  V2 = new { X = 400, Y = 500 },
  V3 = new { X = 200, Y = 300 }
};
var triangulo2 = new
{
  V1 = new { X = 10, Y = 20 },
  V2 = new { X = 40, Y = 50 },
  V3 = new { X = 20, Y = 30 }
};

Console.WriteLine(t1);
Console.WriteLine(t1.V2);
Console.WriteLine(t1.V2.Y);
triangulo1 = triangulo2; //Asignacion correcta, los tipos anónimos son los
mismos
Console.WriteLine(t1);
Console.WriteLine(t1.V2);
Console.WriteLine(t1.V2.Y);
```

Listado 12-27 Uso de tipos anónimos para definir triángulos

La ejecución de este código da la salida que se muestra en la Figura 12-3. Note que implícitamente todo tipo anónimo tiene redefinido el método `ToString` para devolver una cadena formada recursivamente de la forma *{nombre propiedad = ToString de la propiedad, ...}*

```
{V1 = {X = 100, Y = 200}, V2 = {X = 400, Y = 500}, V3 = {X = 200, Y = 300}}
{X = 400, Y = 500}
500
{V1 = {X = 10, Y = 20}, V2 = {X = 40, Y = 50}, V3 = {X = 20, Y = 30}}
{X = 40, Y = 50}
```

Figura 12-3 Resultado de la ejecución del código del Listado 12-27

12.5.4 Consulta en notación más declarativa

La consulta del epígrafe 12.5.3 se puede escribir en notación aún más declarativa al estilo del lenguaje SQL

```
var dramas = from f in cineDB.Films
             where f.Genre.Name== "Drama"
             select new { Titulo = f.Title, Director = f.Director.Name }
```

En este caso es el compilador quien hace la transformación del código anterior a la notación punto como la de 12.5.3 utilizando los métodos `Where` y `Select`.
El lenguaje LINQ incluye en su sintaxis **operadores de consulta** para lograr casi toda la funcionalidad que se puede lograr con una consulta SQL. El código (Listado 12-28) lista el título, género y nombre del director de los filmes mejor evaluados y ordenados por el título

```
 var mejoresFilms =
   from f in cineDB.Films
   where f.Rating == 5
   orderby f.Title
   select new { Titulo = f.Title , Genero = f.Genre.Name,
                Director = f.Director.Name };

Console.WriteLine("Films calificados con 5");
Console.WriteLine("----------------------\n");
 foreach (var f in mejoresFilms)
    Console.WriteLine("{0,-24}  {1,15}  ({2})",
                      f.Titulo, f.Genero, f.Director);
```

Listado 12-28 Consulta declarativa de los mejores films ordenados por el titulo

LINQ es un ejemplo de lo que se denomina un DSL (Domain Specific Language) textual embebido en este caso en el lenguaje C#. En este caso es el compilador el que se encarga de transformar el código escrito en LINQ a código C#.

La consulta del Listado 12-29 nos da los mejores films agrupados por el género

```
var mejoresFilmsPorGenero =
  from f in cineDB.Films
  where f.Rating == 5
  group f by f.Genre;
```

Listado 12-29 Uso del operador group by. Mejores films agrupados por género.

12.5.5 Conexión con una base de datos

Como ya se explicó, la magnitud de LINQ se sale del alcance de este libro y su inclusión en este capítulo ha sido para servir de colofón para mostrar las capacidades de programación integrada y consistente de C#. No obstante, veamos algunas aclaraciones finales para no dejar nada oscuro.

En este epígrafe hemos utilizado como modelo de datos de cine el expresado en el Listado 12-24.

Los ejemplos posteriores se han basado en que de alguna manera la aplicación ha poblado la instancia de `cineDB` con las correspondientes interrelaciones entre las

instancias. Note que en una instancia de `Film` se usan instancias de `Genre` y de `Director`, a su vez en una instancia de `Director` y en una instancia de `Genre` se usan instancias de `Film`.

En una aplicación real, lo más probable es que la fuente de datos no haya sido poblada por la propia aplicación que la consulta y procesa, sino que la fuente de datos sea de alguna manera "externa" a la aplicación, por ejemplo una base de datos relacional SQL.

La Figura 12-4 nos muestra tres tablas de la base de datos con films, directores y géneros. Lo que faltaría por hacer es definir un "mapeador" que conecte transparentemente a la aplicación con la fuente de datos para consultar (y actualizar si fuese también el caso) los datos en la fuente original. LINQ con Visual Studio ofrecen diferentes herramientas para conectarse con una base de datos y generar transparentemente el código que mapea los datos en las tablas relacionales de la base de datos con objetos al estilo del modelo expresado en el modelo del Listado 12-24.

La política de nombrado del mapeador es bastante simple, pero no viene al caso entrar en detalles aquí. Considere que el desarrollador de la aplicación conoce los nombres asignados a las distintas fuentes correspondientes a las tablas de la base de datos (en este ejemplo han quedado con los nombres en plural, o sea `Films`, `Directors` y `Genres`) proveedores de los datos de las tablas `Film`, `Director` y `Genre` respectivamente, y que los enlaces entre objetos son consecuencia de la utilización de llaves foráneas en las tablas originales.

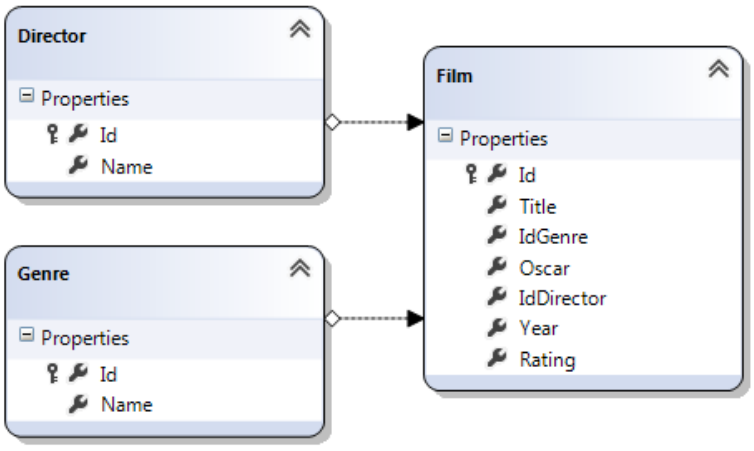

Figura 12-4 Tablas de una base de datos SQL

La aplicación simplemente debe crear una instancia del objeto que expresa el "mapeo"

```
DataClasses1DataContext cineDB = new DataClasses1DataContext();
```

Precisamente la inferencia de tipos en C# nos libera de tener que conocer todos los detalles de ese "mapeo" siempre que, por supuesto, obviamente conozcamos los nombres y tipo de la información que queremos consultar.

En este caso, cuando el compilador de C# procesa una consulta como

```
var dramas = from f in cineDB.Films
             where f.Genre.Name== "Drama"
             select new { Titulo = f.Title, Director = f.Director.Name};
```

Lo que genera realmente es una expresión lambda que se representa como árbol (un tipo `IQueryable<tipo anónimo>`) que actuará como `IEnumerable` y que en ejecución al iterar sobre él transformará la consulta a una secuencia de comandos SQL que se le aplican a la base de datos para ir transformando la información obtenida de la base de datos en los objetos del "mapeo".