Errores Comunes en C Sharp

Usar Foreach o For en vez de LINQ

En casi todas las aplicaciones, eventualmente se deberá enumerar los valores y almacenarlos en una Lista o Colección. Podríamos tener que recorrer miles de registros.

Consideremos sacar una lista de clientes, por ejemplo. Si tenemos 100.000 clientes, la iteración a través de cada uno de ellos solo para encontrar un conjunto de datos específico no es eficiente. En lugar de usar un bucle foreach o for, usemos LINQ (Language-Integrated Query), que es una característica integrada de .NET que está diseñada para facilitar la consulta de objetos como colecciones y listas.

Muchas veces pensamos que, debido a la similitud entre las palabras clave LINQ y las declaraciones SQL, su único uso es en el código que consulta las bases de datos.

Si bien la consulta de la base de datos es un uso muy frecuente de las declaraciones LINQ, en realidad funcionan sobre cualquier colección enumerable (es decir, cualquier objeto que implemente la interfaz IEnumerable).

Ejemplo:

Forma ineficiente

```
foreach (Cliente cliente in ListaCliente) {
   if (cliente.Comunidad == "MADRID") {
     saldo += cliente.saldo;
   }
 }
```

Forma eficiente

```
var clientes = (from cliente in ListaCliente
                where cliente.Comunidad == "MADRID"
                select cliente.saldo).Sum();
```

Con esta línea de código LINQ nos devuelve 1000 clientes, en lugar de recorrer 100.000 clientes. Trabajar con 1.000 clientes a la vez es mucho más eficiente que iterar a través de cada uno de los 100.000.

No se deben considerar los objetos subyacentes en una declaración LINQ

LINQ es ideal para abstraer la tarea de manipular colecciones, ya sean objetos en memoria, tablas de base de datos o documentos XML. Pero el error aquí es asumir que vivimos en un mundo perfecto. De hecho, las declaraciones LINQ idénticas pueden devolver resultados diferentes cuando se ejecutan en los mismos datos exactos, si esos datos se encuentran en un formato diferente.

Por ejemplo, considere la siguiente declaración:

```
decimal total = (from cliente in ListaCliente
                 where cliente.Status == "activo"
                 select cliente.Saldo).Sum();
```

¿Qué sucede si uno de los objetos cuenta.Status es igual a "Activo", es decir, con la letra "A" mayúscula? Si el DbSet ListaCliente se tratara de un objeto, la expresión **where** todavía coincidirá con ese elemento, pero, si ListaCliente estuviera en una matriz en memoria, no coincidiría y, por lo tanto, produciría un resultado diferente para la variable **total**.

Cuando los objetos subyacentes en una declaración LINQ son referencias a datos de la tabla SQL como es en este caso con el objeto DbSet de Entity Framework en este ejemplo, la declaración se convierte en una declaración T-SQL, con lo que los operadores siguen las reglas de programación de T-SQL, no las reglas de programación de C #, por lo que la comparación en el caso anterior termina por no distinguir entre mayúsculas y minúsculas.

En general, a pesar de que LINQ es una forma útil y coherente de consultar colecciones de objetos, en realidad, todavía necesitamos saber si nuestra declaración se traducirá o no a C # "*detrás de la escena*" para garantizar que el comportamiento del código sea como se espera en el tiempo de ejecución.

Usar "Var" innecesariamente

En muchos casos, no siempre conocemos el resultado que vamos a esperar cuando usamos LINQ para recuperar una colección de valores y "**var**" nos ayuda a evitar errores de código si se devuelve un tipo de datos nulo o que no esperábamos.

Sin embargo, debemos definir un tipo de datos si sabemos cuál será este. Esto nos ayuda con la legibilidad, ya que otros programadores podrán trabajar con nuestro código y mantenerlo sin tener que esforzarse para entender la lógica y la salida.

Modificando el ejemplo anterior:

```
var total = (from cliente in ListaCliente
                 where cliente.Status == "activo"
                 select cliente.Saldo).Sum();
```

En este ejemplo, sabemos que la suma es probablemente un decimal. Sin embargo, si sabemos que el resultado es siempre un valor decimal, debemos declarar su variable como **decimal** y no usar **var**. Cuando otros desarrolladores lean nuestro código, sabrán que el valor de retorno es un **decimal** y no un tipo numérico alternativo, como un **entero**. Con lo cual la forma correcta de declarar esta consulta es:

```
decimal total = (from cliente in ListaCliente
                 where cliente.Status == "activo"
                 select cliente.Saldo).Sum();
```

Usar variables públicas en vez de Propiedades

Las propiedades son comunes en la programación orientada a objetos (POO), pero ¿por qué usarlas cuando puedes hacer públicas todas las variables de tu clase? Porque podemos controlar quién puede establecer una propiedad con propiedades POO, pero no podemos con una variable pública.

Si vemos este código:

```
public decimal Total { get; protected set; }
```

En esta declaración, solo la clase en sí misma o las clases derivadas pueden establecer el **total**. Consideremos una clase de pedidos que calcula el total para el pedido de un cliente. No deseamos que ninguna clase externa cambie el pedido, pero sí deseamos que la clase **Pedidos** y las clases derivadas agreguen o resten valores almacenados por la variable **Total**. Si simplemente hace pública la variable, cualquier clase puede cambiar el total de un pedido sin ninguna restricción.

No Liberar Objetos

El entorno CLR emplea un recolector de basura, por lo que no necesitamos liberar explícitamente la memoria creada para ningún objeto. De hecho, no podemos, pero eso no significa que solo podamos olvidarnos de todos los objetos una vez que hayamos terminado de usarlos ya que muchos tipos de objetos encapsulan algún otro tipo de recurso del sistema, como por ejemplo, un archivo de disco, una conexión de base de datos, socket de red, etc... Dejar estos recursos abiertos puede agotar rápidamente la cantidad total de recursos del sistema, mermando el rendimiento y, en última instancia, provocando fallas en el programa.

Si bien los métodos destructores se pueden definir en cualquier clase de C #, el problema con los destructores es que no se puede saber con seguridad cuándo se llamarán ya que estos son llamados por el recolector de basura en un momento indeterminado en el futuro. Tratar de sortear estas limitaciones forzando la recolección de basura mediante **GC.Collect()** no es una buena práctica de C # , ya que bloqueará el hilo durante un tiempo desconocido mientras recopila todos los objetos elegibles para la recolección.

Las fugas de recursos son una preocupación en casi cualquier entorno. Sin embargo, C # proporciona un mecanismo robusto y fácil de usar que, si se utiliza, puede hacer que las fugas sean mucho más raras. El .NET Framework define la interfaz **IDisposable**, que consiste únicamente en el método **Dispose()**. Cualquier objeto que implementa **IDisposable** espera llamar a este método siempre que el consumidor del objeto termine de manipularlo. Esto da como resultado la liberación explícita y determinista de los recursos.

Si estamos creando y desechando un objeto dentro del contexto de un solo bloque de código, es prácticamente imposible olvidarse de llamar **Dispose()**, porque C # proporciona una declaración **using** que asegurará que se llame **a Dispose()**, sin importar cómo se salga del bloque de código.

Ejemplo:

No Eficiente

```
file.Read(buffer, 0, 100);
```

En el código anterior, si no libera el objeto de archivo, crea una pérdida de memoria en la aplicación. Puede evitar esta situación con la instrucción using.

Eficiente

```
using (FileStream archivo = File.ObboligrafoRead("numbers.txt")) {
archvio.Read(buffer, 0, 100);
}
```

Ahora la aplicación lee un archivo y elimina el objeto cuando está terminado.

Usar “” en vez de string.Empty

Esta es una molestia menor para los desarrolladores, y se trata más de legibilidad y mantenimiento de código que de eficiencia. La diferencia de rendimiento es menor, pero es difícil de leer y puede pasarse por alto como otra cosa. Por ejemplo, "" puede pasarse por alto y leerse como " ", que es un valor completamente diferente.

En lugar de usar "" para inicializar una cadena, es mejor usar string.Empty. Este valor inicializa la cadena y no puede leerse accidentalmente como un valor diferente.

Usar Excepciones genéricas Try-Catch

Muchos desarrolladores nuevos usan la clase de excepción genérica en lugar de especificar la excepción que se lanzó. Todas las demás clases de C # se derivan de la clase base **Exception**, y podemos crear clases de excepción personalizadas que heredan la clase base. Sin embargo, siempre se debe utilizar excepciones específicas.

No Eficiente

```
try {
nuevoEntero = int.Parse(astring);
    } catch (Exception e)
    {
        // código fuente
    }
```

Eficiente

```
try {
nuevoEntero = int.Parse(astring);
    } catch (FormatException) {
        // código fuente
    }
```

Este tipo de diseño try-catch especifica la excepción que se produce, por lo que puede registrar más fácilmente los errores, la depuración y la solución de problemas. Podemos usar la clase base **Exception** para posibles excepciones desconocidas, pero debe usarse con moderación.

Envolver métodos completos en un bloque Try-Catch

Ejemplo:

```
try {
nuevoEntero = int.Parse(astring);
    } catch (FormatException) {
        // código fuente
    }
```

Se encapsula solo una declaración con el controlador de excepciones. Un error común que muchos desarrolladores nuevos cometen es envolver un bloque try-catch alrededor de un método completo.

Debemos usar los bloques try-catch en las secciones lógicas del código. Por ejemplo, no usar un bloque cuando tenemos un método que lee un archivo, almacena el contenido en una variable y luego envía los datos a una base de datos. Es mejor dividir los bloques para manejar la lectura del archivo, los bucles que almacenan datos y luego la sección que carga la información en una base de datos.

Usar la concatenación de cadenas incorrectamente

En los lenguajes de programación más antiguos, era común utilizar el signo más **(+)** para concatenar cadenas. El problema con esto es que es una forma ineficiente de concatenar cadenas, por lo que Microsoft presentó **StringBuilder** para ayudar con la memoria y los problemas de rendimiento.

Es conveniente usar **StringBuilder** cuando queremos concatenar cadenas o manipularlas a lo largo del código. No siempre es necesario usarlos para cadenas simples y básicas, pero es útil cuando necesitamos tomar una lista de valores como los de un archivo y juntarlos para crear una entrada que luego se envía al usuario, o se almacena en su base de datos.

No Registrar Errores

¿Qué sucede cuando un usuario nos llama para decir que la aplicación está generando un error durante el envío del formulario? ¿Cómo sabemos qué entrada está utilizando el usuario? ¿Cómo sabemos si es el envío del formulario o algún otro evento se realizado correctamente? Para eso están los registros. Siempre se debe registrar los errores utilizando los estándares de excepción y una herramienta de registro de terceros o una personalizada. Hay muchas herramientas de terceros que nos brindan un análisis en profundidad, por lo que a menudo es más fácil usar una aplicación preexistente y confiable en lugar de crear la nuestra propia desde cero.

Usar una referencia como un valor o viceversa

Los programadores de C ++, y muchos otros lenguajes, están acostumbrados a controlar si los valores que asignan a las variables son simplemente valores o son referencias a objetos existentes. Sin embargo, en la programación de C Sharp, esa decisión la toma el programador que escribió el objeto, no el programador que crea una instancia del objeto y lo asigna a una variable. Este es un problema común para aquellos que intentan aprender programación en C #.

Si no sabemos si el objeto que estamos utilizando es un tipo de valor o un tipo de referencia, podríamos encontrarnos con algunas sorpresas.

Por ejemplo:

```
Puntero puntero1 = new Puntero(20, 30);
Puntero puntero2 = puntero1;
puntero2.X = 50;
Console.WriteLine(puntero1.X);       // 20 (¿Te sorprende?)
Console.WriteLine(puntero2.X);       // 50

Bolígrafo bolígrafo1 = new Bolígrafo(Color.Negro);
Bolígrafo bolígrafo2 = bolígrafo1;
bolígrafo2.Color = Color.Azul;
Console.WriteLine(bolígrafo1.Color);     // Azul ((¿Te sorprende?)
Console.WriteLine(bolígrafo2.Color);     // Azul
```

Como podemos ver, tanto los objetos **Puntero** como los objetos **Bolígrafo** se crearon exactamente de la misma manera, pero el valor de se **puntero1** se mantuvo sin cambios cuando **X** se asignó un nuevo valor de coordenadas a **puntero2**, mientras que el valor de **bolígrafo1** se modificó cuando se asignó un nuevo color **bolígrafo2**. Por lo tanto, podemos deducir que **puntero1** y **puntero2** cada uno contiene su propia copia de un objeto **Puntero**, mientras que **bolígrafo1** y **bolígrafo2** contienen referencias al mismo objeto **Bolígrafo**.

¿Pero cómo podemos saber eso sin hacer una prueba?

La respuesta es mirar las definiciones de los tipos de objetos, eso podemos hacerlo en Visual Studio colocando el cursor sobre el nombre del tipo de objeto y presionando F12:

```
public struct Puntero { ... }  // defines a “value” type
public class Bolígrafo { ... } // defines a “reference” type
```

Como se muestra arriba, en la programación de C #, la palabra clave **struct** se usa para definir un tipo de valor, mientras que la palabra clave **class** se usa para definir un tipo de referencia.

Si se depende de algún comportamiento que difiera entre el valor y los tipos de referencia, como la capacidad de pasar un objeto como parámetro del método y hacer que ese método cambie el estado del objeto, asegúrese de que está tratando con el Tipo correcto de objeto para evitar problemas de programación de C #.

Malentendido de los valores predeterminados para las variables sin inicializar

En C #, los tipos de valor no pueden ser nulos. Por definición, los tipos de valor tienen un valor, e incluso las variables no inicializadas de los tipos de valor deben tener un valor. Esto se llama el valor predeterminado para ese tipo. Esto lleva al siguiente resultado, generalmente inesperado, al verificar si una variable no está inicializada:

```
class Program
{
    static Puntero puntero1;
    static Bolígrafo bolígrafo1;
    static void Main(string[] args)
    {
        Console.WriteLine(bolígrafo1 == null);   // Cierto
        Console.WriteLine(puntero1 == null);    // Falso
}
```

¿Por qué **puntero1** no es nulo? La respuesta es que **Puntero** es un tipo de valor, y el valor predeterminado para un **Puntero** es (0,0), no nulo. No reconocer esto es un error muy fácil y común de cometer en C #.

Muchos de los tipos de valor tienen una propiedad **IsEmpty** que puede verificar para ver si es igual a su valor predeterminado:

```
Console.WriteLine(puntero1.IsEmpty);        // True
```

Cuando estamos verificando si una variable se ha inicializado o no, nos debemos asegurar de saber qué valor tendrá una variable no inicializada de ese tipo de manera predeterminada y no confíar en que sea nula.

Uso de métodos de comparación de cadenas inapropiados

Hay muchas formas diferentes de comparar cadenas en C #.

Aunque muchos programadores utilizan el operador == para la comparación de cadenas, en realidad es uno de los métodos menos deseables, principalmente porque no se especifica explícitamente en el código qué tipo de comparación se desea.

La forma preferida de probar la igualdad de cadenas es con el método **Equals**:

```
public bool Equals(string value);
public bool Equals(string value, StringComparison comparisonType);
```

El primer método es en realidad lo mismo que usar el operador ==, pero tiene la ventaja de que se aplica explícitamente a las cadenas. Realiza una comparación ordinal de las cadenas, que es básicamente una comparación byte por byte.

En muchos casos, este es el tipo de comparación que deseamos, especialmente al comparar cadenas cuyos valores se configuran mediante programación, como nombres de archivos, variables de entorno, atributos, etc. La única desventaja de usar el método **Equals** sin una comparisonType es que alguien que esté leyendo el código puede no saber qué tipo de comparación estamos haciendo.

Sin embargo, el uso del método **Equals** que incluye una comparisonType cada vez que compara cadenas, no solo hará que el código sea más claro, sino que también nos hará pensar explícitamente qué tipo de comparación necesitamos hacer. Esto es algo que vale la pena hacer, porque incluso si el inglés no ofrece muchas diferencias entre las comparaciones sensibles y culturales, los otros idiomas ofrecen mucho, e ignorar la posibilidad de otros idiomas nos puede generar numerosos errores. Por ejemplo:

```
string s = "Stranges";

// Salida False:
Console.WriteLine(s == "strangeß");
Console.WriteLine(s.Equals("strangeß "));
Console.WriteLine(s.Equals("strangeß ", StringComparison.Ordinal));
Console.WriteLine(s.Equals("strangeß ", StringComparison.OrdinalIgnoreCase));

// Salida True:
Console.WriteLine(s.Equals("strangeß ", StringComparison.CurrentCulture));
Console.WriteLine(s.Equals("strangeß ",
StringComparison.CurrentCultureIgnoreCase));
```

La práctica más segura es proporcionar siempre un parámetro comparisonType al método **Equals**. Aquí hay algunas pautas básicas:

- Al comparar cadenas que fueron introducidas por el usuario, o que se mostrarán al usuario, usar una comparación sensible a la cultura: **CurrentCulture** o **CurrentCultureIgnoreCase**
- Al comparar cadenas programáticas, usar la comparación ordinal: **Ordinal** o **OrdinalIgnoreCase**
- **InvariantCulture** y **InvariantCultureIgnoreCasegeneralmente** no se deben usar, excepto en circunstancias muy limitadas, porque las comparaciones ordinales son más eficientes. Si es necesaria una comparación de la cultura, por lo general se debe realizar contra la cultura actual u otra cultura específica.

Además del método **Equals**, las cadenas también proporcionan el método **Compare**, que nos ofrece información sobre el orden relativo de las cadenas en lugar de solo una prueba de igualdad. Este método es preferible a los operadores **<, <=,** > y >=, por las mismas razones que hemos visto anteriormente, para evitar problemas C #.

Utilizar el tipo de colección incorrecto para la tarea en cuestión

C # ofrece una gran variedad de objetos de colecciones, siendo el siguiente sólo una lista parcial:

- Array
- ArrayList
- BitArray
- BitVector32
- Dictionary<Tk,Tv>
- HashTable
- HybridDictionary
- List<T>
- NameValueCollection
- OrderedDictionary
- Queue
- Queue<T>
- SortedList
- Stack
- Stack<T>
- StringCollection
- StringDictionary.

Es bueno tomar un poco de tiempo adicional para investigar y elegir el tipo de colección óptimo para nuestro propósito. Es probable que resulte en un mejor rendimiento y menos espacio para el error.

Si hay un tipo de colección específicamente orientado al tipo de elemento que tenemos (como string o bit), inclínate hacia el uso de ese primero. La implementación es generalmente más eficiente cuando está dirigida a un tipo específico de elemento.

Para aprovechar los tipos de seguridad de C #, por lo general, deberíamos preferir una interfaz genérica sobre una no genérica. Los elementos de una interfaz genérica son del tipo que especifica cuando declara su objeto, mientras que los elementos de las interfaces no genéricas son de tipo objeto. Cuando se utiliza una interfaz no genérica, el compilador de C # no puede verificar el código con el tipo. Además, cuando se trata de colecciones de tipos de valores primitivos, el uso de una colección no genérica resultará en una operación de **boxing/unboxing** de esos tipos, lo que puede resultar en un impacto negativo significativo en el rendimiento cuando se compara con una colección genérica del tipo apropiado.

Otro problema común de C # es escribir tu propio objeto de colección. Eso no quiere decir que nunca sea apropiado, pero con una selección tan completa como la que ofrece .NET, probablemente puedas ahorrar mucho tiempo utilizando o extendiendo uno que

ya existe. En particular, la Biblioteca de colecciones genéricas de C5 para C # y CLI ofrece una amplia gama de colecciones adicionales "listas para usar", como estructuras de datos de árbol persistentes, colas de prioridad basadas en queues, listas de matrices indexadas hash, listas vinculadas y mucho más.

FIN